SQLBase

SQL Application Programming Interface Reference

20-2111-1005

Trademarks

Copyright

SQL Application Programming Interface
20-2107-1005
November 1997

Contents

Preface

The *SQL Application Programming Interface* provides information about each SQL Application Programming Interface (SQL/API) function.

This preface describes the following information:

- Who should read this manual.
- The organization of this manual.
- The documentation format.
- The notation conventions used in this manual.
- Related publications.

Who should read this manual

The *SQL Application Programming Interface Reference* is written for application developers using Centura's SQL Application Programming Interface (SQL/API) to write programs that access one or more databases.

This manual assumes you:

- Know how to program in the C language.
- Have some knowledge of relational databases and SQL.

Summary of chapters

This manual is organized in the chapters in the table below. There is also an index and glossary.

1	Introduction to the SQL/API	Provides a context for the SQL/API, lists the components of the product, and shows you how to compile, link, and run programs.
2	SQL/API Concepts	Explains the basics of the SQL/API.
3	Using the SQL/API	Uses flowcharts and code examples to show you how to use SQL/API functions.
4	SQL/API Functions by Category	Groups the SQL/API functions by functional category.
5	SQL/API Function Reference	Provides the syntax, a description, and an example for each SQL/API function.

Notation conventions

The table below show the notation conventions that this manual uses.

Notation	Explanation
You	A developer who reads this manual
User	The end-user of applications that you write
bold type	Menu items, push buttons, and field names. Things that you select. Keyboard keys that you press.
Courier 9	Builder or C language code example
SQL.INI MAPDLL.EXE	Program names and file names
Precaution	**Warning:**
Vital information	**Important:**

<table>
<tr><th>Notation</th><th>Explanation</th></tr>
<tr><td>Supplemental information</td><td>Note:</td></tr>
<tr><td>Alt+1</td><td>A plus sign between key names means to press and hold down the first key while you press the second key</td></tr>
<tr><td>TRUE
FALSE</td><td>These are numeric boolean constants defined internally in Builder:
<table>
<tr><th>Constant</th><th>Value</th><th>Meaning</th></tr>
<tr><td>TRUE</td><td>1</td><td>Successful, on, set</td></tr>
<tr><td>FALSE</td><td>0</td><td>Unsuccessful, off, clear</td></tr>
</table></td></tr>
</table>

Other helpful resources

Centura Books Online. The Centura document suite is available online. This document collection lets you perform full-text indexed searches across the entire document suite, navigate the table of contents using the expandable/collapsible browser, or print any chapter. Open the collection by selecting the Centura Books Online icon from the **Start** menu or by double-clicking on the launcher icon in the program group.

Online Help. This is an extensive context-sensitive online help system. The online help offers a quick way to find information on topics including menu items, functions, messages, and objects.

World Wide Web. Centura Software's World Wide Web site contains information about Centura Software Corporation's partners, products, sales, support, training, and users. The URL is http://www.centurasoft.com.

To access Centura technical services on the Web, go to http:/www.centurasoft.com/support. This section of our Web site is a valuable resource for customers with technical support issues, and addresses a variety of topics and services, including technical support case status, commonly asked questions, access to Centura's Online Newsgroups, links to Shareware tools, product bulletins, white papers, and downloadable product updates.

For information on training, including course descriptions, class schedules, and Certified Training Partners, go to http://www.centurasoft.com/training.

Send comments to...

Anyone reading this manual can contribute to it. If you have any comments or suggestions, please send them to:

Technical Publications Department
Centura Software Corporation
975 Island Drive
Redwood Shores, CA 94065

or send email, with comments or suggestions to:

techpubs@centurasoft.com

Chapter 1

Introduction to the SQL/API

This chapter describes the SQL/API and provides the following information:

- Description of SQL/API components
- Compiling and linking SQL/API applications
- Running a SQL/API application

About the SQL/API

Centura's SQL/API (Application Programming Interface) is a set of functions that you can call to access a database using Structured Query Language (SQL). Using these functions allows you to interface with a database through a procedural language, such as C.

You embed SQL/API functions within your program. Some functions specify SQL commands while other functions specify non-SQL database activities. After you write your application program, you compile it and link it with a Centura C/API library. You then can access database servers such as SQLBase.

The programs you write with the SQL/API are *client* (front-end) applications that connect to a backend database *server*.

Why use the SQL/API?

Using SQL commands is useful to define, manipulate, control, and query data in a relational database. However, SQL is not a programming language. Using the SQL/API functions to call SQL commands gives you the following features which plain SQL commands do not:

- Procedural logic
- Extensive data typing
- Variables

Using the API functions to develop a client application that uses SQL enables you to use SQL without giving up the power and flexibility of the programming language.

Other Centura SQLBase interfaces

The following SQLBase interfaces are available to assist you in creating your application.

SQLBase++: This is a class library that lets you write object-oriented C++ programs to access SQLBase. Read the following for more detailed information:

- README.WRI file in the CGSQL directory
- SQLBase++ Help System (CSQLHELP.HLP)

SQLTalk: This is an interactive user interface for SQL. Using this interface you can call SQL commands directly. For information on using SQLTalk, read the *SQLTalk Language Reference* manual.

How SQL/API applications access SQLBase

A SQL/API client application can access either a *local* SQLBase database engine/server or a *remote* SQLBase database server. Local means that the client application and the database engine or server run on the same machine, and remote means that they run on different machines.

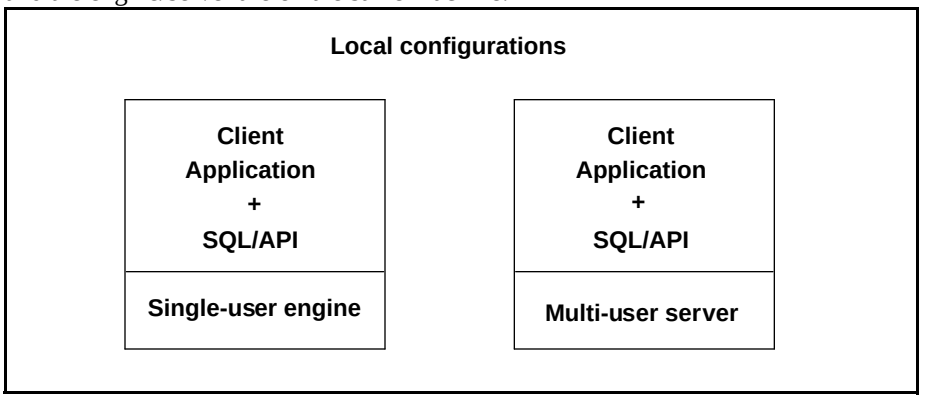
Single-user *engines* reside on client machines, just as applications do. Single-user engines allow only one application to connect to them at a time.

Multi-user *servers* can reside on the same machines as client applications or on different machines. Servers allow multiple applications to connect to them simultaneously and they can support multiple network protocols at the same time.

Refer to the *Communications* chapter of the *Database Administrator's Guide* for detailed information on single-user engines and multi-user servers.

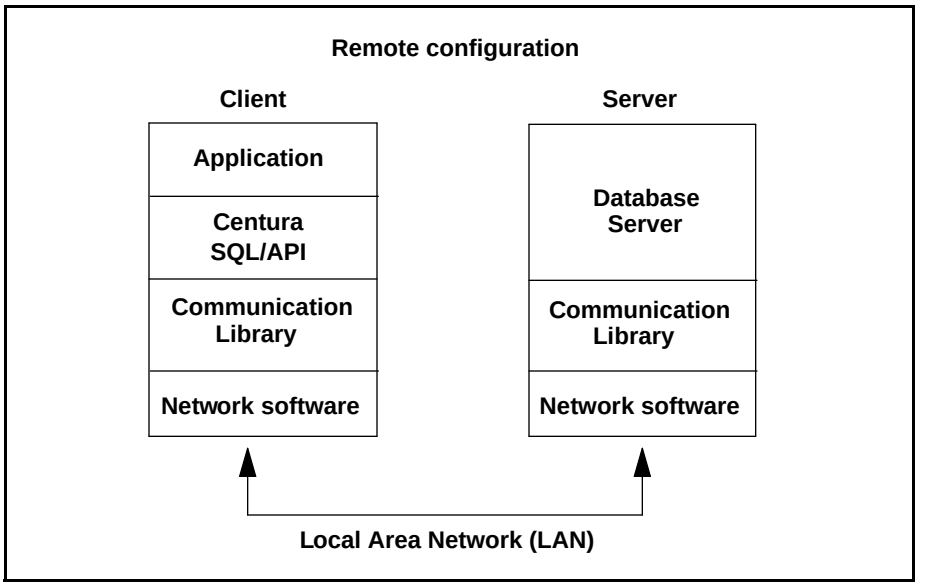
Local configuration

The following diagram shows *local configurations*, where both the client application and the engine/server are on the same machine.



Remote configuration

The following diagram shows a *remote configuration*, where the client and server are both on different machines connected by a Local Area Network (LAN).



In the remote configuration, there are communication libraries on both the client and server machines. Communication libraries provide network protocol-specific support so that client applications can communicate with database servers.

Refer to the *Communications* chapter of the *Database Administrator's Guide* for detailed information on communication options and libraries.

SQL/API components

The files listed below are components of the SQL/API.

sqlapinw.nlm

The NetWare Loadable Module that interfaces between a NetWare SQL/API application and a database server.

errsql.h

An include file that contains defines for all return codes.

qsiext.h

This file contains structure definitions and defined constants used to interface with a SQLBase server and returns extended OSI information.

sql.h

A file that contains definitions for data types (typedefs), codes for each type of SQL command, and system defaults. Include this file in a C source program that uses SQL/API functions. This file contains two macros to accommodate both 16- and 32-bit programs:

- SQL_32BITTARG for 32-bit programs
- SQL_16BITTARG for 16-bit programs

sql32.h

A file that contains definitions for data types (typedefs), codes for each type of SQL command, and system defaults. You can include this file in a 32-bit C source program that uses SQL/API functions instead of *sql.h*.

sqlapiw.lib

The library that interfaces between a Microsoft Windows SQL/API application and a database server.

sqlsrv.h

A file that contains structure and constant definitions used to interface with a database server.

sqlwntm.lib

A library that interfaces between a Windows 95 or Windows NT SQL/API application (built with Microsoft's NT tools) and a database server.

Example programs

Centura supplies the following example programs with the SQL/API. These programs are referenced throughout this manual to illustrate the use of the SQL/API. Read Running the example SQL/API function for details on running the examples under your platform.

ex01.c

Performs a simple database connection using the standard defaults.

ex02.c

Performs a database connection using literals for the connect string. (Create a Payroll database and run the *grant.sql* script in SQLTalk first.)

ex03.c

Performs a database connection using variables for the connect string. (Run the *grant.sql* script in SQLTalk first.)

ex04.c

Compiles and executes a SQL command.

ex05.c

Compiles and executes a SQL command in one function call.

ex06.c

Demonstrates transaction control with the COMMIT and ROLLBACK commands. (Run the *account.sql* script in SQLTalk first.)

ex07.c

Demonstrates a common error routine. (Run the *account.sql* script in SQLTalk first.)

ex08.c

Performs a simple fetch. (Run the *emp.sql* script in SQLTalk first.)

ex09.c

Performs a fetch from multiple columns. (Run the *emp.sql* script in SQLTalk first.)

ex10.c

Demonstrates the describe operation. (Run the *emp.sql* script in SQLTalk first.)

ex11.c

Performs an insert with bind variables. (Reads from the *data* file.)

ex12.c

Performs data binding by name. (Run the *emp.sql* script in SQLTalk first.)

ex13.c

Writes LONG VARCHAR data. (Reads from the *sayings.1* file.)

ex14.c

Reads LONG VARCHAR data. (You must compile and execute *ex13.c* first.)

ex15.c

Uses the *sqlcpy* function. (Run the *emp.sql* script in SQLTalk first.)

ex16.c

Demonstrates the use of multiple cursors. (Run the *bonus.sql* script in SQLTalk first.)

ex17.c

Performs backup and restore operations.

ex18.c

Demonstrates the use of result set and restriction modes.

ex19.c

Demonstrates most of the features of the SQL/API. (Reads from the *sample.txt* file.)

ex20.c

Fetches data. (Reads from the *data* file.)

ex21.c

Demonstrates the use of the SQL/API with Microsoft Windows.

ex22.c

Uses the *sqlgsi* function.

ex23.c

Demonstrates how to execute stored commands and procedures from SQL/API.

ring.c

Displays numbers that you enter in database-internal numeric format.

sqlcbv.c

Uses the *sqlcbv* (Clear Bind Variables) function.

sqlclf.c

Uses the *sqlclf* (Change process activity Log File) function.

sqlcre.c

Uses the *sqlcre* (CREate database) function.

sqldbn.c

Uses the *sqldbn* (DataBase Names) function.

sqlded.c

Uses the *sqlded* (DEinstall Database) function.

sqldel.c

Uses the *sqldel* (DELete database) function.

sqldro.c

Uses the *sqldro* (DiRectory Open) function.

sqldsc.c

Uses the *sqldsc* (DeSCribe column in SELECT list) function.

sqldsv.c

Uses the *sqldsv* (Disconnect from SerVer) function.

sqlfer.c

Uses the *sqlfer* (Full ERror message) function.

sqlfgt.c

Uses the *sqlfgt* (GeT File from server) function.

sqlims.c

Uses the *sqlims* (Input Message Size) function.

sqlind.c

Uses the *sqlind* (INstall Database) function.

sqlnrr.c

Uses the *sqlnrr* (Number of Rows in Result set) function.

sqloms.c

Uses the *sqloms* (Output Message Size) function.

sqlscp.c

Uses the *sqlscp* (Set Cache Pages) function.

sqltio.c

Uses the *sqltio* (TIme Out) function.

test.c

Connects to a database, creates and populates a table, and performs SELECTs and UPDATEs on the table.

testwin.c

Sample Microsoft Windows program.

xdfunc.c

Uses the sqlxdp (eXtended convert Picture to Date).

Support files

The following files accompany the example programs.

account.sql

Creates the savings and checking tables for *ex06.c* and *ex07.c*.

bonus.sql

Creates the emp and bonus tables for *ex16.c*.

data

Company data for *ex11.c* and *ex20.c*.

emp.sql

Creates, indexes, and populates the emp table for *ex08.c*, *ex09.c*, *ex10.c*, *ex12.c* and *ex15.c*.

examples

A listing of the example programs.

grant.sql

Grants connect authority to a user for *ex02.c* and *ex03.c*.

sample.txt

LONG VARCHAR data file for *ex19.c*.

testwin

The makefile for *testwin.c* which builds *testwin.exe*.

testwin.def

The linker definitions file for *testwin.c*. It specifies the stack size, executable type, program name, and exported functions.

testwin.exe

Executable file created from *testwin.c*.

testwin.rc

Resource file for *testwin.c*.

Compiling, linking, and running applications

This section describes how to compile, link, and run applications with embedded SQL/API function calls on the various client platforms. For details on running the example functions included in the SQLBase software, read *Compiling, linking, and running example programs* on page *1-17*.

Running a SQL/API application

1. Compile the program with the compiler of your choice. Read the information in the following sections that pertain to your platform.
2. Link the program with the appropriate Centura SQL/API library for your platform. Read the next section Environment variables to include to choose the correct library for your platform.

Note: Some compilers allow you to compile and link a program in one step.

---

3. Confirm that the database engine or server that you plan to access is running. If you plan to access a remote database server, make sure that the network software on both the client and server machines is loaded and running.
4. Start a router program on the client machine, if necessary.
5. Make sure that the executable file can find and access the database.
6. Run the executable program.

Environment variables to include

Before compiling any SQL/API application, you need to set two environment variables:

- INCLUDE
- LIB

INCLUDE identifies the directory or directories where header files such as *sql.h* or *sql32.h* reside. For details on available header files, read *Header files for 16-bit and 32-bit programs* on page *1-16*.

LIB identifies the directory where the SQL/API library resides. The SQL/API libraries are listed below by platform:

Platform	SQL/API library
Windows	sqlapiw.lib
Windows NT and Windows 95	sqlwntm.lib (used for applications built with Microsoft's NT tools)
NetWare	sqlapinw.nlm

Windows 16-bit programs

This section describes how to compile, link, and run Microsoft Windows C programs that contain SQL/API functions for the Windows 3.x platform. Centura recommends compiling your application with the /w3 option so that the compiler displays *all* warning messages.

After compiling your application, either statically link it with *sqlapiw.lib* or reference IMPORTS through the *.def* file.

Note: *sqlapiw.lib* is model independent.

Microsoft restricts you from calling *sqlapiw.dll* functions from a DLL in the LibMain entry point, either directly or indirectly. This is because MS Windows has not been fully initialized and therefore, has not yet created a message queue for the task. The communications interface requires that initialization be complete before functions in *sqlapiw.dll* are called.

For more information about LibMain, refer to the Microsoft Developer's CD-ROM documentation.

The following two examples illustrate the direct and indirect calling of *sqlapiw.dll* functions:

```
/* Direct; this will not run. */

LibMain ()
{
  sqlcnc ( &cur, “demo”, 0 );
}
and:
/* Indirect; this will not run either. */

LibMain ()
{
  connect ();
}

void FAR PASCAL connect ()
{
  sqlcnc ( &cur, “demo”, 0 );
}
```

Running Windows 3.x

If your application plans to access the local multiple-user Windows engine (*dbwservr.exe)*, follow these steps:

1. Start MS Windows.
2. Start *dbwservr.exe*.
3. Start your application.

If your application plans to access a remote database server, follow these steps:

4. Start MS Windows.
5. Start your application.

Windows 32-bit programs

This section describes how to compile, link, and run Microsoft Windows C programs that contain SQL/API functions for the Windows 95 and Windows NT platforms.

When building 32-bit applications to run on Windows NT or Windows 95, use the Microsoft toolset (compiler, linker, librarian, and so on) that accompanies the Windows NT SDK or use a Windows NT-compatible or Windows 95-compatible toolset.

After compiling your program, link it with *sqlwntm.lib*.

Use the *sqlwntm.lib* library with applications built with Microsoft's NT or 95 tools. This library resolves references to the SQL/API functions at link time into the *sqlwntm.dll* library.

Running Windows 95 and Windows NT

If your application plans to access the local multiple-user Windows engine (*dbntsrv.exe)*, follow these steps:

1. Start Windows 95 or Windows NT.
2. Start *dbntsrv.exe*.
3. Start your application.

If your application plans to access a remote database server, follow these steps:

4. Start Windows 95 or NT.
5. Start your application.

Windows NT character-based application

To build a character-based SQL/API application under Windows NT, make sure that you have the Windows NT SDK installed on your machine. The install process sets the environment variable INCLUDE to the following value:

```
INCLUDE=c:\mstools\h
```

This assumes that the SDK has been installed on drive C:.

You may also need to do the following:

1. Include an additional setting for this variable:

   ```
   INCLUDE=c:\mstools\h;c:\mstools\h\sys
   ```

2. Add search path(s) for your own C header files, including ***sql.h***. All modifications to the INCLUDE environment variable should be made from the Control Panel's System icon.
3. Ensure that the environment variable LIB includes the directory where the library sqlwntm.lib is located.

Once the variables have been set properly, use the following command to compile your program. This example compiles a sample program called *example.c*:

```
cl386 -c -Gs -Od -Zpe -DSQL_32BITTARG=1 -DSTRICT -W1 -D_X86
   example.c
```

This creates the object file *example.obj*.

The program can be linked with the SQL/API library *sqlwntm.lib* using the following command:

```
link32 -debug:full -debugtype:cv -subsystem:console -
       entry:mainCRTStartup -map:example.map example.obj
       libc.lib kernel32.lib sqlwntm.lib -out:example.exe
```

Both *cl386* and *link32* are documented in the *Tools User's Guides* of the Microsoft Win32 Software Development Kit.

C programs for Netware

This section describes how to build a NetWare Loadable Module (NLM) from a C program containing SQL/API functions and how to compile and link your SQL/API applications to the *sqlapinw.nlm* library.

Building the SQL/API NLM

An NLM is a program that you can load into or unload from the NetWare server memory while the NetWare server is running. When loaded, an NLM is part of the NetWare operating system. When unloaded, an NLM releases the memory and resources that were allocated for it.

A SQLBase client NLM is version-independent; it does not matter whether you build it under NetWare 3.x or 4.x. It also can communicate with either the SQLBase Server for NetWare 3.x or 4.x.

You can build an NLM from a C program created on any platform. You can also build an NLM from any of the SQL/API examples accompanying this product to run on NetWare.

The SQL/API import NLM name is *sqlapinw.nlm*. The NetWare communication library is *spxdll.nlm* (if you are using Novell's NETX environment), *tlidll.nlm* (if you are using TLI-TCP/IP support for both NetWare 3.x and 4.x), and *spxdll40.nlm* (if you are using Novell's DOS/VLM).

Compiling and linking SQL/API applications to the NLM

Use the Watcom C/C++ compiler to compile and link your SQL/API applications to the *sqlapinw.nlm* library. The following example compiles a program called *myfile.c* to create *myfile.obj*:

```
wcc386p -3s myfile.c
```

To debug the program, add the -d2 switch to the compile command.

Edit the *sql.h file* with the following modifications:

- Set the #define SQL_32BITTARG to 1.

- Add the following line:

```
#define_stdcall
```

The following command line links the C program and builds the NLM. This example links the *myfile.obj* file to create *myfile.nlm*:

```
wlinkp myfile.lnk
```

To link the program to the NetWare, you must supply a *.lnk* or *.def* file. See the Novell and Watcom documentation listed at the end of this section for information on how to build a *.lnk* or *.def* file.

The following example shows a sample link file called *mytest.lnk*.

```
#BEGIN MYTEST.LNK SAMPLE#
form NOV NLM 'Centura SQLAPI Test NLM'
name \testnlm\mytest#name

#specify map file and version options
option map=\testnlm\mytest.map
option version=1.00

debug NOVELL  #Use this to debug in NetWare
              #Internal debugger only
#debug ALL    # Use this to debug under Watcom
              #video also
option stack=60k
#option caseexact
file \testnlm\mytest.obj

#import libraries
import @\sql\watc\h\clib.imp
#import @\sql\watc\h\mathlib.imp

#Import all SQL/API functions from sqlapinw.nlm that
#will be called in the mytest.nlm program
import sqlcnc
import sqldis
import sqlcom
import sqlexe

#END MYTEST.LNK SAMPLE#
```

For more information, see the following documentation as appropriate:

- Netware NLM Library reference and related documentation.
- Watcom C/C++ compiler documentation.
- *Netware 4.0 NLM Programming* from Novell Press.

Running a SQL/API NLM

To run a SQL/API NLM, first load the following NLMs at the colon according to the listed order:

```
:load dll.nlm
:load dfs.nlm (or) :load dfd.nlm
:load spxdll.nlm (or) :load tlidll.nlm
:load sqlapinw.nlm
```

You can also load the NLMs together as a batch (or *.ncf*) file.

After loading these NLMs, load the SQLAPI application NLM as follows

```
:load myvolume:\mypath\myfile.NLM.
```

Header files for 16-bit and 32-bit programs

SQLBase provides two header files for client applications. The *sql32.h* accommodates 32-bit programs; generally, you should use this header file to compile 32-bit programs. The *sql.h h*eader file accommodates both 16- and 32-bit programs with 2 macros:

- SQL_32BITTARG for 32-bit programs
- SQL_16BITTARG for 16-bit programs

These macros are case sensitive, and must be called in upper-case.

Be aware that by default, SQL_16BITTARG is set to 1 (true), and SQL_32BITTARG is set to 0 (false) in *sql.h*. To compile 32-bit programs with *sql.h*, you must set SQL_32BITTARG to 1. You can do this in several ways:

- Use the -D switch on the compiler command line. For example:

  ```
  -DSQL_32BITTARG=1
  ```

- Define the macro in the user code, and then include *sql.h*. For example:

  ```
  #define SQL_32BITTARG 1
  #include “sql.h”
  ```

- Include the new header file *sql32.h*. Include this new header file in your program instead of the regular *sql.h*:

  ```
  #include “sql32.h”
  ```

The Centura *sql.h* file is compatible with all of Microsoft’s C/C++ compilers (versions 5.1 and up) and with the latest version of WATCOM’s C/C+ compiler. If you are using a compiler other than either of these two, make certain that it uses the flat memory model and supports the __stdcall calling convention.

Compiling, linking, and running example programs

This section describes how to compile, link, and run the example SQL/API functions included in the SQLBase software for the Windows platforms. The example functions that are available with SQLBase are listed under *SQL/API components* on page *1-4*.

Running example programs with Windows 3.x

1. Compile the program as a QuickWin application. For information on compiling QuickWin applications, see the documentation for your specific compiler.
2. Link the program with the *sqlapiw.lib* library. Read *Environment variables to include* on page *1-11* for details on linking the library.
3. Run the executable program.

Running example programs with Windows 95 and Windows NT

1. Create a project of type *Console Application* in Microsoft Visual C (MSVC) 2.0 or later.
2. Link the program with the *sqlwntm.lib* library. Read *Environment variables to include* on page *1-11* for details on linking the library.
3. Compile the example programs (which are DOS programs).
4. Run the executable program.

Chapter 2

Data Types

This chapter describes the three different kinds of data types:

- **Internal database data types** are generic data types. They specify how SQLBase stores data internally. The *sqldes* and *sqlgdi* functions are the only SQL/API functions that reference these data types.
- **Program data types** map to C data types.
- **External data types** map to non-Centura database data types. The *sqldsc* and *sqlgdi* functions are the only SQL/API functions that reference these data types.

Internal database data types

SQLBase stores data internally as one of the following data types. The internal data types are defined in *sql.h*:

Internal Data Type	Description
SQLDBOO	Boolean
SQLDCHR	Character
SQLDDAT	Date/time
SQLDDTE	Date (only)
SQLDHDL	SQL Handle
SQLDLON	Long
SQLDNUM	Numeric
SQLDTIM	Time (only)

Note: Internal data types SQLDBOO and SQLDHL are not stored in the database, but are parameters to stored procedures.

Character data

SQLBase stores character data (including LONG VARCHAR data) as variable-length strings.

For example, if you insert a 20-character string into a column defined as CHAR(30) or VARCHAR (30), SQLBase stores only 20 characters. It does not pad the string to make it 30 characters long.

Numeric data

SQLBase stores numeric data in base 100 floating point format, and maintains precision and scale. Precision refers to the total number of digits while scale refers to the number of digits to the right of the decimal point.

The length of a stored numeric value varies, and can be from 1 to 12 bytes.

Numeric data is cast on input and output to conform to the restrictions of the external data type.

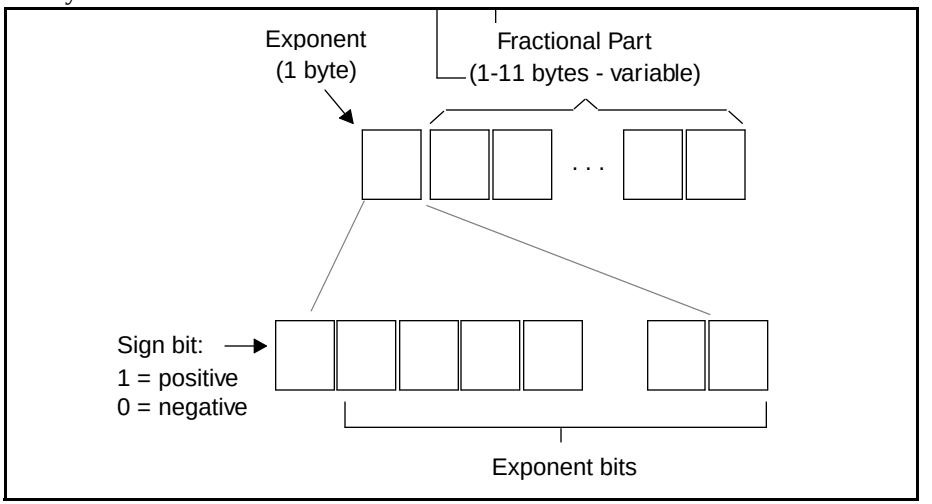
Internal numeric functions

You can use the functions listed below to manipulate numeric data stored in its internal format:

Function	Description
sqlxad	eXtended ADd - Adds two SQLBase internal numbers.
sqlxcn	eXtended CoNvert - Converts a character string to a SQLBase internal number.
sqlxdv	eXtended DiVide - Divides a SQLBase internal number by another SQLBase internal number.
sqlxml	eXtended MuLtiply - Multiplies two SQLBase internal numbers.
sqlxnp	eXtended Number to Picture - Converts a SQLBase internal number to a string using a picture format.
sqlxsb	eXtended SuBtract - Subtracts one SQLBase internal number from another and puts the result in a third SQLBase internal number.

Byte format

The byte format is:



The first byte contains the sign bit and the exponent. The sign bit is the high order bit (80 hexadecimal, 10000000 binary). If this bit is set, the final number is positive; otherwise the number is negative.

The remaining 7 bits of the first byte store the exponent in base 100. The exponent indicates how many bytes in the fractional part to shift the decimal point to the right (or to the left for negative exponents) to get the final number.

The exponent is biased by 64 (40 hexadecimal, 01000000 binary) and ranges from 0 to 127 as a biased number or -64 to 63 unbiased.

For example:

Biased Exponent	Actual exponent (unbiased)
70	6
65	1
64	0
63	-1
58	-6

In the fractional part, there may be 0 to 11 bytes. Each byte contains a binary value of 0 to 99. Each byte represents a base 100 number.

Trailing digits are truncated for positive numbers.

Negative numbers

For negative numbers, the following conversion is applied to the positive number representation:

1. Take the 1's complement of the exponent byte.
2. Take the 100's complement of the fractional part.
3. Add a byte containing 101 to the end of the fractional part.

These steps ensure that negative numbers sort properly.

Here are some examples of internal numeric storage representation:

Number	Internal Representation
123	194,01,23
0.0123	192,01,23
12.3	193,12,30
-123	61,99,77,101

Number	Internal Representation
-0.0123	63,99,77,101
-12.3	62,88,70,101

Here are the steps followed for the fifth example above:

-0.0123 63,99,77,101

1. The binary value of the exponent (63) is 00111111.
2. The high-order bit is zero, which means that the final value is negative and to invert the binary value:

   ```
   11000000
   ```

3. Strip the high-order bit:

   ```
   01000000
   ```

4. which is 64 in decimal. Subtract 64 from 64 and it equals zero, so the decimal point does not need to be shifted.
5. Drop the 101 and take the 100's complement of 99 and 77:

   ```
   01 23
   ```

6. The final value is negative (determined in step 2). The decimal point does not shift, so the final value is:

   ```
   -0.0123
   ```

Date and time data

SQLBase stores date and time data in the same format as for numeric data. The default display format of date and time data in the SQL/API is:

```
yyyy-mm-dd-hh.mi.ss.999999
```

where hours (hh) is based on a 24-hour clock.

Define the buffer that receives date and time data with a length of SQLSCDA (defined in *sql.h*).

Internal date/time functions

You can use the following functions listed below to manipulate date and time data stored in its internal format.

Function	Description
sqlxda	eXtended Date Add - Adds *n* days to a SQLBase internal date.
sqlxdp	eXtended Date to Picture - Converts a SQLBase internal date to a string using the specified picture format.
sqlxpd	eXtended Picture to Date - Converts a null-terminated string to a SQLBase internal date.

Program data types

Use program data types to define data within a SQL/API program.

When inserting data into a database, the program data type does *not* have to match an internal database data type. The SQL/API always tries to convert data in a program variable to the database data type. If the SQL/API cannot convert the data, it returns an error.

The program data types are defined in *sql.h:*

Program Data Type	Description
SQLPBUF	Character buffer
SQLPDAT	Internal datetime
SQLPDOU	Double
SQLPDTE	Date only
SQLPEBC	EBCDIC buffer
SQLPFLT	Float
SQLPLON	Long text string
SQLPLBI	Long binary buffer
SQLPLVR	Char/long varchar >254
SQLPNBU	Numeric buffer
SQLPNST	Numeric string

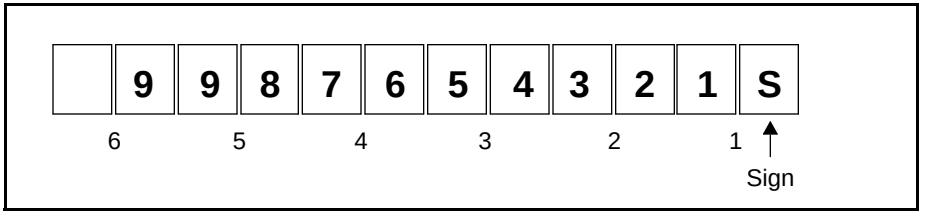
Program Data Type	Description
SQLPNUM	Internal numeric
SQLPSCH	Character
SQLPSIN	Integer
SQLPSLO	Long
SQLPSPD	Signed packed decimal
SQLPSSH	Short
SQLPSTR	String (null-terminated)
SQLPTIM	Time only
SQLPUCH	Unsigned character
SQLPUIN	Unsigned integer
SQLPULO	Unsigned long
SQLPUPD	Unsigned packed decimal
SQLPUSH	Unsigned short

Packed-decimal data types

You can retrieve packed-decimal data into a program. There are data types for unsigned packed decimal (SQLPUPD) and signed packed decimal (SQLPSPD).

If you use a packed decimal type, the data length is the maximum number of digits in the number. Each nibble (4 bits) of each byte holds one digit, except for the rightmost nibble which holds the sign (if requested).

For example, the number 9987654321 has a length of 6 bytes and appears in bytes as shown below.



The leftmost nibble is unused and the rightmost nibble contains a sign (if any).

To determine the number of bytes required:

```
Number of bytes required = (1 + number of digits)/2
```

If it divides evenly, the quotient is the length. If there is a remainder (modulo), add 1 to the quotient. For example, the number 9987654321 contains 10 digits:

```
(1 + 10)/2 = 5 modulo
```

It does not divide evenly, so add 1. The length is 6.

You need only specify the scale argument (number of decimal places) for the *sqlssb*, *sqlbnn*, and *sqlbnd* functions for a packed-decimal data type. If you are not using a packed-decimal data type with one of these functions, specify a 0 for the scale argument.

External data types

The external data types are defined in *sql.h:*

External Data Type	Description
SQLEBIN	BINARY
SQLEBOO	BOOLEAN
SQLECHR	CHAR
SQLEDAT	DATE
SQLEDEC	DECIMAL
SQLEDOU	DOUBLE
SQLEFLO	FLOAT
SQLEGPH	GRAPHIC
SQLEINT	INTEGER
SQLELBI	LONG BINARY
SQLELCH	CHAR >254
SQLELGP	LONG VAR GRAPHIC
SQLELON	LONG VARCHAR
SQLELVR	VARCHAR >254
SQLEMON	MONEY

External Data Type	Description
SQLESMA	SMALLINT
SQLETIM	TIME
SQLETMS	TIMESTAMP
SQLEVAR	VARCHAR
SQLEVBI	VAR BINARY
SQLEVGP	VAR GRAPHIC

Chapter 3

Using the SQL/API

This chapter uses flowcharts and code examples to show you how to use the SQL/API functions. This chapter does not attempt to provide details for each SQL/API function, but it does show the logic flow within a program.

The SQL/API functions are flexible and can be used in different ways. In the code examples, specific techniques are used to perform tasks (for example, using *for* and *while* loops). These techniques are only suggested solutions and you should not interpret them as the only or best way to perform a task.

This chapter refers to example programs that are on the installation diskette. See the *Example programs* section in Chapter 1 for a summary of these example programs. To run the example programs yourself, read *Compiling, linking, and running example programs* on page *1-17*.

Connect and close cursor

Note: This section applies to applications in which you are connecting cursors to a specific database that belong to a single transaction.

To create multiple, independent connections, SQLBase allows you to explicitly create multiple connection handles. For example, you can use connection handles for multiple transactions to the same database within an application, or for creating multi-threaded Win32 applications. For details on creating connection handles, read *Connection handles* on page *3-17*.

Before you can perform database operations in your application, you must connect the cursor to a specific database with a cursor handle (*sqlcnc*). The *sqlcnc* function returns a cursor handle which identifies an implicit connection to the database.

All cursors that you connect to this database belong to a single transaction and to the same implicit connection handle. Read *Cursors* on page *3-27* for more information.

You must disconnect the cursor connection to the database (*sqldis*) before you can exit from the program.

The example programs *ex01.c*, *ex02.c*, and *ex03.c* show how to connect to, and close a cursor from, a database. Here is *ex03.c*:

```
    #include "sql.h"
①   #include <stdio.h>

    main()
    {
②     SQLTCUR     cur=0;  /* SQLBase cursor number*/
③     SQLTRCD     rcd=0;  /* return code */
④     static char dbname[]="PAYROLL/BOSS/SECRET";
      /*
         CONNECT TO THE DATABASE
      */
⑤     if (rcd = sqlcnc (&cur,dbname,0))
      {
⑥       printf("FAILURE ON CONNECT %d\n",rcd);
        printf("Does the PAYROLL database exist?\n");
        printf(Has GRANT.SQL been run\n");
        return (1);
      }
      else
        printf("Connection Established \n");
      /*
        DISCONNECT CURSORS
      */
```

```
⑦    if (rcd = sqldis(cur))
       printf("FAILURE ON DISCONNECT %d\n", rcd);
     else
       printf("Disconnect Performed \n");
   }
```

1. You must include the support file *sql.h* in a program that calls the SQL/API functions.
2. Declare a cursor for the connection.
3. Declare a variable that will hold a return code for each execution of a SQL/API function.
4. Declare the name of the database that you want to connect to.
5. Call the *sqlcnc* function to connect to the database. If the call completes successfully, the cursor handle is returned in the first argument (*cur*). The cursor handle is opaque and you are not aware of its actual value, but you use it in other SQL/API functions to identify a specific connection to the database.

   The second argument is the connect string which can specify the database name, the username and the password. If you do not specify all three parameters (database name, user name, and password), their default values (DEMO, SYSADM, and SYSADM) are used.

   The third argument (length) is zero which means that the second argument points to a string that is null-terminated. The SQL/API will compute the actual length of the string.
6. If the function fails and returns a non-zero value, a user-defined error message (“FAILURE ON CONNECT”) is printed.
7. Call the *sqldis* function to close the cursor connection from the database. Always disconnect all cursors before exiting a program. The last *sqldis* function in a program causes an implicit commit by default. You can change the default setting using the *sqlset* function with the SQLPCCB parameter.

Server security

To perform administrative operations on a server, you must establish a connection to the database server itself and specify the server password (if one exists). This prevents unauthorized users from performing destructive operations on the server.

Define the server name by configuring the *servername* keyword in the server’s configuration file (*sql.ini*). A server name can be up to eight alpha-numeric characters, but it must start with a letter.

Define the server password by configuring the *password* keyword on the line immediately following the *servername* keyword entry. A password can be up to eight alpha-numeric characters.

Use the *sqlcsv* function to establish a server connection. This function requires a server name as input and returns a handle.

Use the *sqldsv* function to break a server connection. This function requires a server handle as input.

Compiling and executing SQL commands

Four things happen when SQLBase compiles a SQL command:

1. It parses the command. This step detects syntax errors and verifies the existence of database objects.
2. It performs a security check.
3. It determines the best access path. The system finds the indexes (if any) that provide the best access path to the data.
4. It translates the command into a series of executable modules.

The *sqlcom* function compiles a SQL command, and SQLBase stores the compiled command in the cursor work space. After compiling a command, you can execute it using the *sqlexe* function.

The *sqlcex* function compiles and executes a SQL command in one step. Use the *sqlcex* function for SQL commands which do not contain bind variables and which will only be executed once. For example, commands which you can compile and execute with *sqlcex* are data definition commands and data control commands such as CREATE, DROP, GRANT, and REVOKE.

Unless cursor-context preservation is on, when you COMMIT a transaction, SQLBase destroys compiled commands for all cursors that the program has connected to the database. This is true for both explicit and implicit COMMITs, including implicit COMMITs which occur when you have autocommit on.

If cursor-context preservation is off, a ROLLBACK (including a ROLLBACK caused by a deadlock) destroys all compiled commands. If cursor-context preservation is on, a ROLLBACK does not destroy compiled commands if both of the following are true:

- The application is in Release Locks (RL) isolation level
- No data definition language (DDL) operations were performed

The example programs *ex04.c* and *ex05.c* show how to compile and execute SQL commands.

Setting SELECT buffers

After you compile a SELECT command with *sqlcom*, you must set up areas within your application to receive the selected data. Do this with the *sqlssb* function.

You must call the *sqlssb* function once for each item in the SELECT list. For example, if you SELECT the columns EMP_NAME, EMP_NO, EMP_DOB, you need to call the *sqlssb* function three times.

You do not need to call the *sqlssb* function for LONG VARCHAR columns. The *sqlrlo* function identifies the receive buffer for a LONG VARCHAR.

The example programs *ex08.c* and *ex09.c* show how to use the *sqlssb* function.

Bind variables

In a SQL statement, you can use a bind variable to represent the value of a column. A bind variable indicates that data from a variable defined in your application will be bound (associated) to it each time you execute the SQL statement.

A bind variable name begins with a colon (:) and is followed by a number or string. For example:

```
SELECT * FROM BOOKS WHERE AUTHOR = :1
```
or:

```
SELECT * FROM BOOKS WHERE AUTHOR = :auth
```
Bind variables allow you to compile a SQL statement once and execute it repeatedly, each time substituting a new set of values in the bind variables.

Binding data

The *sqlbnd* function associates an alphanumeric bind variable in a SQL statement to a variable in your application. The *sqlbnn* function associates a numeric bind variable in a SQL statement to a variable in your application.

Bind functions for LONG VARCHAR columns are explained in the *LONG VARCHAR Handling* section later in this chapter.

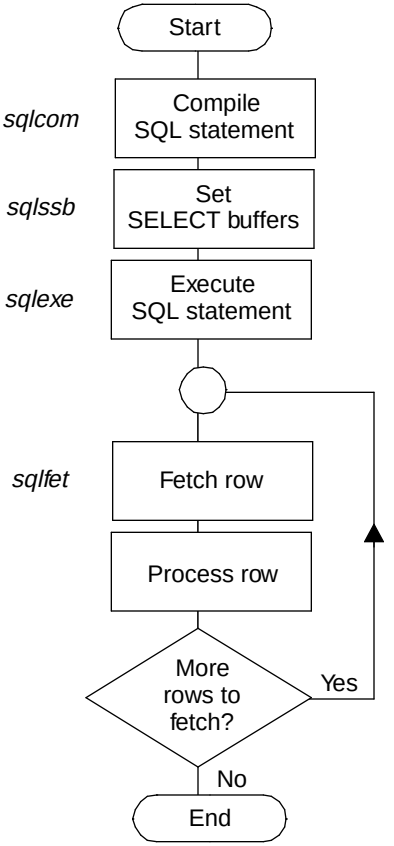
The example programs *ex12.c* and *ex16.c* show how to bind data.

Queries

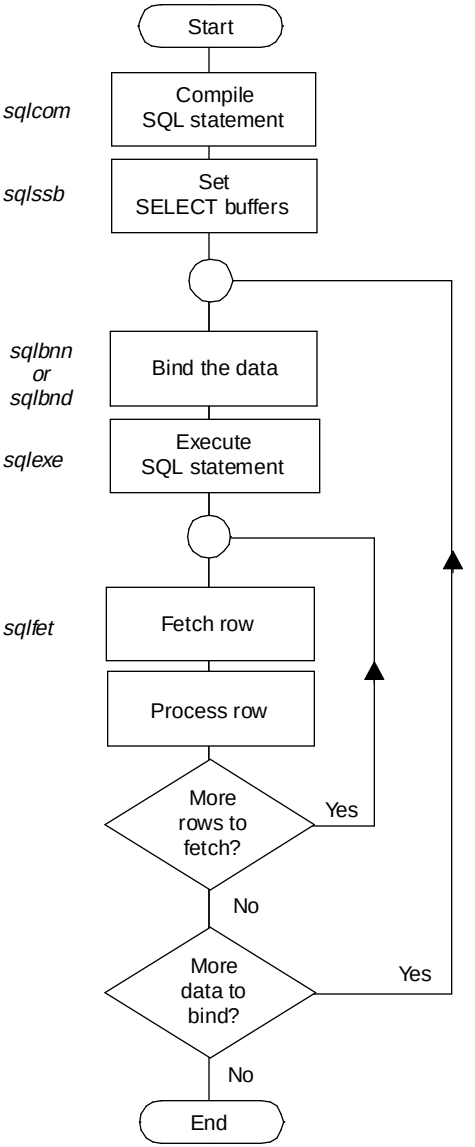
The following two flowcharts show the sequence of operations when performing a SELECT command. The first flowchart shows the sequence if you are *not* using bind variables, and the second flowchart shows the sequence if you *are* using bind variables.

In the first flowchart (a SELECT not using bind variables), note that you can call the *sqlssb* function before or after the *sqlexe* function. However, you must call the *sqlssb* function before the *sqlfet* function.

Access cycle for SELECT command without bind variables



Access cycle for SELECT command with bind variables



SELECT command without bind variables (ex20.c)

This example uses excerpts from *ex20.c* to show how to perform a SQL SELECT statement (without bind variables) using the SQL/API.

```
#include "sql.h"

...

SQLTCUR  cur = 0; /* Cursor number */
SQLTRCD  rcd = 0; /* Return code */

main()
{

...

SQLTDAP  cp;          /* Character pointer */
SQLTDAL  length;      /* Length */
SQLTPDL  pdl;         /* Program buffer length */
SQLTDDT  ddt;         /* Database data type */
SQLTPDT  pdt;         /* Program data type */
SQLTSLC  slc;         /* SELECT list column */
SQLTNSI  nsi;         /* Number of SELECT items */
char     line[200];   /* I/O line */
...
① static    char  selcom1[] ="SELECT A, D, C FROM X";
  /*SELECT command*/

  ...      /*Connect to database, create the table */

  pdt = SQLPBUF;  /* Set program data type of buffer */
  ...
  /* Compile the SELECT command */

② if (sqlcom(cur, selcom1, 0))
       failure("SELECT COMPILE");
  /* Get descriptive information about SELECT */

  cp = line;  /* Set pointer to input line */
③ if (sqlnsi(cur, &nsi))   /* Get # SELECT items */
    failure("GET NUMBER OF SELECT ITEMS");

      for (slc = 1; slc <= nsi; slc++) /* Get information */
                                       /* on each column */
    {
```

```
④      if (sqldes(cur, slc, &ddt, &pdl, /* Failure on */
                                        /* describe? */
            SQLNPTR, SQLNPTR, SQLNPTR, SQLNPTR))
          failure("SELECT DESCRIBE");
⑤      if (sqlssb (cur, slc, pdt, cp, pdl, /* Set SELECT */
             0, SQLNPTR, SQLNPTR))         /* buffer */
          failure("SET SELECT BUFFER");
       cp += (pdl + 1); /* Locate next area */
       }

       /* Execute the SELECT command */

⑥      if (sqlexe(cur))
          failure("SELECT EXECUTE");

       /* Fetch and display the data */

       length = cp -(SQLTDAP)line; /* Compute the length */
       *cp = 0; /* Append a zero to the string */

       for (;;)
             {
             memset(line, ' ', length); /* Fill the line */
                                        /* with spaces */

⑦            if (rcd = sqlfet(cur)) /* Failure or end of */
                  break;             /*   file?  */

             printf("%s\n", line); /* Print the line */
             }

⑧      if (rcd != 1)               /* Failure on fetch */
             failure("FETCH");
       ...
```

1. Declare a string that contains the SELECT statement.
2. You must compile a SQL statement before you can execute it. Compile the SELECT statement with the *sqlcom* function. The first argument is the cursor handle returned by *sqlcnc*. The second argument specifies the variable that contains the SQL command string. The third argument is zero (0) which means that the command string is null-terminated. The SQL/API will compute the actual length of the argument.
3. Call the *sqlnsi* function to get the number of columns in the SELECT list. For some applications, you may not know the number of columns from which data is

being selected. The *sqlnsi* function returns a pointer to the number of SELECT columns in the second argument (*&nsi*).

4. The *for* loop starts with the first SELECT column and continues until SQLBase processes the number of SELECT columns returned by *sqlnsi*. The *sqldes* function retrieves the attributes of each column. In this example, we are only interested in the data type and length (the third and fourth arguments), so we have specified the remaining arguments as SQLNPTR, which is defined in *sql.h* as a null pointer.
5. The *sqlssb* function sets up the data area in the application that receives the data for each column fetched by *sqlfet* (to be performed later). The second argument is the column number in the SELECT list. The third argument (*pdt*) is assigned the value of SQLPBUF (defined in *sql.h* as a character data type). The fourth argument (*cp*) is a pointer to a buffer in the program. The fifth argument (*pdl*) is the program data length. The fifth argument is zero because it is only relevant for a packed-decimal data type. The remaining arguments are not relevant, so they are assigned SQLNPTR (null pointer). After the *sqlssb* function, *cp* is set to point to the program area that will receive the next column.
6. Execute the SELECT statement using the *sqlexe* function. The *sqlexe* function executes the previously-compiled command.
7. Fetch a row at a time using the *sqlfet* function. Repeat this until all rows in the result set have been fetched. In the program, the length of the print line is set and then a *for* loop gets each row in the result set using the *sqlfet* function and prints it.
8. When the *sqlfet* function fails, the *for* loop terminates and program execution continues at the next statement where the return code for *sqlfet* is checked to ensure that a 1 was returned. The normal end-of-fetch indicator for *sqlfet* is 1, meaning that the last row has been successfully fetched. If a 1 is not returned, there must have been an error.

Result sets

A result set is a collection of rows produced by a query (a SELECT statement).

Result set mode and restriction mode

You can use result set mode (also called scroll mode) and restriction mode with queries. These features are useful for browsing applications.

Result set mode. In *result set mode*, once a result set has been created, you can get to any row in the result set without sequentially fetching forward by calling the *sqlprs* function. Once the cursor is positioned, fetches start from that row.

Restriction mode. In *restriction mode*, the result set of a query is the basis for the next subsequent query, with each query further restricting the result set. This continues until you query a different table. Querying a new table drops the previous result set and establishes a new basis from which to start further restrictions.

While in restriction mode, you can "undo" the current result set and return to the result set as it was before the last SELECT with the *sqlurs* function.

Turn on both result set mode and restriction mode with the *sqlsrs* function. After you call *sqlsrs*, you can turn off restriction mode (but leave result set mode on) with the *sqlspr* function. Calling the *sqlstr* function turns restriction mode back on.

You turn off both result set mode and restriction mode with the *sqlcrs* function. The *sqlcrs* function lets you optionally assign a name to the result set and save it.

Saved result sets

To use a saved result set later, call the *sqlrrs* function and specify the saved result set name. The *sqlrrs* function turns on result set mode and restriction mode.

The *sqldrs* function drops a saved result set.

Be cautious about using saved result sets. Internally, a saved result set is a list of row identifiers (ROWIDs) that is stored in the SYSROWIDLISTS system catalog table. A ROWID changes whenever the row is updated. If one of the rows is updated after you have saved and closed a result set, you get an error if you open the result set later and try to fetch the row.

The example program *ex18.c* illustrates result set mode and restriction mode processing.

Fetching

Row-at-a-time processing

If a query returns multiple rows, fetch each row and process it; you do this by calling the *sqlfet* function after compiling and executing a SELECT command. At this point, SQLBase builds the result set and returns the first row. Each subsequent call to *sqlfet* fetches the next row from the result set.

Fetching the last row of a result set

To fetch the last row of a result set, call the *sqlnrr* function to get the number of rows in the result set, position to the last row with a call to the *sqlprs* function, and then fetch the last row with the *sqlfet* function.

Keeping track of the cursor position

If you need to keep track of the current cursor position, create a counter and increment it by 1 each time you fetch a row. If you position the cursor (with the *sqlprs* function) to a particular row, set the counter to that row position.

Example programs

The example programs *ex08.c* and *ex09.c* show how to fetch rows from a result set.

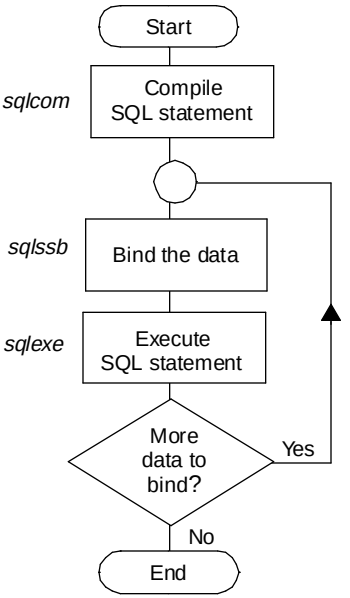
INSERTs, UPDATEs, and DELETEs

The following flowchart shows the sequence of operations necessary to perform an INSERT, UPDATE, or DELETE command using bind variables.

In the flowchart, SQLBase binds the data each time the command executes. This is necessary because in the example program that follows the flowchart, an input line is scanned to find a comma that separates individual values (the values can vary in length). In other words, the input data "changes location," so the bind needs to be done each time the command is executed. If the input data does *not* change location each time, the bind only needs to be done once.

If you are *not* using bind variables, you need only to compile and execute a command using the *sqlcex* function.

Access cycle for INSERT, DELETE, or UPDATE command with bind variables



INSERT with bind variables (ex11.c)

This example shows how to perform an INSERT command using the SQL/API. This program reads a flat file called *data* that contains a row with four column values on each line. Each column value is separated with a comma.

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

SQLTCUR     cur=0;              /* SQLBase cursor number*/
SQLTRCD     rcd=0;              /*SQLBase return code */
int         strscn(char*, char);
void        failure(char*);     /*error handler*/

main ()
{
   FILE*       fp;           /* file pointer*/
   char*       cp;           /* character pointer */
   SQLTDAL     length;       /* length */
   SQLTBNN     bnn;          /* bind number*/
   SQLTNBV     nbv;          /* # of bind variables*/
   char        line[80];     /* input line */
   static char ctbcom[] =    /*CREATE TABLE comand */
     "CREATE TABLE X (A NUMBER, B DATETIME,
        C CHAR(30), D NUMBER)";

① static char  inscom[] =/*INSERT command */
     "INSERT INTO X (A, B, C, D) VALUES (:1, :2, :3, :4)";

  /* CONNECT TO THE DATABASE */
  if (rcd = sqlcnc(&cur, "DEMO", 0))
    failure("CONNECT");

  /*
    CREATE THE TABLE
  */

  if (sqlcex(cur, ctbcom, 0))
    failure("CREATE TABLE");

  /*
    COMPILE THE INSERT COMMAND
  */

② if (sqlcom(cur, inscom, 0))
```

```
    failure("INSERT COMPILE");

  /*
    INSERT THE DATA
  */

  if (!(fp = fopen("DATA", "r"))) /* open input file */
    failure("FILE OPEN");

  while (fgets(line, sizeof(line), fp))/*read the input */
  {
    line[strlen(line) -1] = 0; /* remove new line char */

③   sqlnbv(cur, &nbv);  /* OPTIONAL: could hard code */
                        /* a value of 4*/

    for (cp = line, bnn =1;/*scan the line */
        bnn <= nbv; bnn++)
    {
      length=strscn(cp, ',')/*locate comma*/
④     sqlbnn (cur, bnn, cp,length, 0, SQLPBUF) ;
      cp += length;/*locate end*/
      if (cp* == ',')/*comma ?*/
        cp++;
    }

⑤   if (sqlexe (cur))/*insert row */
      failure ("INSERT EXECUTE");
  }
  /*
    DISCONNECT FROM THE DATABASE
  */
  ...
}       /* end MAIN */
  ...
```

1. Declare the INSERT command.
2. Compile the INSERT command.
3. The *while* loop reads one line of the file at a time. The *sqlnbv* function returns the number of bind variables in the SQL command. The *for* loop finds each column value in the line by scanning for commas.
4. The *sqlbnn* function associates a buffer in the program that contains the data with the appropriate bind variable in the VALUES clause of the INSERT command. Data from the program will be associated with the bind variable in the SQL

command each time the command executes. The arguments for the *sqlbnn* function are the cursor, the sequence number of the bind variable, a pointer to data, the length, the scale (only used for packed-decimal data types), and the program data type.

5. After binding all values in the line, the *sqlexe* function is called to execute the INSERT command.

UPDATE with bind variables (ex19.c)

This example shows how to execute an UPDATE command with a WHERE CURRENT OF clause.

```
#include "stdio.h"
#include "sql.h"
...

① static char updprice[] = /* UPDATE command */
"UPDATE ITEM SET PRICE = :1 WHERE CURRENT OF C1";
SQLTCUR cur1;/* SQLBase first cursor number */
SQLTCUR cur2;/* SQLBase second cursor number */
...

main()
{

/* CONNECT CUR1 TO THE DATABASE */
  if (rcd=sqlcnc(&cur1, dbnam,0))
    cncfail(rcd, "CONNECT");
...
}                    /* end MAIN */

void itemins()
{
  FILE *fp;
  struct item *datap;   /* pointer to input data*/
  int maxitem = 50;   /* highest item number */

  /* Compile insert statement */
  if (sqlcom(cur1, insitem, 0))
    failure(cur1,"COMPILE ERROR");
...
}                  /* end itemins() */

/* The routine fetches each row, including long data,*/
/* updates */
```

```
/* the price by 1 */
void priceupd ()
{
   SQLTDAL   len;     /* Length of data read*/
   SQLTRCD   rcd;     /* Fetch return code*/
   char line [80];    /* output buffer*/
   char newprice[10];/* length of data read*/
   double value;
   char* result;
   char ret_code ='\n';

②  if (sqlscn (cur1,"C1", 2)) /* Name cursor C1 */
      failure(cur1,"SET CURSOR NAME");
   if (sqlcom (cur1, selitem, 0))/*Compile select*/
      failure(cur1,"SELECT COMPILE");
③  if (sqlcom (cur2, updprice, 0)) /* Compile update */
      failure(cur2,"COMPILE ERROR");

   /* Bind price buffer for update statement */

④  if (sqlbnn(cur2,1,(SQLTDAP) &value,sizeof (value),
        0,SQLPDOU))
   failure(cur2,"SQLBNN ERROR ");

  /*
  ** Set buffers for the character columns. Not necessary
  ** for last column,which is a long.
  */
...
   /* Read the long column and display */

   for (; ; )
           /* Update the price according to user input */
...
   for(; ;)
   {
      printf("Enter new price for %s; or return if no
         price change",itembuf);
   /* Get user input */
      result=fgets(newprice, sizeof (newprice), stdin);
      if (*newprice == ret_code)
   {
      printf ("No change in price \n"");
      break;
   }
   else
   {
```

```
        value=atof(newprice) ;
        printf("price=%s\n",newprice) ;
      }
⑤     if (rcd=sqlexe(cur2))
      failure(cur2,"update execute error");
      break;
    }       /* end if */
  }
    if (rcd !=1)     /* If not end of fetch*/
      failure(cur1, "Error on Fetch");
    if (sqlcmt(cur2))/*Commit*/
      failure(cur2, "ON UPDATE COMMIT");
  }
    ...
```

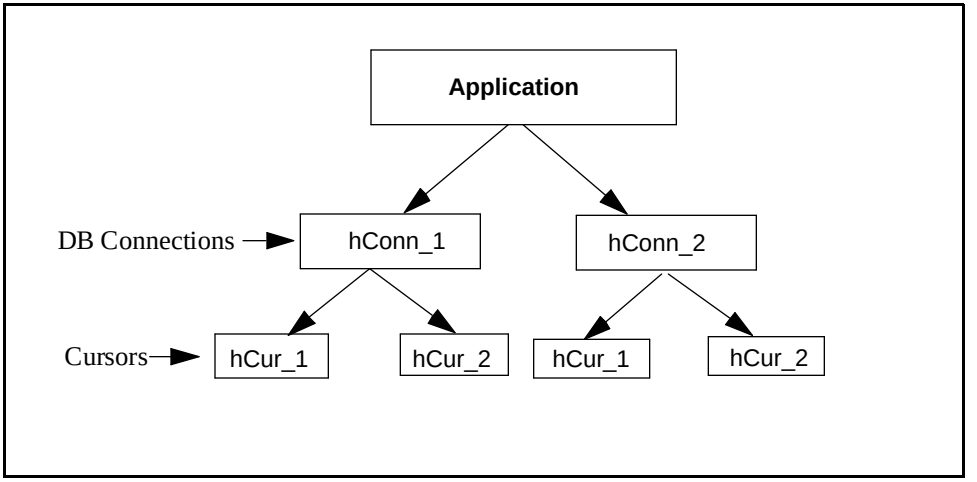
1. Declare the UPDATE command. Note that the CURRENT OF clause specifies "C1". The cursor will be assigned to this name in step 2.
2. The *sqlscn* function assigns a name (second argument) to the cursor specified in the first argument. The third argument is the length of the cursor name.
3. Compile the UPDATE command.
4. Associate the user input to the bind variables in the SET clause of the UPDATE command.
5. The *for* loop accepts the user input for each row that has been fetched. If the user enters a value for the price, the UPDATE command is executed with *sqlexe*.

Connection handles

An *explicit connection handle* defines the scope of a database transaction. Each connection handle represents a separate, independent transaction in the server. An application requests a connection handle by making a *sqlcch* function call, providing the database name, username, and password string. The *sqlcch* function starts a new transaction, returns a connection handle, and authenticates the username and password for the specified database.

For each connection handle, an application can open one or more cursors using the same active connection. An application requests a cursor handle by making a *sqlopc* function call, providing the connection handle as input. The *sqlopc* function call opens a new cursor, associates the cursor with the specified connection, and returns a cursor handle. Since the connection handle is already authenticated and identifies a database, that information no longer is required by the application when opening a new cursor each time. All cursors associated with a connection still belong to the same independent transaction.

Transaction processing operations (such as COMMITs, ROLLBACKs, isolation level changes, and so forth) of one transaction do not affect operations being performed in other transactions. When closing the final cursor in an connection handle, the transaction remains pending. It is either committed or rolled back when the connection handle is terminated using the *sqldch* function call. For details on specifying the closure behavior, read the *sqlset* function description in Chapter 5.



Implicit connection handle

An *implicit connection handle* is created when the *sqlcnc* (CoNnect Cursor) or *sqlcnr* (Connect with No Recovery) functions are issued in the API. An implicit connection encompasses all cursors connected from a given application that use the *sqlcnc* or *sqlcnr* function calls for a specific database. Therefore, an implicit connection represents a single independent transaction per database.

If you are closing the final cursor that is part of an implicit connection handle, a COMMIT, by default, is performed before the cursor is closed. If the cursor was issued using the *sqlcnc* function call, you can specify the ROLLBACK option using the *sqlset* function call with the SQLPCCB parameter. For more details on using *sqlcnc*, *sqlcnr*, and *sqlset*, read the description for these functions in Chapter 5.

Note: Both implicit and explicit connection handles can exist within a single application.

Setting lock time out

Although API calls on different connection handles can be executing on separate threads, a call can be locked out if it is waiting for a thread to complete a task. Similarly, locking can also occur if an application has an implicit connection handle. A cursor may try to enter an API while another cursor is still in it, causing the second cursor to be locked out until the first one exits. By default, the time interval in which SQLBase waits for a lock time out before issuing an error message is 300 seconds for all platforms, except for single-user Windows which is 2 seconds. You can change the setting for the *locktimeout* keyword in the SQL.INI file. For example, to set the time out period to 2 minutes, specify:

```
locktimeout=120
```

Why use connection handles

By creating explicit connection handles within an application, you can establish multiple, independent database connections. This can expand the processing power of your application and increase its performance. Multiple connection handles add these capabilities to an application:

- ability to execute multiple transactions concurrently from the same, single database or different databases.
- ability for you to write applications which are multi-threaded to take advantage of the multi-tasking resource available in win32 platforms. Read *Chapter 6, Creating Multi-threaded Applications* for details.
- ability to create 16-bit MS Windows applications that will later accommodate win32 platforms.

Setting up a connection handle (ex26.c)

This example shows you how to set up connection handles from a single application to the same database. The example is self-explanatory.

```
#include "sql32.h"
#include <stdio.h>
#include <windows.h>
#include <stdlib.h>
#include <ctype.h>
/*-------------------------------------------------------*/

/*             */
/* Example of simple connect using all standard defaults */
/*             */
/*-------------------------------------------------------*/
```

```
main(int argc, char** argv)
{
  SQLTRCD   rcd;/* return code  */
  SQLTCON   con[50];        /* Connection Handle   */
  int       i=1;
  int       j;

/* CONNECTION TO THE DATABASE */
  j = atoi(argv[1]);
  for (i=1;i<=j;i++)
  {
    if (rcd = sqlcch(&con[i], "ISLAND/SYSADM/SYSADM",
0,(SQLTMOD) 0))
  {
     printf("FAILURE ON CONNECTION %d\n",rcd);
     return(1);
  }
  else
     printf("Connection Established \n");
  }
exit(0);
}
```

Transactions

A transaction is a logical unit of work, which is a sequence of SQL statements treated as a single entity.

The scope of a transaction is a single implicit or explicit *connection handle* that an application has connected to the database.

Each connection handle can have multiple cursors which are required to complete the same independent transaction. If there are multiple connection handles set up in the server, a single application can execute multiple transactions to the same or different databases.

An application can request that each SQL statement be committed on completion; otherwise, the database waits for an explicit commit or rollback request from the application. Read *Connection handles* on page *3-17* for more details.

Committing and rolling back

An application gains control when a transaction is committed (made permanent) or rolled back (erased).

A commit (implicit or explicit) destroys all compiled commands for a single connection handle, unless cursor-context preservation is on.

However, when cursor-context preservation is on, SQLBase *does not* preserve cursor context after an isolation level change or a system-initiated ROLLBACK (such as a deadlock, timeout, etc.). SQLBase *does* preserve cursor context after a user-initiated ROLLBACK if *both* of the following are true:

- The application is in Release Locks (RL) isolation level
- No data definition language (DDL) operations were performed

SQLBase either commits or rolls back *all* the data changes made by a transaction. For example, a transaction might add (credit) money to one account and subtract (debit) money from another account. As long as both UPDATES are part of the same transaction, the database is in no danger of being left in an inconsistent state. SQLBase either commits both UPDATEs, or rolls both back.

The *sqlcmt* function causes a commit and the *sqlrbk* function causes a rollback.

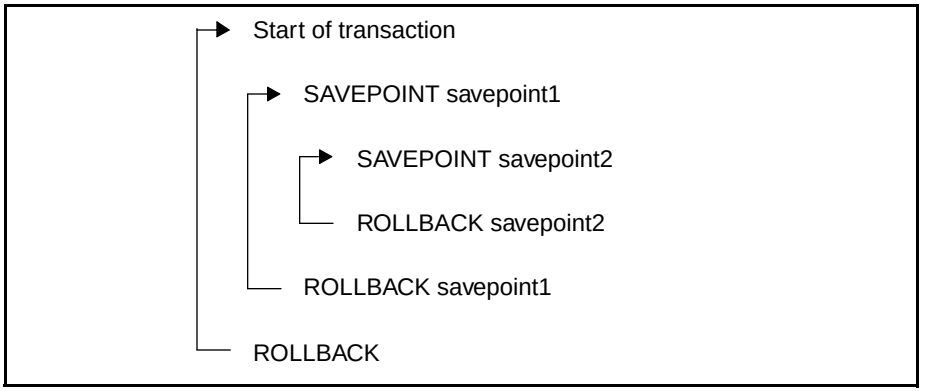
Savepoints

A savepoint is a user-defined and -named point within a transaction. Savepoints let you roll back *portions* of a transaction, rather than forcing you to commit or roll back an *entire* transaction.

The SAVEPOINT command lets you specify a point within a transaction to which you can later roll back if you want to undo part of that transaction. You can specify multiple *savepoints* within a transaction.

The ROLLBACK command has an optional savepoint identifier that lets you name the savepoint to which you want to roll back.

The following graphic illustrates the use of the SAVEPOINT and ROLLBACK commands:



Rolling back to a savepoint does *not* release locks. Rolling back an entire transaction *does* release locks.

You can check the rollback flag (*sqlrbf*) to see whether the previous operation caused a server-initiated rollback.

Distributed transactions

Note: Distributed transactions are not supported with multiple, independent connections to the same database or different databases. Therefore, if you are using connection handles, distributed transactions cannot be enabled. Use the *sqlset* function in conjunction with the SQLPDTR parameter to set distributed transaction mode off. The default for this parameter is off (0). For details on setting this parameter, read the *sqlset* function in Chapter 5.

A distributed transaction coordinates SQL statements among multiple databases that are connected by a network. The databases that participate in a distributed transaction can reside anywhere on the network.

In a distributed transaction, the coordinating application communicates among the participant databases and verifies data integrity. It maintains this integrity even when a crash occurs.

A distributed transaction conforms to the same data consistency rules as a single database transaction — either all of the transaction's statements commit, or none at all.

Server connects *(sqlcsv)* and connects with recovery turned off cannot participate in a distributed transaction. In addition, an application cannot connect to a database in both distributed and non-distributed transaction mode.

In a distributed transaction, one of the participating database servers must also be the *commit server*. The commit server logs information about the distributed transaction and assists in recovery after a network failure. To enable commit server capability for a server, set the *commitserver* keyword to 1 (on) in *sql.ini*.

Databases participating in a distributed transaction must conform to the following communication requirements:

- They must reside on the same network.
- Each participating database server that has commit service enabled must be able to connect to all other servers involved in the distributed transaction. If all the servers have commit service capability, they all must be able to connect with each other.
- If you are using Novell's NetWare, specify the [nwclient] section for each server that is participating in a distributed transaction. This allows servers to communicate mutually. Communication between servers only occurs when a commit server:

- Verifies it can talk to all other participating servers at the time of a distributed commit. (This is performed at most once per participant.)
- Attempts to contact other participating servers under a failure condition.

Use the *sqlset* function in conjunction with the SQLPDTR parameter to set distributed transaction mode on. Once you set this parameter on, all subsequent commands automatically become part of a distributed transaction.

Setting up a transaction (ex06.c)

This example shows you how to set up a transaction that updates multiple tables. The commit (*sqlcmt*) and the rollback (*sqlrbk*) functions ensure that either both tables are updated or that neither is updated.

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

main()
{
SQLTCUR     cur;     /* SQLBASE cursor number */
SQLTRCD     rcd;     /* return code*/

① static char savupdt [] = /* UPDATE savings command */
"UPDATE SAVINGS SET SAV_DOLLARS =
SAV_DOLLARS - 100 WHERE SAV_ACC_NO = 951";

② static char chkupdt [] = /* UPDATE checking command */
"UPDATE CHECKING SET CHK_DOLLARS =
CHK_DOLLARS + 100 WHERE CHK_ACC_NO = 1495";

/*
  CONNECT TO THE DATABASE
*/
...

/*
COMPILE AND EXECUTE UPDATE OF SAVINGS ACCOUNT
*/

③ if (rcd = sqlcex(cur, savupdt, 0))
{
   printf("FAILED UPDATING SAVINGS, rcd = %d\n",rcd);
   sqldis(cur);
   return(1);
}
else
```

```
        printf("ONE HUNDRED DOLLARS SUBTRACTED FROM
        SAVINGS \n");
     /*COMPILE AND EXECUTE UPDATE OF CHECKING ACCOUNT */

④    if (rcd = sqlcex(cur, chkupdt, 0))
     {
        printf("FAILED UPDATING CHECKING (TRANSACTION
             ROLLBACK),rcd = %d\n",rcd);
⑤       sqlrbk(cur);
        sqldis(cur);
        return(1);
     }
     else
        printf("ONE HUNDRED DOLLARS ADDED TO CHECKING \n");

     /* COMMIT TRANSACTION */

⑥    if (rcd = sqlcmt(cur))
        printf("FAILURE ON COMMIT, rcd = %d\n",rcd);
     else
        printf("TRANSFER FROM SAVINGS TO CHECKING
             COMPLETED\n");

     /* DISCONNECT FROM DATABASE */
     ...
  }
```

1. Declare the UPDATE command for the first table.
2. Declare the UPDATE command for the second table.
3. The *sqlcex* function compiles and executes the UPDATE command for the first table in one step. You can use the *sqlcex* function in place of the *sqlcom* and *sqlexe* functions if the SQL statement does not contain bind variables and if you plan to execute it only once.
4. If the UPDATE command for the first table compiled and executed successfully, the UPDATE command for the second table is compiled and executed.
5. If the second UPDATE command is not successful, call the *sqlrbk* function to undo all data modifications.
6. If the second UPDATE command is successful, call the *sqlcmt* function to make permanent all data modifications and release any and all locks.

Setting up a distributed transaction

This example shows how to set up a distributed transaction using the *sqlset* function in conjunction with the SQLPDTR parameter.

Note: Connection handles are not supported for use with distributed transactions. Therefore, this example reflects the use of cursors to connect to multiple databases.

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

void  main(argc, argv)
int  argc;  /* argument count    */
char*argv[];/* -> argument vector    */
{
①   SQLTDPV dtr=1; /*Distributed transaction turned on*/
  SQLTCUR cur1;  /* cursor 1*/
  SQLTCUR cur2;  /* cursor 2 */
  SQLTRCD rcd;   /* return code */
  int     account_number;
  int     transfer_amount;
  char* Decrement_Account = "Update account set
    balance=balance-:1 where account_num = :2";
  char* Increment_Account = "Update account set
    balance=balance+:1 where account_num = :2";

account_number = 14560;
transfer_amount = 500;
if (rcd=sqlset(0, SQLPDTR, (SQLTDAP)&dtr, 0))
  failure(rcd,"SQLSET");
if (rcd=sqlcnc(&cur1, "DALLAS/SYSADM/SYSADM", 0))
  failure(rcd,"CONNECT TO DALLAS");
if (rcd=sqlcnc(&cur2, "AUSTIN/SYSADM/SYSADM", 0))
failure(rcd,"CONNECT TO AUSTIN");

/*
  First decrement the balance from DALLAS
*/
if (rcd = sqlcom(cur1, Decrement_Account, 0))
{
   sqlrbk(cur1);
   failure(rcd,"COMPILE of Decrement_Account");
}
if (rcd = sqlbnu(cur1,(SQLTBNN)2,
  (SQLTDAP)(&account_number), sizeof(int),0,SQLPSIN, 0))
{
```

```
        sqlrbk(cur1);
        failure(rcd,"BIND of account_number for
          Decrement_Account");
      }
      if (rcd = sqlbnu(cur1,(SQLTBNN)1,
        (SQLTDAP)(&transfer_amount), sizeof(int),0, SQLPSIN,0))
      {
        sqlrbk(cur1);
        failure(rcd, "BIND of transfer_amount for
        Decrement_Account");
      }
        if (rcd = sqlexe(cur1))
      {
        sqlrbk(cur1);
        failure(rcd,"EXECUTE of Decrement_Account");
      }

      /*
        Now increment the balance from AUSTIN
      */
      if (rcd = sqlcom(cur2,Increment_Account, 0))
      {
②       sqlrbk(cur1);
        failure(rcd,"COMPILE of Increment_Account");
      }
      if (rcd = sqlbnu(cur2,(SQLTBNN)2,
        (SQLTDAP)(&account_number), sizeof(int), 0, SQLPSIN,
          0))
      {
        sqlrbk(cur1);
        failure(rcd,"BIND of account_number for
          Increment_Account");
      }
      if (rcd = sqlbnu(cur2,(SQLTBNN)1,(SQLTDAP)
        (&transfer_amount), sizeof(int),0, SQLPSIN, 0))
     {
        sqlrbk(cur1);
        failure(rcd,"BIND of transfer_amount for
          Increment_Account");
      }
      if (rcd = sqlexe(cur2))
      {
        sqlrbk(cur1);
        failure(rcd,"EXECUTE of Increment_Account");
      }

③     if (rcd=sqlcmt(cur1))
```

```
{
   failure(rcd,"COMMIT");
}

}      /* end MAIN */

int failure(rcd,str)
SQLTRCD  rcd;
char     *str;
{
printf("ERROR IN %s: %d\n",str,rcd);
exit(0);
}
```

1. Turn on distributed transaction mode.
2. **Each of the rollback statements (***sqlrbk(cur***)) imply a rollback on cur2.**
3. This distributed transaction requires only a single COMMIT statement, since there is only one transaction. You can use any of the cursors to perform the COMMIT.

Cursors

The term cursor refers to one of four things in the SQL/API:

- When the cursor belongs to an explicit connection handle, it identifies a task or activity within a transaction. This task or activity can be compiled/ executed independently within a single connection thread.

  When an application connects to a database using the *sqlcch* function call, SQLBase returns a connection handle. When the connection handle is included in a function call to open a new cursor, the function call returns a cursor handle. You use the cursor handle in subsequent SQL/API calls to identify the connection thread.

- When a cursor belongs to an implicit connection handle, it identifies a database connection.

  When an application connects to a database using the *sqlcnc* or *sqlcnr* function calls, SQLBase returns a cursor handle. You use the cursor handle in subsequent SQL/API calls to identify the connection.

- A row position in a result set.
- A work space in memory used for processing a SQL command.

Cursor work space information

You can retrieve information about a SQL command associated with a particular cursor using the SQL/API functions listed below.

For most of the functions, pass both a cursor handle and a pointer to a variable where the value is returned. The variables are defined in *sql.h* with typedefs.

Function	Description	Typedef
sqlcty	**Command TYpe** - The SQL command type. *Sql.h* defines a code for each command type.	SQLTCTY
sqlepo	**Error POsition** - The offset (starting with 0) of the error within the SQL command which caused the syntax error.	SQLTEPO
sqlnbv	Number of Bind Variables - The number of bind variables associated with a SQL command.	SQLTNBV
sqlnsi	Number of SELECT Items - The number of items in the query's SELECT list.	SQLTNSI
sqlrbf	Rollback flag - The status of the system rollback flag: 1 after a server-initiated rollback and 0 otherwise.	SQLTRBF
sqlrcd	**Return code** - The return code of the most recent SQL/API function: a 0 if the function was successful and a non-zero value otherwise.	SQLTRCD
sqlrow	**Number of rows** - The number of rows affected by the SQL command.	SQLTROW

Cursors and connection handles

To perform tasks that access a single database, you can first create an explicit connection handle using the *sqlcch* function call in your SQL/API application and then open cursors within the connection handle using the *sqlop*c function call. Within an *sqlopc* call, you can assign each cursor its own SQL command. All cursors that access the single database belong to the explicit connection handle and represent a single transaction.

If you have used the *sqlcnc* or *sqlcnr* function calls, your cursors connect directly to a specified database, under a user name and password. An implicit connection handle is automatically created for you and all cursors that connect to the same database, regardless of the user name and password belong to the implicit connection.

By explicitly creating multiple connection handles on Win32 applications, you can have multiple transactions that may access the same database or different databases within the same application. Each connection handle represents a separate thread and

can concurrently enter an API and execute independently. This is known as a multi-threaded application. For details on creating multi-threaded applications with SQLBase, read Chapter 6, *Creating Multi-threaded Applications*.

Connecting to the same database

Cursors that are part of the same implicit or explicit connection handle allow a transaction to connect to the same database. This is useful, for example, when updating a column in one table based on the value in a column of another table. Having already executed a SELECT command on the first cursor, you can subsequently fetch each row of the result set with that same cursor and UPDATE the fetched rows with a second cursor.

Because all of an application's cursors that are associated with the same connection handle are part of the same transaction, a commit or rollback (implicit or explicit) by *any* one of the transaction's cursors commits or rolls back the work done by *all* of the transaction's cursors.

Connecting to different databases

When implicit or explicit connection handles exist for different databases, the databases can be located on the same or different servers and each database maintains its own transaction and rollback information.

Consider an application with six connection handles, which are connected to six different databases. The application has established six separate transactions.

Because only those cursors that are connected to the same connection handle are part of the same transaction, a commit or rollback (implicit or explicit) request by the application commits or rolls back *only* the work done by that connection handle.

Using multiple cursors and connection handles (ex16.c)

This example connects to two cursors (cur1 and cur2). One cursor (cur1) sets the select buffers, the other cursor (cur2) compiles the SQL UPDATE command.

This program scans an employee table and asks a supervisor which employee to award a bonus. It compiles and executes the SQL SELECT command using the cur1 cursor. Then it sets the select buffers using the cur1 cursor. Using the cur2 cursor, it compiles the SQL UPDATE command. Next it fetches a row with the cur1 cursor and ask the supervisor to enter the desired bonus amount, then updates the BONUS table utilizing the cur2 cursor. It continues fetching until an end of fetch. Next it asks the supervisor to specify the desired bonus amount.

①
```
SQLTCUR  cur1 = 0;/* scan cursor  */
SQLTCUR  cur2 = 0;/* update cursor  */
SQLTRCD  rcd1 = 0;/* return code (cur1) */
SQLTRCD  rcd2 = 0;/* return code (cur2) */
```

```
  void      failure(char*); /* error handler      */

  main()
  {
    int       dollars;    /* amount of the bonus  */
    int       employe;    /* employe to grant bonus */
    char      empnam[21]; /* employe name fetched   */
    char      buf[80];    /* input buffer area  */
    long      lnum;       /* long number      */

② static    char    selcom[] =/* SQL select string  */
    "SELECT EMP_NO,EMP_NAME FROM EMP";
  static  char    updcom[] =/* SQL update string  */
     "UPDATE BONUS SET BONUS_AMOUNT =
      :dollars WHERE BONUS_EMP_NO = :employe";

  /* CONNECT TO BOTH CURSORS (use the demo database and all
  /* defaults) */

③ if (rcd1 = sqlcnc(&cur1, "DEMO", 0))
     failure("FIRST CONNECT");

  if (rcd2 = sqlcnc(&cur2, "DEMO", 0))
    failure("SECOND CONNECT");

  /* COMPILE AND EXECUTE SELECT COMMAND (selcom) */

④ if (rcd1 = sqlcex(cur1,selcom,0))
     failure("COMPILE OF SELECT COMMAND");

    /*  SET FETCH BUFFERS (select EMP_NO into employee */
    /* & EMP_NAME into empnam) */
    /*
⑤ if (rcd1 = sqlssb(cur1,1,SQLPUIN,(char  &employe,
       sizeof(employe),0,SQLNPTR,SQLNPTR))
       failure("SET FIRST SELECT BUFFER");

  if (rcd1 = sqlssb(cur1,2,SQLPSTR,empnam,
       sizeof(empnam),0,SQLNPTR,SQLNPTR))
     failure("SET SECOND SELECT BUFFER");

  /* COMPILE UPDATE COMMAND (updcom) */

⑥ if (rcd2 = sqlcom(cur2,updcom,0))
     failure("COMPILE OF UPDATE");
```

```
  /* BIND UPDATE VARIABLES (bind variables with variables */
  /* of same name) */

⑦ if (rcd2 = sqlbnd(cur2,"dollars",0,(char *)
       &dollars,sizeof(dollars),0,SQLPUIN))
       failure("DOLLARS BIND");

  if (rcd2 = sqlbnd(cur2,"employe",0,(char *)
       &employe,sizeof(employe),0,SQLPUIN))
       failure("EMPLOYE BIND");

  /* FETCH ALL EMPLOYEES AND SPECIFY ANY BONUS AMOUNTS */

⑧ while (!(rcd1 = sqlfet(cur1)))
    for (;;)
    {
      printf("\nEnter Bonus Amount for %s ",empnam);
      fflush(stdout);
      fgets(buf,sizeof(buf),stdin);/* read bonus amount  */
      lnum = atol(buf);     /* convert dollar amount  */
      if (strlen(buf) <= 0 ||/* invalid number? or  */
           lnum < 0 ||        /* negative bonus amt? or */
          lnum > 32000)         /* too big a bonus?      */
           continue;            /* ask user for amt again */
      if (!lnum)        /* no amount?      */
           break;             /* no bonus for employe   */
      dollars = (int)lnum;/* set bonus dollar amt   */
⑨    if (rcd2 = sqlexe(cur2))/* perform update      */
  failure("UPDATE");
      break;
    }

if (rcd1 != 1)
    failure("FETCH");

  /* DISCONNECT BOTH CURSORS */

⑩if (rcd1 = sqldis(cur1))
    failure("DISCONNECT OF SELECT CURSOR");

  cur1 = 0;

  if (rcd2 = sqldis(cur2))
    failure("DISCONNECT OF UPDATE CURSOR");

  return(0);
}
```

```
voidfailure(ep)
char*     ep;      /* -> failure msg string  */
{
  SQLTEPO   epo;  /* error position         */
  char    errmsg[SQLMERR];/* error msg text buffer  */

  printf("Failure on %s \n", ep);

  if (rcd1)/* error on cursor 1?  */
  {
    sqlerr(rcd1, errmsg);
    sqlepo(cur1, &epo);
  }

  if (rcd2)/* error on cursor 2?  */
  {
    sqlerr(rcd2, errmsg);
    sqlepo(cur2, &epo);
  }

  if (cur1)/* cursor 1 exists?  */
    sqldis(cur1);

  if (cur2)/* cursor 2 exists?  */
    sqldis(cur2);

  printf("%s(error: %u, position: %u) \n",errmsg,rcd1,epo);
  exit(1);
}
```

1. Declare two cursors and two return codes.
2. Declare the SELECT and the UPDATE commands.
3. Perform two *sqlcnc* functions. Both connections are to the same database, but each connection is associated with a different cursor.
4. Compile and execute the SELECT command with the *sqlcex* function. The SELECT command is associated with the first cursor.
5. Perform the *sqlssb* function to set up the areas in the program that will receive the fetched rows.
6. Compile the UPDATE command with the *sqlcom* function. The UPDATE command is associated with the second cursor.
7. Bind the data for the UPDATE command with the *sqlbnd* function. The first *sqlbnd* function binds the bonus dollars entered by the user. The second *sqlbnd* function binds the employee number from the fetched row.

8. The *while* loop displays each fetched row.
9. The *for* loop prompts the user to enter a bonus amount for each fetched row. If the user enters an amount, the UPDATE command is executed with the *sqlexe* function. If the user does not enter an amount and just presses the return key, the next row is fetched.
10. After displaying and processing the fetched rows, disconnect both cursors.

LONG VARCHAR handling

The LONG VARCHAR data type can hold values longer than 254 bytes. Since the length of the data can be unlimited, you must set up a program loop to read or write LONG VARCHAR data in specified portions.

Reading LONG VARCHAR data. Use *sqlrlo* to read a LONG VARCHAR after fetching a row with *sqlfet*. The *sqlrlo* function identifies the receive buffer for a LONG VARCHAR, so you do not need to call *sqlssb*.

Writing LONG VARCHAR data. Use *sqlwlo* to write a LONG VARCHAR after a compile (*sqlcom)* and bind (*sqlbld* or *sqlbln*), but *before* an execute (*sqlexe)*.

The *sqlbld* function associates a bind variable with an alphanumeric name in a SQL command to a program variable. The *sqlbln* function associates a bind variable with a numeric name in a SQL command to a program variable.

Getting LONG VARCHAR length. Use *sqlgls* to return the number of bytes in a LONG VARCHAR column after fetching a row with *sqlfet*.

Positioning in LONG VARCHAR data. Use *sqllsk* to set a position within a LONG VARCHAR from which to start reading.

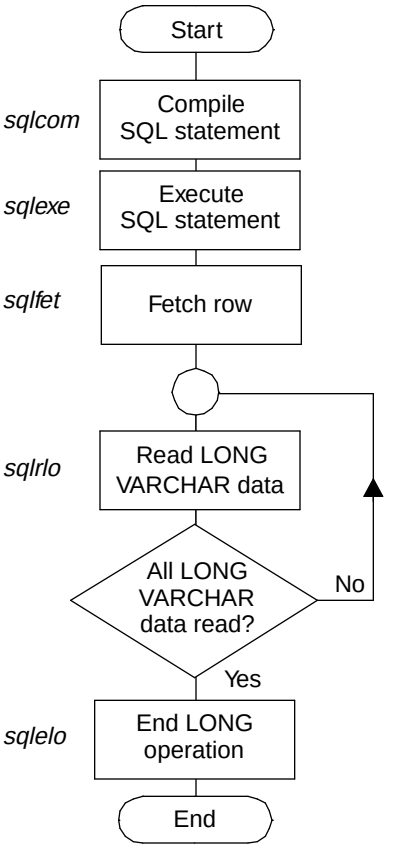
Ending a LONG VARCHAR operation. You must process LONG VARCHAR columns one at a time and the entire long operation must be complete before you can process another LONG VARCHAR. After reading or writing a LONG VARCHAR, call *sqlelo* to end the long operation.

The example programs *ex14.c* and *ex13.c* show how to read and write LONG VARCHAR columns.

Reading LONG VARCHAR columns (ex14.c)

The following flowchart shows the sequence of operations to read LONG VARCHAR columns.

Access cycle to read a LONG VARCHAR with a SELECT statement



This example reads data from a LONG VARCHAR column. Call the *sqlrlo* function to read a LONG VARCHAR after executing a SELECT statement and fetching the row.

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

SQLTCUR  cur;            /* SQLBase cursor number */
SQLTRCD  rcd;            /* Error number */
```

```
char    errmsg[SQLMERR]; /* Error msg text buffer */
void    failure();       /* Error handler*/

main()
{
  int       count;      /* Saying number */
  SQLTDAL   length;     /* Length of data read */
  char*     cp;         /* Character pointer */
  char      buf[50];    /* Buffer to read long */

①  staticchar  select [] = /* SELECT statement */
    "SELECT SAY_NO, SAY_TEXT FROM SAYINGS";

   /* CONNECT TO THE DATABASE */

   if(rcd=sqlcnc(&cur,"ISLAND",0))
   {
      sqlerr(rcd,errmsg);   /* get error message text */
      printf("%s \n",errmsg);
      return(1);
   }

②  /* COMPILE SELECT STATEMENT */

   if (sqlcom(cur,select,0))
      failure("COMPILE OF SELECT");

   /* SET SELECT BUFFER FOR SAYINGS NUMBER */

③  if (sqlssb(cur, 1, SQLPUIN,(charR*)&count,
      sizeof(count), 0, SQLNPTR, SQLNPTR))
   failure("SET SELECT BUFFER");

   /* EXECUTE SELECT STATEMENT */

④  if (sqlexe(cur))
     failure("EXECUTE OF SELECT");

   /* FETCH DATA */

⑤  while (!(rcd = sqlfet(cur)))
   {
      printf("\nSAYING NUMBER %d \n",count);
      for (;;)      /* Read long data */
      {
         memset(buf,' ',sizeof(buf)); /* Clear input */
                                      /* buffer */
```

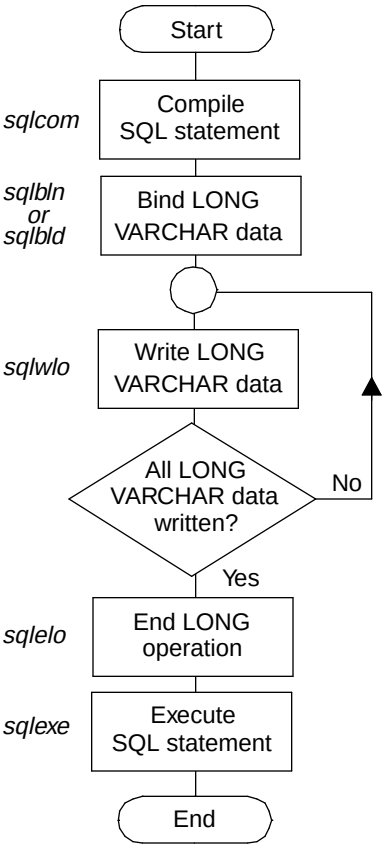
```
⑥          if (sqlrlo(cur, 2, buf, sizeof(buf) - 1,
              &length))
                 failure("READING LONG DATA");

⑦          if (!length)       /* End of long data? */
           {
⑧             if (sqlelo(cur) /* End long operation */
                 failure("ENDING LONG OPERATION");
              break;
           }
           buf[sizeof(buf) - 1] = '\0'; /* Add string * /
                                        /* terminator */
           while (cp = strchr(buf,'\n'))/* Remove */
                                        /* newline char */
                *cp = ' ';
           while (cp = strchr(buf,'\t')) /* Remove tab */
                                         /* characters */
                *cp = ' ';
           printf("%s\n",buf);        /* Print long data */
        }
     }

     if (rcd != 1)
        failure("FETCH");

     /* DISCONNECT FROM THE DATABASE */
     if (sqldis(cur))
        failure("DISCONNECT");
     }                      /* end MAIN */

     void    failure (ep)
     char*   ep;         /* ->failure msg char string*/
     {
        SQLTEPO   epo;   /*Error position*/
        printf("Failure on %s \n", ep);

        sqlrcd(cur, &rcd);  /*Get the Error*/
        sqlepo(cur,&epo);   /*Get Error position*/
        sqlerr(rcd,errmsg); /*Get error message text*/

        sqldis(cur);

        printf("&s (error:%u, position: %u)
           \n",errmsg,rcd,epo);
        exit(1);
     }
```

1. Declare the SELECT statement.
2. Compile the SELECT statement.
3. Set the areas in the program that will receive the fetched (non-long) data with the *sqlssb* function. Note that the LONG VARCHAR column does not need to be set up with the *sqlssb* function.
4. Execute the SELECT statement.
5. Call the *sqlfet* function.
6. Perform the *sqlrlo* function to read the LONG VARCHAR column. The arguments are cursor, column number, buffer, and bytes to read. The *sqlrlo* function performs the equivalent function of *sqlssb*.
7. Continue to read until the length returned by *sqlrlo* is zero.
8. End the long operation with the *sqlelo* function.

Writing LONG VARCHAR columns (ex13.c)

The following flowchart shows the sequence of operations to write LONG VARCHAR columns.

Access cycle to write a LONG VARCHAR with an INSERT or UPDATE statement



This example reads a flat file called *sayings*.1 that contains text and writes the text to a LONG VARCHAR column.

Since the length of the LONG VARCHAR is unlimited (and unknown), you must set up a loop to write the value in fixed portions. You must process LONG VARCHAR data columns one at time and the entire long operation must be complete before you can process the next LONG VARCHAR.

LONG VARCHARs have their own bind functions.

Call *sqlwlo* to write a LONG VARCHAR after compiling an INSERT or UPDATE statement but *before* executing the statement.

```
#include "sql.h"
#include "errsql.h"
#include <stdio.h>
#include <stdlib.h>

SQLTCUR   cur;             /* SQLBase cursor number */
SQLTRCD   rcd;             /* Error number */
char      errmsg[SQLMERR]; /* Error msg text buffer */
void      failure(char*);  /* Error handler */

main()
{
     FILE*   fp;           /* File pointer */
     SQLTROW rows;         /* Number of rows */
     int     count;        /* Saying number to use */
     char    buf[80];      /* Long varchar write buf */

①    static char create [] = /*CREATE TABLE statement*/
       "CREATE TABLE SAYINGS (SAY_NO NUMBER NOT NULL,
          SAY_TEXT LONG VARCHAR)";
     static char insert [] = /*INSERT statement*/
       "INSERT INTO SAYINGS VALUES (:1, :2)";
     /*CONNECT TO THE DATABASE*/
     if (rcd = sqlcnc(&cur,"DEMO",0))
     {
        sqlerr(rcd, errmsg);/* Get Error message text */
        printf("%s \n",errmsg);
        return(1);
     }

     /* CREATE SAYINGS TABLE */
     if (rcd = sqlcex(cur, create,0))
     {
        if (rcd != EXEETVS)/* Not error if tbl exists */
           failure("CREATE SAYINGS TABLE");
     }
     else
        printf("SAYINGS TABLE CREATED\n");

     /* COMPUTE SAYINGS NUMBER */

     if (sqlgnr(cur, "SAYINGS", 0, &rows))
        failure("GET NUMBER OF ROWS");
```

```
     count = (int)rows + 1; /* Compute sayings number */

     /* COMPILE INSERT STATEMENT */

②    if (sqlcom(cur, insert, 0))
        failure("COMPILE OF INSERT");

     /* BIND BY NUMBER*/

③    if (sqlbnn(cur, 1, (SQLTDAP) &count, sizeof(count), 0,
        SQLPUIN))
        failure("BINDING COUNT");

④    if (sqlbln(cur,2))
        failure("BINDING LONG");

     /* WRITE LONG DATA */
     if (!(fp = fopen("SAYINGS.1", "r")))/* Open saying * /
                                          /* text file */
        failure("FILE OPEN");
     while (fgets(buf,sizeof(buf),fp))/* Read the saying */
                                      /*text */
⑤       if (sqlwlo(cur,buf,0))
           failure("WRITE LONG");

     if (fclose(fp))
        failure("FILE CLOSE");

     /* END LONG OPERATION */

⑥    if (sqlelo(cur))
        failure("ENDING LONG OPERATION");

     /* EXECUTE INSERT STATEMENT */

⑦    if (sqlexe(cur))
        failure("EXECUTE");
     else
        printf("SAYING NUMBER %d SUCCESSFULLY
           INSERTED\n",count);

     /* DISCONNECT FROM THE DATABASE */

     if (sqldis(cur))
        failure("DISCONNECT"));
  }             /* end MAIN */
```

```
void failure(ep)
char* ep;          /*->failure msg string*/
{
   SQLTEPO epo;    /*Error position*/

   printf("Failure on &s \n", ep);
      sqlrcd(cur, &rcd);          /*Get the error*/
   sqlepo(cur, &epo);      /*Get error position*/
   sqlerr(rcd, errmsg);    /* Get error message text*/
   sqldis(cur);
   printf("%s (error, %u, position: &u)
      \n",errmsg,rcd,epo);
   exit(1);
}                  /* end MAIN */
```

1. Declare the SQL commands.
2. Compile the INSERT command with the *sqlcom* function.
3. Bind the non-long data with *sqlbnn*.
4. Use the *sqlbln* function to bind the LONG VARCHAR input area to the INSERT command.
5. Read the input data for the LONG VARCHAR data. The *while* loop reads 80 bytes of input data at a time with *fgets* and then performs the *sqlwlo* function. The loop repeats until *fgets* reads a null.
6. Call the *sqlelo* function when all the data has been written for the column value.
7. Call the *sqlexe* function to execute the INSERT command.

Calling stored commands and procedures

You can execute stored commands and procedures from SQL/API. Using the *sqlsto* function, you can store a SQL query, data manipulation command, or procedure for later execution. SQLBase stores the command or procedure in the SYSCOMMANDS system catalog table of a database.

Note that the *sqldst* function allows you to drop a stored command or procedure.

For details on creating stored procedures, read *Chapter 7, Procedures, Triggers, and Events*, of the *SQL Language Reference Manual.*

Executing a stored procedure from SQL/API (ex23.c)

Assume you have stored the following procedure (which uses a table called CHECKING with columns ACCOUNTNUM number and BALANCE number) to update and return bank account balances:

```
PROCEDURE: WITHDRAW
Parameters
  Number: nAccount
  Number: nAmount
  Receive Number: nNewBalance
Local Variables
  String: sUpdate
  String: sSelect
Actions
  Set sUpdate = 'UPDATE CHECKING \
    set BALANCE = BALANCE - :nAmount \
    where ACCOUNTNUM = :nAccount'
  Call SqlImmediate(sUpdate)
  Set sSelect = 'SELECT BALANCE from CHECKING \
    where ACCOUNTNUM = :nAccount \
    into :nNewBalance'
  Call SqlImmediate(sSelect)
\
1,100,,
/
```

The following SQL/API code shows how the procedure WITHDRAW is executed:

```
#include "sql.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
void           failure();          /* error handler */
SQLTCUR        cur;
/*
This program shows how a stored procedure (WITHDRAW,
which has already been stored before) will get executed
*/
main()
{
  int nAccount=1;          /* Account number */
  int nAmount=100;         /* Amount value */
  int nNewBalance;         /* Value of new balance */
  int    n;                /* number value */
  /*
    CONNECT TO THE DATABASE
  */
```

```
    if (sqlcnc(&cur, "island", 0))
      failure("Connect to island");
    /*
      Retrieve the stored procedure
    */
①  if ( sqlret(cur,(SQLTDAP)"WITHDRAW",0))
   failure("WITHDRAW");

②  // bind variables
   if (sqlbnn(cur, 1, (SQLTDAP)&nAccount,sizeof(nAccount),
   0,SQLPSIN)||sqlbnn(cur, 2,
   (SQLTDAP)&nAmount,sizeof(nAmount), 0,SQLPSIN)
   ||sqlbnn(cur, 3,
   (SQLTDAP)&nAmount,sizeof(nAmount), 0,SQLPSIN)
   ||sqlbnn(cur, 3,
   (SQLTDAP)&nNewBalance,sizeof(nNewBalance), 0,SQLPSIN))
   failure("SQLBNN");

③  // set select buffer for receive parameter(s)
   if ( sqlssb(cur, (SQLTSLC)1, SQLPSIN,
   (SQLTDAP)&nNewBalance,sizeof(int),0,0,0))
   failure("SQLSSB");

④  // execute
   if (sqlexe(cur))
   failure("SQLEXECUTE");

⑤ // fetch result
   n=sqlfet(cur);
   printf("%d\n",n);
   printf("The value of new balance is %d\n",nNewBalance);

      if (sqldis(cur))

        failure("DISCONNECT");
      return(0);
   }

   void    failure(ep)
   char*             ep; /* -> failure msg string  */
{

      printf("Failure on %s \n", ep);
      sqldis(cur);
      exit(1);
}
```

1. Retrieve the stored procedure with the *sqlret* function.
2. Bind values for all input and output parameters in the stored procedure. Note the procedure has two input variables and one (output) receive variable.
3. Set the SELECT buffer for the receive parameter with the *sqlssb* function.
4. Execute the stored procedure with the *sqlexe* function.
5. Fetch the result set with the *sqlfet* function.

Functions used with procedures and commands

The following functions can be used with procedures and stored commands:

SQL/API Function	Description
sqlbnd	Bind input data by name.
sqlbnn	Bind input data by number.
sqlbnv	Get the number of input parameters.
sqlcbv	Clear bind variables.
sqlcex	Compile and execute a non-stored command or non-stored procedure.
sqlcom	Compile a non-stored command or non-stored procedure.
sqlcty	Return the command type.
sqldes	Describe output parameters in terms of internal data types and lengths.
sqldii	Describe an INTO variable.
sqldsc	Describe output parameters in terms of external data types and lengths.
sqldst	Drop a stored command or stored procedure.
sqlepo	Retrieve error position.
sqlexe	Execute a command or procedure that has either been previously-compiled or stored.
sqlfet	Fetch next row from result set.

SQL/API Function	Description
sqlget	Return the statement trace status (enabled/disabled) with the SQLPTRC parameter, and the file name of the trace output file with the SQLPTRF parameter.
sqlnbv	Retrieve number of bind variables.
sqlnii	Get the number of INTO variables.
sqlret	Retrieve a command or procedure.
sqlsto	Store a SQL command or procedure in the SYSCOMMANDS system catalog table of a database

Note: If you simultaneously compile and execute a procedure with the *sqlcex* function, SQLBase does not attempt to optimize the SQL statements contained within the procedure. The reason for this is that it offers no real performance advantage, and it incurs a certain amount of overhead.

Bulk execute mode

The bulk execute feature reduces network traffic for multi-row inserts, deletes, and updates. In bulk execute mode, SQLBase buffers data values so that *many* rows can be sent to the server in one message.

Three SQL/API functions support the bulk execute feature:

- *sqlblk* - turns bulk execute mode on or off.
- *sqlbef* - flushes data in the bulk execute buffer.
- *sqlber* - returns error codes for bulk execute operations.

The number of operations per message depends upon the size of the output message buffer which you can set with the *sqloms* function.

You can use the bulk execute feature with chained commands if the chained commands do not contain SELECT statements.

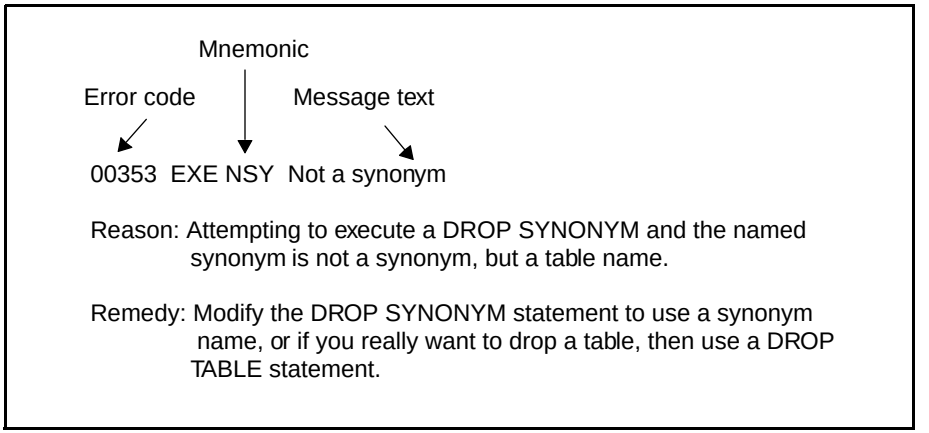
You cannot turn on bulk execute while the autocommit feature is on.

Error handling

All SQLBase error messages are stored in a common error message file called *error.sql*. This file must be present on *all* client and server computers that run SQLBase software.

As the diagram below shows, for each error message there is an:

- Error message text
- Error reason
- Error remedy



The first line of any error contains an error code, a mnemonic, and a message text. When an application detects an error condition, it uses the error code to look up the message text.

Finding error.sql

SQLBase uses this search order to find *error.sql*:

1. Current directory
2. CENTURA directory on the current drive
3. Root directory on the current drive
4. Directories specified by the PATH or DPATH environment variable

If the SQLBASE environment variable is set, SQLBase looks *only* in the directory to which it points. It does not follow the search order outlined above.

Checking the return code

Each SQL/API function returns a code that indicates the success or failure of the function. You should *always* check the return code and continue processing accordingly. For example:

```
if (rcd = sqlcnc(&cur,dbname,0))
  {
  printf("FAILURE ON CONNECT %d\n",rcd);
  exit(1);
  }
else
  printf("Connection Established \n");
```

As another example, if the most-recently executed SQL statement was not successful, you may want to rollback the transaction, disconnect, and exit:

```
if (rcd = sqlcex(cur, chkupdt, 0))
  {
  printf("FAILED UPDATING CHECKING (TRANSACTION ROLLBACK),
         rcd = %d\n",rcd);
  sqlrbk(cur);
  sqldis(cur);
  exit(1);
  }
```

Retrieving the return code

If, unlike the examples above, you did not check the return code when calling a particular function, you can use the *sqlrcd* function to retrieve the return code for the most-recent SQL/API function.

Retrieving the message text

The *error.sql* file contains message text for every return code. Use the *sqlerr* function to retrieve the error message text (without the mnemonic) associated with a return code. Otherwise, use the *sqlfer* function to retrieve the error message text *and* the mnemonic associated with a return code.

In the second example, the application receives the return code into the variable *rcd*. The application could have used the *sqlerr* function to retrieve the error message text, and displayed it or written it to a file before disconnecting.

Retrieving the syntax error position

The *sqlepo* function returns the error position within the most-recently executed SQL statement when SQLBase detects a syntax error.

Retrieving the rollback flag

The *sqlrbf* function returns the rollback flag which is set to 1 after a server-initiated rollback caused by a deadlock or system failure.

Retrieving the reason and remedy

You can use the *sqletx* function to retrieve one or more of the following for a given error code:

- Error message text
- Error reason
- Error remedy

The example program *ex07.c* shows how to handle errors returned from SQL/API functions.

Translating errors

You can create a file that maps SQLBase return codes to other RDBMS vendors' return codes or to return codes that you define yourself. The file should contain lines in this format:

```
x,y
```

where *x* is a SQLBase return code found in *error.sql* and *y* is the corresponding return code that you want SQLBase to return. (There should be no white space after the comma.)

Suppose, for example, that you want SQLBase to return DB2 error codes instead of SQLBase error codes. You need to map SQLBase return codes to their equivalent DB2 return codes. Consider the following: SQLBase returns a value of 1 to indicate an end of fetch condition, while DB2 returns a value of 100. If you want your application to return the value 100 instead of 1 when an end of fetch condition occurs, specify this entry in the translation file:

```
1,100
```

When the end of fetch condition causes an error, your application must call the *sqltec* function to translate the return code from 1 to 100.

As another example, if a CREATE TABLE command specifies the same column name more than once, SQLBase returns 924, but DB2 returns -612. If you want your application to convert 924 to -612, then create this entry in the translation file:

```
924,-612
```

Your application must call the *sqltec* function when an error occurs in order for the return code to be converted from 924 to -612.

If you call the *sqltec* function and the SQLBase return code does not exist, you get a non-zero return code meaning that the translation did not occur. If you always want *some* translation to occur, specify an asterisk ("*") as the *x* value to indicate a global translation. You could specify a generic catch-all return code like 999 to indicate that a system error was reported for an error code not found in the translation table.

For example, SQLBase return code 101 means that an invalid function call was made. If DB2 has no corresponding return code, you can cause a generic value of 999 to be returned when error 101 occurs by specifying:

```
*,999
```

When the application calls *sqltec*, it does not find SQLBase error 101, so it returns 999.

The errorfile configuration keyword

Specify the name of the translation file with the *errorfile* keyword in a client's *sql.ini* file. Configure the keyword as shown below:

```
errorfile=filename
```

where *filename* is the name of the translation file.

Read the *Configuration* chapter in the *Database Administrator's Guide* for more information about this keyword.

Error handling (ex20.c)

The void function from *ex20.c* is called if an error occurs when you execute a SQL/API function.

```
   ...
   void failure ();
   main ()
   {
   ...
   }
①  void failure(p)
②  char* p;              /* Pointer to a string */
   {
③     SQLTEPO epo;      /* Error position */
④     if (cur)          /* Is cursor connected? */
         {
⑤           sqlrcd (cur, &rcd);  /* Get the error */
⑥           sqlepo (cur, &epo);   /* Get error position */
⑦           sqldis (cur);
         }
⑧     printf ("Failure on %s rcd=%d, epo=%d\n", p, rcd,
         epo);
```

```
⑨       exit (1);
    }
```

1. Declare the function.
2. The function has one argument which is a pointer to a character string. You set this argument to a specific value when you call the function.
3. The variable *epo* receives the error position in a SQL command in step 6.
4. Check to see that the cursor is still connected.
5. Use the *sqlrcd* function to retrieve the return code for the most-recent SQL/API function.
6. Use the *sqlepo* function to retrieve the error position within a SQL command.
7. Disconnect from the database.
8. Print an error message that shows the string that was passed to the error-handling function, the return code, and the error position.
9. Call the *exit* function to terminate the program.

Errors

This section describes the following information:

- The common message files called *error.sql* and *message.sql* that are shared by SQLBase client and server programs.
- The SQLBase error window.

About error.sql

All SQLBase error messages are stored in a common error message file called *error.sql*. This file must be present on *all* client and server computers that run SQLBase software.

As the diagram below shows, each error message has message text, a reason, and a remedy.

```
00353 EXE NSY Object <name> specified in DROP SYNONYM
   is not a synonym

      Reason: Attempting to execute a DROP SYNONYM
         and the named synonym is not a synonym but a
         table or view name.

      Remedy: Modify the DROP SYNONYM statement to
         use a synonym name or if you really want to
         drop a table then use a DROP TABLE
         statement.
```

The error message text line contains an error code (in this case, 00353), a mnemonic (EXE NSY), and a message text (Not a synonym). When a program detects an error condition, it uses the error code to look up the error message.

About message.sql

The *message.sql* file contains prompts, confirmations, and non-error messages. This file must be present on *all* client and server computers that run SQLBase software.

SQLBase uses this search order to find *error.sql* and ***message.sql*** on a client or server:

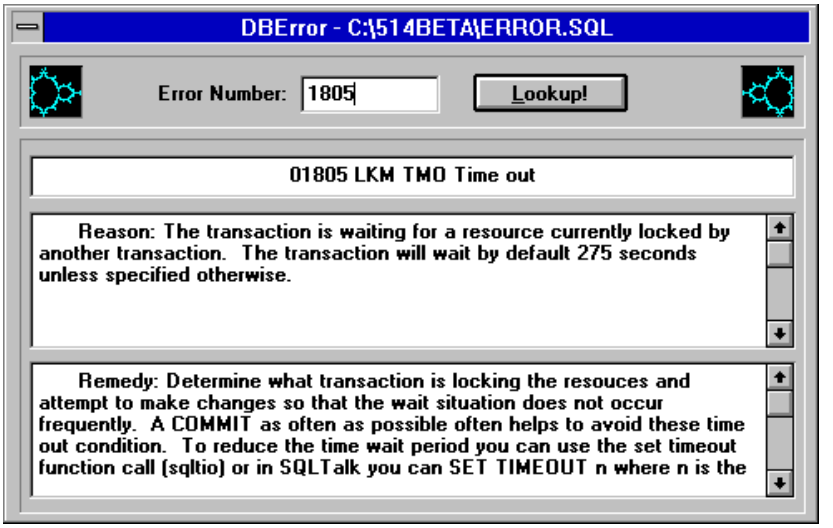
1. Current directory.
2. \SQLBASE directory on the current drive.
3. The root directory.
4. Directories specified by the PATH environment variable.

Displaying errors

SQLBase provides a window that displays the message text, reason, and remedy for a given error code. The program looks up this information in *error.sql*.

The error window program is installed on the client machine when you install SQLBase client software, and is assigned an icon in the client program group or folder.

To access the error window, click on the **Dberror** icon. You access the following window:



To display information about a specific error, enter the error code in the Error Number field, and click **Lookup!**

Tokenized error messages

SQLBase returns one or more error message tokens when an error occurs and substitutes them into an error message's variables if you call *sqltem*(). For example, if you incorrectly specify the directory name from which to restore a database or log files, SQLBase displays error 5132:

```
Missing FROM <directory> clause
```

as:

```
Missing FROM C:\DEMOBKP clause
```

Use this parameter with the *sqlget* function to retrieve the object name (token) returned in an error message.

Use this parameter with the *sqlset* function to set the error token string to customize user errors.

Creating a user-defined error

Assume a table *emp* with referential integrity constraints from which someone attempts to delete a row that contains information about a manager who still has employees assigned to him. SQLBase would return error 383:

```
Cannot delete row until all the dependent rows are deleted
```

You can create an error message specific to this particular violation of referential integrity by using the ALTER TABLE command and editing the *error.sql* file:

1. Edit the *error.sql* file to contain the new error message. With SQLPEMT, you can set the error token string and customize the error to:

   ```
   20001 xxx xxx<manager_name> cannot be removed until all
         subordinates are reassigned
   ```

2. Add the new error message:

   ```
   ALTER TABLE emp ADD USERERROR 20001 FOR 'DELETE_PARENT'
         OF PRIMARY KEY;
   ```

The next time someone attempts to delete a row that contains information about a manager who still has employees assigned to him, SQLBase would return error 20001:

```
<manager name> cannot be removed until all subordinates are
      reassigned
```

Your application is responsible for supplying the error token with which SQLBase replaces the variable (*manager_name*).

The error message token string must be a series of null-terminated strings that ends with a double-null terminator, for example:

```
"first token\0second token\0third token\0\0"
```

Returning an error

Use the *sqltem* (Tokenize Error Message) to return a tokenized error message. This function formats an error message with tokens in order to provide users with more informational error messages.

The *sqltem* function returns one or more of the following from the *error.sql* file for the specified cursor handle:

- Error message
- Error reason
- Error remedy

Each API function call returns a code. You can retrieve the most recent return code with the *sqltem* function, and use it to look up the error message, error reason, and error remedy.

For example, formerly, SQLBase error 175:

```
SQL OLC Cannot open local client workstation file
```

is now:

```
SQL OLC Cannot open local client workstation file <filename>
```

where *filename* is a variable that gets replaced with the name of the file that SQLBase was unable to open.

Tokenizing error messages makes integrity error checking much more informative as well. Instead of reporting only that a data page is corrupt or an index is bad, SQLBase reports the table or index name too.

Non-SQLBase database servers

By default, the *sqltem* function returns the native error code and message from non-SQLBase database servers, but does not return the error reason or remedy.

For example, if you are connected to the Informix server and you receive an error for a table that already exists, the error returned is Informix error code 310:

```
An attempt was made to create a tablespace which already
exists
```

not SQLBase's equivalent 338:

```
Table, view, or synonym <name> already exists
```

If you are accessing a non-SQLBase database server and have set error mapping on, any non-SQLBase error that doesn't have a corresponding SQLBase error is mapped to a generic error message. You can use the *sqltem* function to retrieve the native error code and message that caused the problem.

Note: The other error message handling functions (*sqlerr*, *sqlfer*, and *sqletx*) use a specified return code to retrieve the corresponding error message from the *error.sql* file. An error message returned by any of these functions contains the variable, not the object name; only the *sqltem* function replaces the variable with an actual object name.

Example

```
#include <sql.h>

char    emt [SQLMEMT + 1];/* Error message token */
                          /* buffer */
SQLTCUR cur;              /* Cursor */
SQLTRCD rcd;              /* Return code */

strcpy(emt, "Bob Mitchell");
if (rcd = sqlset(cur, SQLPEMT, emt, 0))/* Set error */
                                       /* message tokens */
{
```

```
        printf("Failure Setting Error Message Tokens (rcd =
        %d)\n", rcd);
}
```

Back up and restore

You can recover from media failures and operator errors which have damaged a database *if* you make backups of a database and its log files regularly.

There are three phases to the process:

- Backup

  Copying a database and its logs to a backup directory. There are two type of backups: online and offline.

- Restore

  Copying a backup of a database and its log files to a database directory.

- Recovery

  Applying one or more log files to a database to bring it up-to-date. This is also called a *rollforward*.

Recovery

There are two kinds of recovery: crash recovery and media recovery. SQLBase performs crash recovery, and the DBA is responsible for media recovery.

Crash recovery

A database can be damaged in a number of ways such as by a power failure or an operator error in bringing down the server. When an event like this happens, SQLBase tries to restore the database to a consistent state by performing crash recovery automatically when a user connects to a crashed database that has just been brought back online. Crash recovery consists of using the transaction logs to redo any committed transactions which had not yet been written to the database and to undo any uncommitted transactions which were still active when the server crashed.

There are situations where SQLBase will not be able to return a database to a consistent state such as when the transaction logs have been damaged during a media failure.

Media recovery

Maintenance is a necessary part of a DBA's job, and involves preparing for events such as a disk head crash, operating system crash, or a user accidentally dropping a database object. You can recover from media failures and user errors which damage a database *if* you back up a database and its log files regularly. Making backups of your

database and log files from which you can restore the database is the only way you can prevent loss of data.

How often you backup the database and its log files is up to you and depends on how much data you can afford to lose. In general, the following are good guidelines:

- Backup the database once a week.
- Backup the transaction log files once a day.

You can minimize loss of data due to a media failure by backing up transaction logs frequently. You should backup all logs since the last database backup so that in the case of a media failure they can be used to recover the database up to the point of that last log backup.

In addition, you should save the database and log files from the last several sets of backups taken. For example, if you make a backup of the database and its logs (snapshot) every Sunday, and make log backups every night, a backup set would consist of the Sunday snapshot, and Monday through Saturday's log file backups. Never rely on just one backup!

Important: Never delete transaction log files. SQLBase automatically deletes log files either when they are backed up or when they are no longer needed for transaction rollback or crash recovery, depending on whether the SQLPLBM parameter is on or off. A database file is useless without its associated log files.

Online backups

An online backup is a copy of a database (*.dbs*) file and its log (*.log*) files that you make using an API function while the server program is running (users are connected to the database and transactions are in progress). The online backup options include:

- *sqlbss*

  Backs up only the database file and those log files needed to restore the database to a consistent state. This includes the current active log file since the *sqlbss* call forces a log rollover. This command is the only backup command which does not require LOGBACKUP to be on. If LOGBACKUP *is* on, the log files left in the database directory should be backed up with a *sqlblf* call. SQLBase will then delete them automatically.

- *sqlbdb*

  Backs up the database file. You should *never* back up a database without also backing up the log files with it.

- *sqlblf*

  Backs up the log files and then deletes them.

The advantage of an online backup is that users can access the database while the backup is being done. This is important to sites which require the database to be up 24 hours a day.

Offline backups

An offline backup is a copy of the database file and log files that you make with an operating system utility or command (such as COPY) after successfully bringing the server down.

The advantage of an offline backup is that you can back up directly to archival media. Online backup commands will not back up files to a tape drive, for example.

Before you can make an offline backup, you must shut down the server gracefully. For details on shutting down the server, read *Chapter 6, Starting and Stopping SQLBase* in the *Database Administrator's Guide*.

You make an offline backup using an operating system command or utility. Below is an example of an offline backup done using the COPY command:

- COPY C:\CENTURA\MYDBS\MYDBS.DBS C:\BACKUPS\MYDBS.BAK
- COPY C:\CENTURA*.LOG C:\BACKUPS

Follow an offline backup with a *sqlset* call specifying the SQLPNLB parameter to tell SQLBase that an offline backup of one or more log files has occurred. SQLBase now knows that these backed up log files are candidates for deletion. If you had backed up the log files with an API function, the files would have been automatically deleted. In the above case, the value of SQLPNLB would be 3.

You restore an offline backup in one of two ways:

- If the backup consists of only a database file, restore it by copying it over the existing damaged database file, making sure the extension is .dbs (you may have changed it, for example, to .bkp when you backed it up), and then connecting to the database. All changes made since the offline backup was done will be lost.
- If the backup consists of a database file and one or more log files, use the *sqlrdb* function to restore the database and then call the *sqlrof* function to apply the logs to bring it up-to-date. The *sqlrdb* copies the backup to the database subdirectory, and the *sqlrof* applies the committed and logged changes made to the database since the offline backup of the database was taken. If SQLBase cannot find the log files to rollforward, you can restore them by either a *sqlrlf* call (which automatically does a *sqlcrf*) or with a copy utility, and then call the *sqlcrf* function explicitly to apply the log files.

In order for the *sqlrdb* call to work, the name of the database backup file must be *database_name*.bkp.

Backing up a database and its log files

The recommended way to backup a database and its log files is with the *sqlbss* function call because it is easy and provides you with a backup from which you can recover the database in one step.

The *sqlbdb* and *sqlblf* function calls are provided for sites with large databases who wish to do incremental backups. Between database backups (both *sqlbss* and *sqlbdb*), you should back up log files using the *sqlblf* function. For example, you could back up the database and logs every Sunday, while on Monday through Saturday, you could back up only the logs.

A backup directory can be on a client or server computer. Once you have backed up a database and its log files to a directory, you can copy the backup files to archival media and delete the backup files from the client or server disk.

Before you can use *sqlbdb* or *sqlblf*, you must set log backup mode on using the SQLPLBM parameter and the *sqlset* function. It is best to set SQLPLBM on just after you create a database and then not change the setting.

Restoring and recovering a database and its log files

Users cannot be connected to the database during a restore and recovery. You should deinstall a multi-user database using the *sqlded* function, perform the restore and rollforward, and then install the database with the *sqlind* function.

If a database becomes damaged, you can restore it from backup with the *sqlrss* function if you created the backup with the *sqlbss* function. After calling *sqlrss*, no further action is necessary because the command will copy not only the backup database file but also the backup log files to the database subdirectory.

If you did not make the backup with *sqlbss* or did a *sqlbss* and want to rollforward as much as possible, you can restore the database with the *sqlrdb* function or a file copy utility.

To rollforward changes made after the database backup and bring the database up-to-date, call the *sqlrof* function:

- Roll forward through all log files available (the default). This recovers as much of the user's work as possible.
- Roll forward to the end of the backup restored. This recovers all committed work up to the point when the database backup was completed. This is essentially a *sqlrss*.

- Roll forward to a specified date and time. This allows you to recover a database up to a specific point in time, and in effect rolls back large "chunks" of committed and logged work that you no longer want applied to the database. For example, if data is erroneously entered into the database, you would want to restore the database to the state it was in before the bad data was entered.

You must have backed up *all* the database's log files and must apply them in order or the rollforward will fail. If you are missing any of the log files, you will not be able to continue rolling forward from the point of the last consecutive log. For example, if you have *1.log*, *2.log*, *4. log*, and *5.log*, but *3.log* is missing, you will only be able to recover the work logged up to *2.log*. *4.log* and *5.log* cannot be applied to the database. An unbroken sequence of log files is required to recover a database backup to its most recent state.

The rollforward operation stops if SQLBase cannot find a log file that it needs. In this situation, you can restore the appropriate log file with a *sqlrlf* function call. The *sqlrlf* function copies the log files needed to recover a restored database from the backup directory to the current log directory and applies them to the restored database. The *sqlrlf* function continues restoring logs until it has exhausted all the logs in the backup directory that can be applied.

If there are more logs to be processed than can fit on disk at one time, you can call the *sqlrlf* function repeatedly to process all the necessary logs.

If a log file requested is not available, you can call *sqlenr* to end recovery using the data restored up to that point.

In summary, the general steps to performing media recovery are:

1. Call *sqlrdb* to restore the database.
2. Call *sqlrof* to declare where rollforward recovery is to terminate.
3. Call *sqlgnl* and *sqlrlf* in a loop to restore and apply any logs needed to perform the wanted rollforward recovery. The *sqlgnl* function returns the name of the next log file needed for recovery and *sqlrlf* restores one or more logs from the specified backup directory.
4. Call *sqlenr* to finish the media recovery process and prepare the database for active use.

Example

The example program *ex17.c* shows how to perform backup and restore operations.

Load and unloading databases

This section describes how to use the SQL/API to load and unload databases.

Loading

There are two ways you can load database information using the SQL/API:

- Using the *sqlldp* function.
- Creating a customized SQL/API function.

It is recommended that you use the standard *sqlldp* function whenever possible. You should only create a custom load function when you need to manipulate the load buffer, such as when you are retrieving database information from a different media.

Using the sqlldp function

Generally, you use the *sqlldp* function (Load Operation) to load database information. The following example shows how this function calls the LOAD command and inputs a file name that exists online:

```
static char loadcmd[] =
"LOAD SQL db.unl ON SERVER";
ret = sqlldp(cur, loadcmd, 0);
```

Creating a customized SQL/API load function

You can also create a customized program to manipulate the load input buffer in the client yourself. For example, you may wish to create a load program that loads information that does not exist online, but perhaps on a tape or an archived file.

The following example creates a SQLTAPI function called *loadx*. This is a customized load function, which processes the load command. You can invoke a program such as this directly from your application program.

This sample operation is similar to writing a LONG VARCHAR type column to the database.

The example *loadx* function processes the load operation and sends it to the backend for compilation and execution. If the load source file resides on the server, the execution is handled completely at the server. If it is on the client, this function handles the retrieval of load data and sends it to the server, in chunks.

This function returns a code after the load operation. If the load operation was successful, this field will contain a zero. In all other cases, this field will contain an error code indicating the error encountered. The *error.sql* file contains a list of error codes and corresponding error messages.

```
#include "sql.h"
#include "errsql.h"
#include "stdio.h"

#define BUFFER_SIZE 1024/* read 1k buffers    */

SQLTAPI  loadx(cur, cmdp, cmdl)
  SQLTCUR   cur;      /* cursor number    */
  SQLTDAP   cmdp;     /* -> command buffer  */
  SQLTDAL   cmdl;     /* command length     */
{
SQLTRCD rcd;        /* return code    */
SQLTDPV on_client;    /* source file ON CLIENT?    */
int len;              /* length    */
FILE *fp;             /* file type    */
char fname[SQLMFNL+1];/* load file name    */
SQLTDAL flen;         /* load file length    */
char buf[BUFFER_SIZE];/* load data buffer    */
int no_more_data;   /* flag for indicating end of data */

if ((rcd = sqlcom(cur, cmdp, cmdl))|| /* compile the */
                                      /* load command */
①   (rcd = sqlget( cur,          /* get ON CLIENT value */
      (SQLTPTY)SQLPCLI,
      (SQLTDAP)&on_client,&len)))
{
 return(rcd);
}
if (on_client)
{
     /* get the load file name    */
②    if (rcd = sqlget(cur, SQLPFNM,fname,&flen))
        return(rcd);
     fname[flen] = 0;

     /* open the local source file for obtaining the
     /* load data.*/
   if ((fp = fopen(fname, "r")) == NULL)
        return(SQLECOF);

     /* Bind the long data by number. */
     if (rcd = sqlbln(cur, 1))
```

```
            return(rcd);

③       no_more_data = 0;
        while(!no_more_data)
            {
                /* read a chunk of the file */
                len = fread(buf, 1, BUFFER_SIZE, fp);
                if (len != BUFFER_SIZE) /*current file */
                                        /* reaches EOF*/
                    no_more_data = 1;

④           if (rcd = sqlwlo(cur, buf, len))/* send the */
                                       /* data to server */
            {
                 sqlelo(cur);     /* end the write operation */
                 return(rcd);
            }                       /* end if */
      }                             /* end while    */

      if ((rcd = sqlelo(cur)) != 0)/* end the long write */
            return(rcd);

      fclose(fp);

      if ((rcd = sqlexe(cur)) != 0)/* execute the load    */
            return(rcd);
    }                               /* end if */
    else
    {
⑤     if (rcd = sqlexe(cur))      /* execute the load    */
            return(rcd);
    }                               /* end else */

    return(rcd = 0);                /* success    */
    }                               /* end function */
```

1. The *sqlget* function returns the value of the ON CLIENT/ON SERVER clause to the LOAD command. The default value is ON CLIENT.
2. Source file is on the client. The code reads the load data and sends it to the backend (SERVER). The load data is sent to the server in a way similar to the inserting of LONG VARCHAR value.
3. This code segment reads chunks of unloaded data from the load file, and sends it to the server, using the *sqlwlo* function call until there is no more data to send.

4. Some data was read from load file. The code sends this data over to the server for processing.
5. The *sqlexe* for the load file on server case executes the load command.

Unloading

There are two ways you can unload database information using the SQL/API:

- Using the *sqlunl* function.
- Creating a customized SQL/API function.

It is recommended that you use the standard *sqlunl* function whenever possible. You should only create a custom load function when you need to manipulate the unload buffer in the client, such as when you need to unload information to an archive.

Using the sqlunl function

Generally, you use the *sqlunl* function (Unload) to unload database information. The following example calls the UNLOAD command and inputs a file name that exists online:

```
static char unlcmd[] =
"UNLOAD COMPRESS DATA SQL db.unl ALL ON SERVER ;";
ret = sqlunl(cur, unlcmd, 0);
```

Creating a customized unload function

The following example creates a SQLTAPI function called *unloadx*. This is a customized unload function, which processes the UNLOAD command. You can invoke a program such as this directly from your application program.

This function processes the unload command and sends it to the backend for compilation and execution. If the unload file destination is on the server, the execution is handled completely at the server. If it is on the client, this function retrieve the unload data from the server and writes it to the destination file.

```
#include "sql.h"
#include "stdio.h"
#include "errsql.h"

#define BUFFER_SIZE 1024/* read 1k buffers    */

SQLTAPI  unloadx (cur, cmdp, cmdl)
    SQLTCUR   cur;                /* cursor number    */
    SQLTDAP   cmdp;               /* -> command buffer */
    SQLTDAL   cmdl;               /* command length */
{
```

```
    SQLTDPVon_client;                /* ON CLIENT flag */
    int    len;                      /* length indicator */
    char   fname[SQLMFNL+1];         /* unload file name*/
    SQLTDALflen;                     /* file name length*/
    SQLTRCDrcd;                      /* return code    */
    FILE   *fp;                     /* unload file pointer*/
    char   buf[BUFFER_SIZE];         /* unload data buffer*/

①  if ((rcd = sqlcom(cur, cmdp, cmdl))|| /* compile unload
*/
                                          /* command */
       /* get ON CLIENT value */
       (rcd = sqlget( cur, (SQLTPTY)SQLPCLI,
          (SQLTDAP)&on_client,&len)))
       {
         return(rcd);   /* if error, report it    */
       }

       if (on_client)
       {
          /* get the unload file name    */
②        if (rcd = sqlget(cur, SQLPFNM,(SQLTDAP)fname,
             (SQLTDAL*)(&flen)))
          return(rcd);   /* if error, report it    */
          fname[flen] = 0;    /* null terminate the  */
                              /* the filename */

          /* Create and open the unload file. */
          if ((fp = fopen(fname, "w")) == NULL)
             return(SQLECOF);/* error: cannot create file
*/

          /* execute the unload command */
          if (rcd = sqlexe(cur))
             return(rcd);

          /* Retrieve the unload data. */
          while(!(rcd = sqlfet(cur)))/* while not end of */
                                     /* fetch */
          {
             while(1)
             {
②               if (rcd = sqlrlo(cur, (SQLTSLC)1, buf,
                   (SQLTDAL)BUFFER_SIZE, &len))
                return(rcd);/* if error report it */

③               if (len)  /* any data retrieved ? */
```

```
                    {
                         fwrite(buf, 1, len, fp);/* write * /
                                 / *data into unload file */
                    }
                    else
                       break;  /* reached the end of data */
               }              /* end while */
                     if (rcd = sqlelo(cur))/* end of */
                             / * long for this fetch */
                          return(rcd);
               }              /* end while    */

               if (rcd > 1)     /* if not end of fetch */
               return(rcd);     /* report error */

               fclose(fp);      /* close the unload file*/
          }                     /* end if */

          else                  /* unload is on the server*/
          {
④             if (rcd = sqlexe(cur))/* just execute the
                                  /* unload command  */
               return(rcd);
          }
          return(rcd = 0);       /* return success    */
          }
```

1. This segment compiles the unload command and gets the information about whether the unload happens on the client or on the server.
2. Destination file is on the client. The code retrieves the unload data in a way similar to the retrieving of a LONG VARCHAR value. The retrieved data is stored in the destination file on client.
3. The unload data is fetched and written to the unload file until end of data is reached.
4. The unload file is on the server, so the unload operation is handled completely on the server.

Microsoft Windows applications

The *sqlini* function initializes the library used for Microsoft Windows and sets up a callback function so that control can pass to Windows while a SQL/API function is executing. You can successfully yield to other tasks or even continue processing within the current task as long as you avoid any interaction with the SQL/API while the application is yielding.

Call the *sqlini* function before the first *sqlcnc*. Call the *sqldon* function before exiting a Microsoft Windows application.

Define LINT_ARGS in your program before other include files.

You must declare *all* pointers used as arguments for SQL/API functions as far pointers. This happens automatically when you include *sql.h*.

The example program *ex21.c* shows how to use SQL/API functions in a Microsoft Windows program.

Chapter 4

SQL/API Functions by Category

This chapter groups the SQL/API functions by functional category, and provides brief descriptions of the functions.

Function categories

This chapter identifies the following SQL/API categories in the SQL/API, and lists the functions in each one.

- Backup and restore functions
- Binding functions
- Bulk execute mode functions
- Compiling and executing functions
- Connecting and disconnecting functions
- Database administration functions
- Environment control functions
- Error handling functions
- Load and Unload functions
- LONG VARCHAR operation functions
- Query functions
- Restriction mode and result set mode functions
- Server file and directory access functions
- Server security functions
- SQLBase internal number functions
- Stored command/procedure functions
- Transaction control functions
- Miscellaneous functions

Backup and restore

Function	Description
sqlbdb	Backup DataBase
sqlblf	Backup Log Files
sqlbss	Backup SnapShot
sqlcrf	Continue RollForward
sqlenr	ENd Rollforward
sqlgnl	Get Next Log

Function	Description
sqlrdb	Restore DataBase
sqlrel	RELease log
sqlrlf	Restore Log Files
sqlrof	ROllForward
sqlrss	Restore SnapShot

Binding

Function	Description
sqlbld	Bind Long Data by name
sqlbln	Bind Long data by Number
sqlbna	Bind data by NAme (with null indicator)
sqlbnd	BiNd Data by name
sqlbnn	BiNd data by Number
sqlbnu	Bind data by NUmber (with null indicator)
sqlcbv	Clear Bind Variables
sqlnbv	Number of Bind Variables

Bulk execute mode

Function	Description
sqlbbr	Bulk execute Return
sqlbef	Bulk Execute Flush
sqlber	Bulk Execute Return
sqlblk	BuLK insert mode

Compiling and executing

Function	Description
sqlcex	Compile and EXecute
sqlcom	COMpile
sqlexe	EXEcute

Connecting and disconnecting

Function	Description
sqlcch	Create Connection Handle
sqlcnc	CoNnect Cursor
sqlcnr	Connect with No Recovery
sqldch	Destroy Connection Handle
sqldis	DISconnect
sqldon	DONe

Database administration

Function	Description
sqlcdr	Cancel Database Request
sqlcre	CREate database
sqldbn	Database Names
sqlded	DEinstall Database
sqldel	DELete database
sqldir	DIRectory of databases
sqlind	INstall Database
sqlldp	LoaD oPeration
sqlsdn	ShutDowN database
sqlsdx	ShutDown database eXtended

Function	Description
sqlunl	UNLOAD Command

Environment control

Function	Description
sqlgbc	Get Backend Cursor
sqlget	GET database parameter
sqlgsi	Get Server Information
sqlims	Input Message Size
sqloms	Output Message Size
sqlrsi	Reset Statistical Information
sqlscp	Set Cache Pages
sqlset	SET database parameter
sqlsta	STAtistics

Error handling

Function	Description
sqlepo	Error POsition
sqlerr	ERRor message
sqletx	Error message TeXt
sqlfer	Full ERror message
sqlrbf	Roll Back Flag
sqlrcd	Return CoDe
sqltec	Translate Error Code
sqltem	Tokenize Error Message
sqlxer	eXternal ERror

Load and Unload operations

Function	Description
sqlldp	Load operation
sqlunl	Unload operation

LONG VARCHAR operations

Function	Description
sqlelo	End Long Operation
sqlgls	Get Long Size
sqllsk	Long SeeK
sqlrlo	Read LOng
sqlwlo	Write LOng

Queries

Function	Description
sqldes	DEScribe items in a SELECT
sqldsc	DeSCribe item of SELECT
sqlfet	FETch next row from result set
sqlfqn	Fully Qualified column Name
sqlgdi	Get Describe Information
sqlgfi	Get Fetch Information
sqlnrr	Number of Rows in Result set
sqlnsi	Number of SELECT Items
sqlssb	Set Select Buffer

Restriction mode and result set mode

Function	Description
sqlcrs	Close Result Set
sqldrs	Drop Result Set
sqlprs	Position in Result Set
sqlrrs	Restart ReStriction mode and Result Set mode
sqlspr	StoP Restriction mode
sqlsrs	Start ReStriction mode and Result Set mode
sqlstr	STart Restriction mode
sqlurs	Undo Result Set

Server file and directory access

Function	Description
sqldox	Directory Open eXtended
sqldrc	DiRectory Close
sqldro	DiRectory Open
sqldrr	DiRectory Read
sqlfgt	File GeT
sqlfpt	File Put
sqlmcl	reMote CLose server file
sqlmdl	reMote DeLete server file or directory
sqlmop	reMote OPen server file
sqlmrd	reMote ReaD server file
sqlmsk	reMote SeeK server file
sqlmwr	reMote WRite server file

Server security

Function	Description
sqlcsv	Connect to SerVer
sqldsv	Disconnect from SerVer
sqlsab	Server ABort process
sqlsds	ShutDown Server
sqlstm	Server TerMinate

SQLBase internal numbers

Function	Description
sqlxad	eXtended ADd
sqlxcn	eXtended Character to Number
sqlxda	eXtended Date Add
sqlxdp	eXtended convert Date to Picture
sqlxdv	eXtended DiVide
sqlxml	eXtended MuLtiply
sqlxnp	eXtended convert Numeric to Picture
sqlxpd	eXtended convert Picture to Date
sqlxsb	eXtended SuBtract

Stored commands and procedures

Function	Description
sqldst	Drop STored SQL command/procedure
sqlret	RETrieve compiled SQL command/ procedure
sqlsto	STOre compiled SQL command/ procedure

Transaction control

Function	Description
sqlcmt	CoMmiT
sqlrbk	RollBacK

Miscellaneous

Function	Description
sqlclf	Change process activity Log File
sqlcpy	CoPY
sqlcty	Command TYpe
*sql*dii	Describe Into Variable
sqlexp	EXecution Plan
sqlgbi	Get Backend Information
sqlgnr	Get Number of Rows
sqlini	INItialize (Microsoft Windows)
sqllab	LABel information
sqlnii	Get number of Into Variables
sqlrow	number of ROWs
sqlscl	Set CLient name
sqlscn	Set Cursor Name
sqlsil	Set Isolation Level
sqltio	TIme Out

Chapter 5

SQL/API Function Reference

This chapter is organized alphabetically by SQL/API function name, and contains the syntax, a detailed description, and an example for each function.

sqlbbr - Bulk execute Return

Syntax

```
#include <sql.h>

SQLTAPI  sqlbbr(cur, rcd, errbuf, buflen, errrow, rbf, errseq)

SQLTCUR       cur;   /* Cursor handle */
SQLTXER  PTR  rcd;   /* Return code */
SQLTDAP  err  buf;   /* Ptr to receiving buffer */
SQLTDAL  PTR  buflen;/* Length of receiving buffer */
SQLTBIR  PTR  errrow;/* Error row number */
SQLTRBF  PTR  rbf;   /* Roll back flag */
SQLTBIR       errseq;/* Error sequence number */
```

Description

This function returns the error return code of the previous bulk execute operation that took place against a non-SQLBase database server.

This function is like *sqlber*, but it also returns the error message text from the non-SQLBase database server.

You can also call *sqlbbr* when processing against SQLBase databases. This means that you can use *sqlbbr* for database-independent applications.

In bulk execute mode, several rows are processed in one call to *sqlexe*. If *sqlexe* returns an error, use *sqlbbr* to find the row that caused the error. Rows that are processed are numbered consecutively. When you call *sqlbbr*, you specify the error sequence number (*errseq*) and *sqlbbr* returns the row number in *errrow*.

For example, if you INSERT 6 rows, they are numbered 1, 2, 3, 4, 5, and 6. If the rows numbered 2, 4, and 6 caused an error, you would call *sqlbbr* and specify 1 in *errseq* and *sqlbbr* would return 2 in *errrow* (meaning row 2 caused an error). Continue to call *sqlbbr*, incrementing the number in *errseq* each time. When *sqlbbr* returns 0 in *rcd*, there are no more errors. This is shown in the table below.

	rcd	errrow	errseq
First *sqlbbr* call	-	2	1
Second *sqlbbr* call	-	4	2
Third *sqlbbr* call	-	6	3

Fourth *sqlbbr* call	0	-	4

Parameters

cur

The cursor handle associated with this function.

rcd

A pointer to the variable where this function returns the status code for the row that caused the error.

Some database servers may return zero value when the operation was not successful, so you should also check to see if *errrow* is greater than zero.

errbuf

A pointer to the buffer where this function copies the error message text.

buflen

A pointer to the variable where this function returns the number of bytes in the retrieved error message text.

errrow

A pointer to the variable where this function returns the row number that was in error.

rbf

A pointer to the variable where this function returns the rollback status indicator:

0	No Rollback
1	Rolled back

errseq

The sequence number of the error to retrieve. Set the errseq parameter to 1 to get the first error, 2 to get the second error, and so on. If the errseq parameter exceeds the number of error messages returned for the last bulk execute, rcd is set to zero to show there are no more error messages.

Return value

If this function returns zero, it was successful. If this function returns a non-zero value, it was unsuccessful.

Related functions

sqlbef *sqlber* *sqlblk*

sqlbdb - Backup DataBase

Syntax

```
include <sql.h>

SQLTAPI   sqlbdb (shandle, dbname, dbnamel, bkpdir, bkpdirl,
                  local, over)

SQLTSVH   shandle;  /* Server handle */
SQLTDAP   dbname;   /* Database name */
SQLTDAL   dbnamel;  /* Length of database name */
SQLTFNP   bkpdir;   /* Backup directory */
SQLTFNL   bkpdirl;  /* Backup directory length */
SQLTBOO   local;    /* True: backup directory on local (client)
                    node */
SQLTBOO   over;     /* True: overwrite existing file */
```

Description

This function copies a database to the specified directory. The database is backed up to a file with the name:

database-name.BKP

If this function finds a control file in the backup directory, the function performs a segmented backup based on the contents of the control file. For details, read the *Database Administrator's Guide*.

Transactions that are committed when the backup starts are included in the backup. Active transactions are not included.

Before you can use *sqlbdb*, you must turn on log backup mode using the SQLPLBM parameter and the *sqlset* function. You only need to do this once for a database (such as just after it has been created), and the setting stays on until you turn it off.

Once a database file is backed up to a directory, you can transfer the backup to archival media; then delete the backup files from the hard disk.

Parameters

shandle

The server handle returned by *sqlcsv*.

dbname

A pointer to the string that contains the database name.

dbnamel

The length of the string pointed to by *dbname*. If the string pointed to by *dbname* is null-terminated, specify zero and the system will compute the length.

bkpdir

A pointer to the string that contains the backup directory name.

bkpdirl

The length of the string pointed to by *bkpdir*. If the string pointed to by *bkpdir* is null-terminated, specify 0 and the system will compute the length.

local

Destination of backup:

0	Backup directory on server.
1	Backup directory on local (client) node.

over

Overwrite indicator:

0	Do not overwrite existing file.
1	Overwrite existing file.

Return value

If this function returns zero, it was successful. If this function returns a non-zero value, it was unsuccessful.

Example

```
SQLTSVH   shandle;
char*     svrname;
char*     password;
SQLTDPV   lbmset;
SQLTFNP   bkpdir;
SQLTFNL   bkpdirl;
SQLTRFM   mode=SQLMEOL;
SQLTLNG   lognum;
SQLTBOO   local,over;
SQLTCUR   cur1;

static  char  dbname1[] = "omed";
strcpy(svrname,”SERVER1”);
password = 0;

bkpdir = "\\BACKUP\\OMED";
bkpdirl = strlen(bkpdir);

printf("value of bkpdir = %s \n",bkpdir);

local=1;
over=1;

/* CONNECT TO OMED */

if (rcd = sqlcnc(&cur1,dbname1,0))
  apierr("SQLCNC");
else
  printf("Connected to OMED \n");

/* SET LOGBACKUP MODE ON */

lbmset=1;
if (rcd = sqlset(cur1,SQLPLBM,(ubyte1p)&lbmset,0))
  apierr("SQLSET");
else
  printf("Logbackupmode is set to %d \n", lbmset);

/* MAKE BACKUP DIRECTORIES */

system("mkdir \\backup");
system("mkdir \\backup\\omed");

/* CONNECT TO SERVER*/
```

```
if (rcd = sqlcsv(&shandle,srvname,password))
  apierr("SQLCSV");
/* BACKUP DATABASE */

if (rcd =
sqlbdb(shandle,dbname1,0,bkpdir,bkpdirl,local,over))
  apierr("SQLBDB");
else
  printf("Backed Up Database \n");

/* RELEASE LOG */

if (rcd = sqlrel(cur1))
  apierr("SQLREL");
else
  printf("Released Logs \n");

/* BACKUP LOGS */

if (rcd =
sqlblf(shandle,dbname1,0,bkpdir,bkpdirl,local,over))
  apierr("SQLBLF");
else
  printf("Backed Up Logs \n");
```

Related functions

sqlblf	*sqlenr*	*sqlrlf*
sqlbss	*sqlgnl*	*sqlrof*
sqlcrf	*sqlrdb*	*sqlrss*
sqlcsv	*sqlrel*	

sqlbef - Bulk Execute Flush

Syntax

```
#include <sql.h>
SQLTAPI   sqlbef (cur)
SQLTCUR   cur;      /* Cursor handle */
```

Description

This function flushes the data (if any) in the bulk execute buffer to the server for processing.

Parameters

cur

The cursor handle associated with this function.

Return value

If this function returns zero, it was successful. If this function returns a non-zero value, it was unsuccessful.

Example

```
/* Set bulk execute mode on */
if (sqlblk(cur, 1))
  goto cleanup;

/* Compile the insert statement */

if (sqlcom(cur, "insert into test values (:1)"))
  goto cleanup;
/* Bind the data and insert the row */

for (i = 0; i < N; i++)
{
  if (sqlbnn(cur, 1, &data[i], 0, 0, SQLPBUF))
    goto cleanup;
  if (sqlexe(cur))
    {
```

```
    /* Error occurred on the execution of the bulk execute,
           retrieve the error messages by calling sqlber() */

    for (j = 1; ; j++)
      {

      /* Retrieve the next error message */

      if (sqlber(cur, &rcd, &rownum, &rbf, j))
         goto cleanup;

      /* Break out of loop, if no more error messages */

      if (!rcd)
         break;

      /* Report the error */

      printf("error on row #%d, rcd = %d0, rownum, rcd);
      }
    }
}

/* Flush out the unprocessed bulk execute buffer to the server
*/

if (sqlbef(cur))
{
    for (j = 1; ; j++)
    {
      if (sqlber(cur, &rcd, &rownum, &rbf, j))
           goto cleanup;
      if (!rcd)
        break;
      printf("error on row #%d, rcd = %d0, rownum, rcd);
    }
}

/* Reset bulk execute mode */

if (sqlblk(cur, 0))
  goto cleanup;
```

Related functions

sqlbbr *sqlber* *sqlblk*

sqlber - Bulk Execute Return

Syntax

```
#include <sql.h>

SQLTAPI  sqlber (cur, rcd, errrow, rbf, errseq)

SQLTCUR        cur;    /* Cursor handle */
SQLTRCD  PTR   rcd;    /* Return code */
SQLTBIR  PTR   errrow; /* Error row number */
SQLTRBF  PTR   rbf;    /* Roll back flag */
SQLTBIR        errseq; /* Error sequence number */
```

Description

This function returns the error return code for the previous bulk execute operation.

In bulk execute mode, several rows are processed in one call to *sqlexe*. If *sqlexe* returns an error, use *sqlber* to find the row that caused the error. Rows that are processed are numbered consecutively. When you call *sqlber*, you specify the error sequence number (*errseq*) and *sqlber* returns the row number in *errrow*.

For example, if you INSERT 6 rows, they are numbered 1, 2, 3, 4, 5, and 6. If the rows numbered 2, 4, and 6 caused an error, you would call *sqlber* and specify 1 in *errseq* and *sqlber* would return 2 in *errrow* (meaning row 2 caused an error). Continue to call *sqlber*, incrementing the number in *errseq* each time. When *sqlber* returns 0 in *rcd*, there are no more errors. This is shown in the table below.

	rcd	errrow	errseq
First *sqlber* call	-	2	1
Second *sqlber* call	-	4	2
Third *sqlber* call	-	6	3
Fourth *sqlber* call	0	-	4

Parameters

cur

The cursor handle associated with this function.

rcd

A pointer to the variable where this function returns the status code for the row that caused the error.

errrow

A pointer to the variable where this function returns the row number that was in error.

rbf

A pointer to the variable where this function returns the rollback status indicator:

0	No Rollback
1	Rolled back

errseq

The sequence number of the error to retrieve. Set the *errseq* parameter to 1 to get the first error, 2 to get the second error, and so on. If the *errseq* parameter exceeds the number of error messages returned for the last bulk execute, *rcd* is set to zero to show there are no more error messages.

Return value

If this function returns zero, it was successful. If this function returns a non-zero value, it was unsuccessful.

Example

The following example shows how to use the *sqlber* function when errors happen during a bulk execute operation.

```
/* Set bulk execute mode on */

if (sqlblk(cur, 1))
  goto cleanup;

/* Compile the insert statement */
```

```
if (sqlcom(cur, "insert into test values (:1)"))
  goto cleanup;

/* Binding the data and insert the row */

for (i = 0; i < N; i++)
{
  if (sqlbnn(cur, 1, &data[i], 0, 0, SQLPBUF))
    goto cleanup;
  if (sqlexe(cur))
    {

    /* Error occurred on the execution of the bulk execute,
       retrieve the error messages by calling sqlber() */

    for (j = 1; ; j++)
      {

      /* Retrieve the next error message */

      if (sqlber(cur, &rcd, &rownum, &rbf, j))
        goto cleanup;
      /* Break out of loop, if no more error messages */

      if (!rcd)
        break;
      /* Report the error */

      printf("error on row #%d, rcd = %d0, rownum, rcd);
      }
    }
}

/* Flush out the unprocessed bulk execute buffer to the
server */

if (sqlbef(cur))
{
  for (j = 1; ; j++)
  {
    if (sqlber(cur, &rcd, &rownum, &rbf, j))
      goto cleanup;
    if (!rcd)
      break;
    printf("error on row #%d, rcd = %d0, rownum, rcd);
  }
}
```

```
/* Reset bulk execute mode */

if (sqlblk(cur, 0))
  goto cleanup;
```

Related functions

sqlbbr *sqlbef* *sqlblk*

sqlbld - Bind Long Data by name

Syntax

```
#include <sql.h>

SQLTAPI  sqlbld (cur, bnp, bnl)

SQLTCUR  cur;     /* Cursor handle */
SQLTBNP  bnp;     /* Name of variable */
SQLTBNL  bnl;     /* Length of variable name */
```

Description

This function associates an alphanumeric bind variable (such as: *comments)* for a LONG VARCHAR column with a variable defined in the program.

The function is called after the *sqlcom* function and before the *sqlwlo* function. Note that *sqlwlo* (not *sqlbld*) specifies the variable that stores the data.

Only one LONG VARCHAR column can be processed at a time. The complete sequence of functions which bind, write, and end the operation must be completed before the next bind for a LONG VARCHAR.

Parameters

cur

The cursor handle associated with this function.

bnp

A pointer to a string that contains the bind variable name.

bnl

The length of the string pointed to by *bnp*. If the string pointed to by *bnp* is null-terminated, specify zero and the system will compute the length.

Return value

If this function returns zero, it was successful. If this function returns a non-zero value, it was unsuccessful.

Example

```
static  char ins[] =   /* sql statement */
  "insert into mytable values (:id, :comm)";
short  ret;                    /* return code */

ret = sqlbld (cur, "comm", 0);
```

Related functions

sqlbln	*sqlcbv*	*sqlrlo*
sqlbna	*sqlelo*	*sqlwlo*
sqlbnu	*sqlnbv*	

sqlblf - Backup Log Files

Syntax

```
#include <sql.h>

SQLTAPI   sqlblf (shandle, dbname, dbnamel, bkpdir, bkpdirl,
                  local, over)

SQLTSVH   shandle;  /* Server handle */
SQLTDAP   dbname;   /* Database name */
SQLTDAL   dbnamel;  /* Length of database name */
SQLTFNP   bkpdir;   /* Backup directory */
SQLTFNL   bkpdirl;  /* Backup directory length */
SQLTBOO   local;    /* True: backup directory on local
                       (client) node */
SQLTBOO   over;     /* True: overwrite existing file */
```

Description

This function copies unpinned log files to the specified directory. When this command completes successfully, SQLBase deletes the log files that were backed up from the current log directory.

Before you can use *sqlblf*, you must set log backup mode to ON using the SQLPLBM parameter and the *sqlset* function. You only need to do this once for a database (such as just after it has been created), and the setting stays on until you turn it off.

Once the log files are backed up to a directory, the backup files can be transferred to archival media and then deleted from the hard disk.

Note: SQLBase supports filenames up to 256 characters including the terminating null character.

Parameters

shandle

The server handle returned by *sqlcsv.*

dbname

A pointer to the string that contains the database name.

dbnamel

The length of the string pointed to by *dbname.* If the string pointed to by *dbname* is null-terminated, specify zero and the system will compute the length.

bkpdir

A pointer to the string that contains the backup directory name.

bkpdirl

The length of the string pointed to by *bkpdir*. If the string pointed to by *bkpdir* is null-terminated, specify zero and the system will compute the length.

local

Destination of backup:

0	Backup directory on server.
1	Backup directory on local (client) node.

over

Overwrite indicator:

0	Do not overwrite existing file.
1	Overwrite existing file.

Return value

If this function returns zero, it was successful. If this function returns a non-zero value, it was unsuccessful.

Example

```
SQLTSVH     shandle;
char*       svrname;
char*       password;
SQLTDPV     lbmset;
SQLTFNP     bkpdir;
SQLTFNL     bkpdirl;
SQLTRFM     mode=SQLMEOL;
SQLTLNG     lognum;
SQLTBOO     local,over;
SQLTCUR     cur1;

static  char  dbname1[] = "omed";
strcpy(svrname,”SERVER1”);
password = 0;

bkpdir = "\\BACKUP\\OMED";
bkpdirl = strlen(bkpdir);

printf("value of bkpdir = %s \n",bkpdir);

local=1;
over=1;

/* CONNECT TO OMED */

if (rcd = sqlcnc(&cur1,dbname1,0))
  apierr("SQLCNC");
else
  printf("Connected to OMED \n");

/* SET LOGBACKUP MODE ON */
```

```
lbmset=1;
if (rcd = sqlset(cur1,SQLPLBM,(ubyte1p)&lbmset,0))
  apierr("SQLSET");
else
  printf("Logbackupmode is set to %d \n", lbmset);

/* MAKE BACKUP DIRECTORIES */

system("mkdir \\backup");
system("mkdir \\backup\\omed");

/* CONNECT TO SERVER */

if (rcd = sqlcsv(&shandle,srvname,password))
  apierr("SQLCSV");
/* BACKUP DATABASE */

if (rcd =
sqlbdb(shandle,dbname1,0,bkpdir,bkpdirl,local,over))
  apierr("SQLBDB");
else
  printf("Backed Up Database \n");

/* RELEASE LOG */

if (rcd = sqlrel(cur1))
  apierr("SQLREL");
else
  printf("Released Logs \n");

/* BACKUP LOGS */

if (rcd =
sqlblf(shandle,dbname1,0,bkpdir,bkpdirl,local,over))
  apierr("SQLBLF");
else
  printf("Backed Up Logs \n");
```

Related functions

sqlbdb	*sqlenr*	*sqlrlf*
sqlbss	*sqlgnl*	*sqlrof*
sqlcrf	*sqlrdb*	*sqlrss*
sqlcsv	*sqlrel*	

sqlblk - BuLK execute

Syntax

```
#include <sql.h>

SQLTAPI   sqlblk (cur, blkflg)

SQLTCUR   cur;     /* Cursor handle */
SQLTFLG   blkflg;  /* 0 = off; 1 = on */
```

Description

This function turns on bulk execute mode.

In bulk execute mode, rows are buffered so that many rows can be sent to the server in one message. This improves the performance of bulk operations on a table, particularly across a network.

The number of rows per message depends upon the size of the output message buffer which can be set with the *sqloms* function.

After performing the operations, use the *sqlbef* function to physically complete the INSERT, UPDATE, or DELETE.

You can use the bulk execute feature with chained commands if they do not contain SELECT commands.

The bulk execute feature *cannot* be turned on at the same time that the autocommit feature is turned on.

Bulk execute mode is a cursor-specific setting.

Parameters

cur

The cursor handle associated with this function.

blkflg

Bulk execute mode setting:

0	Off
1	On

Return value

If this function returns zero, it was successful. If this function returns a non-zero value, it was unsuccessful.

Example

```
/* Set bulk execute mode on */

if (sqlblk(cur, 1))
  goto cleanup;

/* Compile the insert statement */

if (sqlcom(cur, "insert into test values (:1)"))
  goto cleanup;

/* Binding the data and insert the row */

for (i = 0; i < N; i++)
{
  if (sqlbnn(cur, 1, &data[i], 0, 0, SQLPBUF))
    goto cleanup;
  if (sqlexe(cur))
    {

    /* Error occurred on the execution of the bulk execute,
         retrieve the error messages by calling sqlber() */

    for (j = 1; ; j++)
      {

      /* Retrieve the next error message*/

    if (sqlber(cur, &rcd, &rownum, &rbf, j))
           goto cleanup;

      /* Break out of loop, if no more error messages*/

      if (!rcd)
        break;

      /* Report the error */

      printf("error on row #%d, rcd = %d0, rownum, rcd);
      }
```

```
    }
}

/* Flush out the unprocessed bulk execute buffer to the
     server*/

if (sqlbef(cur))
{
  for (j = 1; ; j++)
  {
    if (sqlber(cur, &rcd, &rownum, &rbf, j))
      goto cleanup;
    if (!rcd)
      break;
    printf("error on row #%d, rcd = %d0, rownum, rcd);
  }
}

/* Reset bulk execute mode */

if (sqlblk(cur, 0))
  goto cleanup;
```

Related functions

sqlbbr *sqlber* *sqloms*
sqlbef

sqlbln - Bind Long data by Number

Syntax

```
#include <sql.h>

SQLTAPI   sqlbln (cur, bnn);

SQLTCUR   cur;   /* Cursor handle */
SQLTBNN   bnn;   /* Bind variable number */
```

Description

This function associates a numeric bind variable (such as *:3*) for a LONG VARCHAR column with a variable defined in the program.

The function is called after the *sqlcom* function and before the *sqlwlo* function. Note that *sqlwlo* (not *sqlbln*) specifies the variable that stores the data.

Only one LONG VARCHAR column can be processed at a time. The sequence of binding, writing, and ending the operation must be completed before the next bind for a LONG VARCHAR.

Parameters

cur

The cursor handle associated with this function.

bnn

The number of the bind variable in the SQL command. Bind variable numbers must be unique in SQL commands.

Return value

If this function returns zero, it was successful. If this function returns a non-zero value, it was unsuccessful.

Example

```
static  char ins[] = "insert into mytable values (:1, :2)";
short   ret;    /* return code */

ret = sqlbln(cur, 2);
```

Related functions

sqlbld	*sqlcbv*	*sqlrlo*
sqlbna	*sqlelo*	*sqlwlo*
sqlbnu	*sqlnbv*	

sqlbna - Bind data by NAme (with null indicator)

Syntax

```
#include <sql.h>

SQLTAPI  sqlbna (cur, bnp, bnl, dap, dal, sca, pdt, nli);

SQLTCUR  cur;   /* Cursor handle */
SQLTBNP  bnp;   /* Name of bind variable */
SQLTBNL  bnl;   /* Length of bind variable name */
SQLTDAP  dap;   /* Bind data buffer */
SQLTDAL  dal;   /* Bind data length */
SQLTSCA  sca;   /* Scale of packed decimal data */
SQLTPDT  pdt;   /* Bind program data type */
SQLTNUL  nli    /* Null indicator */
```

Description

This function associates an alphanumeric bind variable (such as *:comments*) for a column with a variable defined in the program.

The *sqlbna* function is identical to *sqlbnd* with one exception: *sqlbna* has an additional argument for the null indicator. This function is used with SQLNetwork routers and gateways to bind null values for non-SQLBase databases. You can use this function with SQLBase databases, but SQLBase ignores the *nli* argument.

Call this function after *sqlcom* and before *sqlexe*.

Parameters

cur

The cursor handle associated with this function.

bnp

A pointer to the string that contains the bind variable name. Bind variable names must be unique in SQL commands.

bnl

The length of the string pointed to by *bnp*. If the string pointed to by *bnp* is null-terminated, specify zero and the system will compute the length.

dap

A pointer to the variable that will be associated to the bind variable.

dal

The length of the value pointed to by *dap*.

If the value pointed to by *dap* is a string *and* null-terminated, specify zero and the system will compute the length.

In all other cases, a calculated length of zero or a specified length of zero causes the column to contain a null value.

sca

The scale (number of decimal places) for a packed-decimal data type. This argument is ignored for other data types. If you are not using a packed-decimal data type, specify zero.

pdt

The program data type of the program variable being bound. Data is converted to the program data type if the SQL data is compatible.

The program data types are shown below. These are defined in *sql.h*.

Program Data Type	Description
SQLPBUF	Character buffer
SQLPDAT	Internal datetime
SQLPDOU	Double
SQLPDTE	Date only
SQLPEBC	EBCDIC buffer
SQLPFLT	Float
SQLPLON	Long text string
SQLPLBI	Long binary buffer
SQLPLVR	Char/long varchar >254
SQLPNBU	Numeric buffer
SQLPNST	Numeric string
SQLPNUM	Internal numeric

Program Data Type	Description
SQLPSCH	Character
SQLPSIN	Integer
SQLPSLO	Long
SQLPSPD	Signed packed decimal
SQLPSSH	Short
SQLPSTR	String (null-terminated)
SQLPTIM	Time only
SQLPUCH	Unsigned character

nli

Null indicator. Before calling *sqlbna*, set this argument to indicate whether or not the value being bound is null:

-1	The data being bound is null. The SQLNetwork router or gateway will generate the native null character for the database server.
0	The data being bound is not null.

Return value

If this function returns zero, it was successful. If this function returns a non-zero value, it was unsuccessful.

Related functions

sqlbld *sqlbnn* *sqlcbv*
sqlbln *sqlbnu* *sqlnbv*
sqlbnd

sqlbnd - BiNd Data by name

Syntax

```
#include <sql.h>

SQLTAPI  sqlbnd (cur, bnp, bnl, dap, dal, sca, pdt);

SQLTCUR  cur;     /* Cursor handle */
SQLTBNP  bnp;     /* Name of bind variable */
SQLTBNL  bnl;     /* Length of bind variable name */
SQLTDAP  dap;     /* Bind data buffer */
SQLTDAL  dal;     /* Bind data length */
SQLTSCA  sca;     /* Scale of packed decimal data */
SQLTPDT  pdt;     /* Bind program data type */
```

Description

This function associates an alphanumeric bind variable (such as *:comments*) for a column with a variable defined in the program.

Call this function after *sqlcom* and before *sqlexe*.

Parameters

cur

The cursor handle associated with this function.

bnp

A pointer to the string that contains the bind variable name. Bind variable names must be unique in SQL commands.

bnl

The length of the string pointed to by *bnp*. If the string pointed to by *bnp* is null-terminated, specify zero and the system will compute the length.

dap

A pointer to the variable that will be associated to the bind variable.

dal

The length of the value pointed to by *dap*.

If the value pointed to by *dap* is a string *and* null-terminated, specify zero and the system will compute the length.

In all other cases, a calculated length of zero or a specified length of zero causes the column to contain a null value.

sca

The scale (number of decimal places) for a packed-decimal data type. This argument is ignored for other data types. If you are not using a packed-decimal data type, specify zero.

pdt

The program data type of the program variable being bound. Data is converted to the program data type if the SQL data is compatible.

The program data types are shown below. These are defined in *sql.h*.

Program Data Type	Description
SQLPBUF	Character buffer
SQLPDAT	Internal datetime
SQLPDOU	Double
SQLPDTE	Date only
SQLPEBC	EBCDIC buffer
SQLPFLT	Float
SQLPLON	Long text string
SQLPLBI	Long binary buffer
SQLPLVR	Char/long varchar >254
SQLPNBU	Numeric buffer
SQLPNST	Numeric string
SQLPNUM	Internal numeric
SQLPSCH	Character
SQLPSIN	Integer
SQLPSLO	Long
SQLPSPD	Signed packed decimal

Program Data Type	Description
SQLPSSH	Short
SQLPSTR	String (null-terminated)
SQLPTIM	Time only
SQLPUCH	Unsigned character

Return value

If this function returns zero, it was successful. If this function returns a non-zero value, it was unsuccessful.

Example

```
static  char  insitem =        /* SQL insert statement */
  "insert into item values (:item, :name)";

int    item;     /* item number */
char   name[25]; /* item name */
short  ret;      /* return code */

/* Bind integer data */

ret = sqlbnd(cur,"item",0,&item,sizeof(item),0,SQLPUIN);

/* Use defaults for pdt and bnl */

ret = sqlbnd(cur,"name",0,name,25,0,SQLPBUF);
```

Related functions

sqlbld *sqlbnn* *sqlcbv*
sqlbln *sqlbnu* *sqlnbv*
sqlbna

sqlbnn - BiNd data by Number

Syntax

```
#include <sql.h>

SQLTAPI  sqlbnn (cur, bnn, dap, dal, sca, pdt);

SQLTCUR   cur;     /* Cursor handle */
SQLTBNN   bnn;     /* Bind variable number */
SQLTDAP   dap;     /* Bind data buffer */
SQLTDAL   dal;     /* Bind data length */
SQLTSCA   sca;     /* Scale of packed decimal data */
SQLTPDT   pdt;     /* Bind program data type */
```

Description

This function associates a numeric bind variable (such as *:3*) for a column with a variable defined in the program.

You must call this function after *sqlcom* and before *sqlexe*.

Parameters

cur

The cursor handle associated with this function.

bnn

The number of the bind variable in the SQL command. Bind variable numbers must be unique in SQL commands.

dap

A pointer to the program variable that will be associated to the bind variable.

dal

The length of the value pointed to by *dap*.

If the value pointed to by *dap* is a string *and* null-terminated, specify zero and the system will compute the length.

In all other cases, a calculated length of zero or a specified length of zero causes the column to contain a null value.

sca

The scale (number of decimal places) for a packed-decimal data type. This argument is ignored for other data types. If you are not using a packed-decimal data type, specify zero.

pdt

The program data type of the program variable being bound. Data is converted to the program data type if the SQL data is compatible.

The program data types are shown below. These are defined in *sql.h*.

Program Data Type	Description
SQLPBUF	Character buffer
SQLPDAT	Internal datetime
SQLPDOU	Double
SQLPDTE	Date only
SQLPEBC	EBCDIC buffer
SQLPFLT	Float
SQLPLON	Long text string
SQLPLBI	Long binary buffer
SQLPLVR	Char/long varchar >254
SQLPNBU	Numeric buffer
SQLPNST	Numeric string
SQLPNUM	Internal numeric
SQLPSCH	Character
SQLPSIN	Integer
SQLPSLO	Long
SQLPSPD	Signed packed decimal
SQLPSSH	Short
SQLPSTR	String (null-terminated)
SQLPTIM	Time only

Program Data Type	Description
SQLPUCH	Unsigned character
SQLPUIN	Unsigned integer
SQLPULO	Unsigned long
SQLPUPD	Unsigned packed decimal
SQLPUSH	Unsigned short

Return value

If this function returns zero, it was successful. If this function returns a non-zero value, it was unsuccessful.

Example

```
char len = 7;      /* length of phone number */
short ret;         /* API return code */
char pnum[8];      /* phone number */

static char updt[] = /* SQL statement */
  "UPDATE CUST SET PHONE = :1 WHERE CURRENT OF CURSOR";

ret = sqlbnn(cur, 1, pnum, sizeof(pnum) ,0, SQLPBUF);
```

Related functions

sqlbld *sqlbnd* *sqlcbv*
sqlbln *sqlbnu* *sqlnbv*
sqlbna

sqlbnu - Bind data by NUmber

Syntax

```
#include <sql.h>

SQLTAPI sqlbnu (cur, bnn, dap, dal, sca, pdt, nli);

SQLTCUR  cur;   /* Cursor handle */
SQLTBNN  bnn;   /* Bind variable number */
SQLTDAP  dap;   /* Bind data buffer */
SQLTDAL  dal;   /* Bind data length */
SQLTSCA  sca;   /* Scale of packed decimal data */
SQLTPDT  pdt;   /* Bind program data type */
SQLTNUL  nli    /* Null indicator */
```

Description

This function associates a numeric bind variable (such as *:3*) for a column with a variable defined in the program.

The *sqlbnu* function is identical to *sqlbnn* with one exception: *sqlbnu* has an additional argument for the null indicator. This function is used with SQLNetwork routers and gateways to bind null values for non-SQLBase databases. You can use this function with SQLBase databases, but SQLBase ignores the *nli* argument.

Call this function after *sqlcom* and before *sqlexe*.

Parameters

cur

The cursor handle associated with this function.

bnn

The number of the bind variable in the SQL command. Bind variable numbers must be unique in SQL commands.

dap

A pointer to the program variable that will be associated to the bind variable.

dal

The length of the value pointed to by *dap*.

If the value pointed to by *dap* is a string *and* null-terminated, specify zero and the system will compute the length.

In all other cases, a calculated length of zero or a specified length of zero causes the column to contain a null value.

sca

The scale (number of decimal places) for a packed-decimal data type. This argument is ignored for other data types. If you are not using a packed-decimal data type, specify zero.

pdt

The program data type of the program variable being bound. Data is converted to the program data type if the SQL data is compatible.

The program data types are shown below. These are defined in *sql.h*.

Program Data Type	Description
SQLPBUF	Character buffer
SQLPDAT	Internal datetime
SQLPDOU	Double
SQLPDTE	Date only
SQLPEBC	EBCDIC buffer
SQLPFLT	Float
SQLPLON	Long text string
SQLPLBI	Long binary buffer
SQLPLVR	Char/long varchar >254
SQLPNBU	Numeric buffer
SQLPNST	Numeric string
SQLPNUM	Internal numeric
SQLPSCH	Character
SQLPSIN	Integer
SQLPSLO	Long
SQLPSPD	Signed packed decimal

Program Data Type	Description
SQLPSSH	Short
SQLPSTR	String (null-terminated)
SQLPTIM	Time only
SQLPUCH	Unsigned character
SQLPUIN	Unsigned integer
SQLPULO	Unsigned long
SQLPUPD	Unsigned packed decimal
SQLPUSH	Unsigned short

nli

Null indicator. Before calling *sqlbnu*, set this argument to indicate whether or not the value being bound is null:

-1	The data being bound is null. The SQLNetwork router or gateway will generate the native null character for the database server.
0	The data being bound is not null.

Return value

If this function returns zero, it was successful. If this function returns a non-zero value, it was unsuccessful.

Related functions

sqlbld *sqlbnd* *sqlcbv*
sqlbln *sqlbnn* *sqlnbv*
sqlbna

sqlbss - Backup SnapShot

Syntax

```
#include <sql.h>

SQLTAPI  sqlbss (shandle, dbname, dbnamel, bkpdir, bkpdirl,
  local, over)

SQLTSVH  shandle;  /* Server handle */
SQLTDAP  dbname;   /* Database name */
SQLTDAL  dbnamel;  /* Database name length */
SQLTFNP  bkpdir;   /* Backup directory */
SQLTFNL  bkpdirl;  /* Backup directory length */
SQLTBOO  local;    /* True: backup directory on local
                      (client) node */
SQLTBOO  over;     /* True: overwrite existing file */
```

Description

This function copies a database and its associated transaction log files to the specified directory.

The *sqlbss* function is the recommended way to backup a database and its log files because there is only one step (*sqlrss*) needed to bring a database and its log files to a consistent state.

Transactions that are committed when the backup is started are included in the backup. Active transactions are not included.

This function call forces a log rollover (*sqlrlf*) automatically.

Once a database and its transaction log files are backed up to a directory, you can transfer the copies to archival media and then delete them from the hard disk.

You cannot call *sqlbss* while in Read-Only (RO) isolation level.

Note: SQLBase supports filenames up to 256 characters including the terminating null character.

Parameters

shandle

The server handle returned by *sqlcsv.*

dbname

A pointer to the string that contains the database name.

dbnamel

The length of the string pointed to by *dbname*. If the string pointed to by *dbname* is null-terminated, specify zero and the system will compute the length.

bkpdir

A pointer to the string that contains the backup directory name.

bkpdirl

The length of the string pointed to by *bkpdir*. If the string pointed to by *bkpdir* is null-terminated, specify zero and the system will compute the length.

local

Destination of backup:

0	Backup directory on server.
1	Backup directory on local (client) node.

over

Overwrite indicator:

0	Do not overwrite existing file.
1	Overwrite existing file.

Return value

If this function returns zero, it was successful. If this function returns a non-zero value, it was unsuccessful.

Example

```
SQLTSVH   shandle;
char*     password;
SQLTDPV   lbmset;
SQLTFNP   bkpdir;
SQLTFNL   bkpdirl;
SQLTRFM   mode=SQLMEOL;
SQLTLNG   lognum;
```

```
SQLTBOO   local,over;

static  char  dbname1[] = "island";  /* default database
                                     /* name */
static  char  srvname[] = "SERVER1"; /* server name */

password = 0;
local=1;
over=1;

/* CONNECT TO SERVER */

if (rcd = sqlcsv(&shandle,srvname,password))
apierr("SQLCSV");

/* MAKE BACKUP DIRECTORIES */

system("mkdir \\backup\\snapshot");

bkpdir = "\\BACKUP\\SNAPSHOT";
bkpdirl = strlen(bkpdir);

/* BACKUP SNAPSHOT */

if (rcd =
sqlbss(shandle,dbname1,0,bkpdir,bkpdirl,local,over))
  apierr("SQLBSS");
else
  printf("Backup Snapshot Database \n");
```

Related functions

sqlbdb	*sqlenr*	*sqlrlf*
sqlblf	*sqlgnl*	*sqlrof*
sqlcrf	*sqlrdb*	*sqlrss*
sqlcsv	*sqlrel*	

sqlcbv - Clear Bind Variables

Syntax

```
#include <sql.h>

SQLTAPI  sqlcbv(cur)

SQLTCUR  cur;     /* Cursor handle */
```

Description

This function clears all information stored for bind variables for a cursor.

When a program variable is bound, information about the variable is saved. This includes pointers to the data and the name of the bind variable (if bound by name). This function clears this information and frees the memory that stores it.

Parameters

cur

The cursor handle associated with this function.

Return value

If this function returns zero, it was successful. If this function returns a non-zero value, it was unsuccessful.

Example

```
if (rcd = sqlcbv(cur))
  apierr("SQLCBV");
else
  printf("Cleared Bind Variables \n");
```

Related functions

sqlbld *sqlbnd* *sqlbnu*
sqlbln *sqlbnn* *sqlnbv*
sqlbna

sqlcch - Create Connection Handle

Syntax

```
#include <sql.h>

SQLTAPI  sqlcch(hConp, dbnamp, dbnaml, flag)

SQLTCNH   PTR  hConp;   /* Connection handle */
SQLTDAP        dbnamp; /* Pointer to identification string */
SQLTDAL        dbnaml; /* Identification string length */
SQLTMOD        flag;   /* future flag                  */
```

Description

This function establishes a new connection to the specified database. It issues a connection handle to identify the database. There can be a maximum of 256 connection handles.

Parameters

hConp

A pointer to a connection handle where this function returns a new connection handle.

dbnamp

A pointer to an identification string that contains the database name, username, and password, separated by forward slashes:

databasename/*username*/*password*

If the database name, username, or password is not specified, then the system uses the current default. For example, you can specify the following connect string in which case the default database name and username are used:

```
//password
```

These rules are used:

- The characters before the first forward slash are the database name.
- Any characters after the first forward slash and before the second forward slash are the username.
- Any characters after a second forward slash are the password.

The default database name, username, and password are determined by:

- The defaultdatabase, defaultuser, and defaultpassword keywords in SQL.INI.
- The default of DEMO/SYSADM/SYSADM

dbnaml

The length of the string pointed to by *dbnamp*. If the string pointed to by the *dbnamp* is null-terminated, you can specify zero for the dbnaml and the system will compute the length.

flag

Future flag. Currently not defined. You can specify zero for this parameter.

Return value

If this function returns zero, it was successful. If this function returns a non-zero value, it was unsuccessful.

Example

```
SQLTCNH   hCon;    /* Connection Handle*/
SQLTCUR   cur;     /* Cursor */
SQLTRCD   rcd;     /* return code */

if (rcd = sqlcch(&hCon, “PAYROLL/BOSS/SECRET”,0,0))
{
  printf(“Failure establihsing a connection (rcd =%d)\n”,rcd);
  exit(0);
}
else printf(“New connection established\n”);

if(rcd = sqlopc(&cur, hCon, 0))
{
  printf(“Failure on cursor open (rcd = %d)\n”, rcd);
  exit(0);
}
else printf(“New cursor opened\n”);

     .
     .
     .
if(rcd = sqldis(cur))
{
  printf(“Failure closing cursor (rcd = %d)\n”, rcd);
  exit(0);
}
```

```
else printf("Cursor closed\n");

if (rcd = sqldch(hCon))
{
  printf("Failure terminating connection (rcd = %d)\n", rcd);
  exit(0);
}
else printf("Connection terminated\n");
```

Related functions

sqldch *sqldis* *sqlopc*

sqlcdr - Cancel Database Request

Syntax

```
#include <sql.h>

SQLTAPI  sqlcdr (shandle, cur)

SQLTSVH   shandle;      /* Server Handle */
SQLTDAP   cur;          /* Cursor Handle */
```

Description

This function cancels a SQL command.

When a database request is in progress and taking too long, *sqlcdr* can be invoked from another process to send a cancel message to the server. If the server is processing a request, it stops processing it and rollbacks the transaction and the process that started the request returns an error code.

If the server receives the cancel message before or after processing a request, the message is ignored.

Parameters

shandle

The server handle returned by *sqlcsv.*

cur

The cursor handle associated with the request that you want to cancel.

Return value

This function returns zero if the cancel message was received by the server while processing a request. If this function returns a non-zero value, it was unsuccessful.

Related functions

sqlsab *sqlsdn* *sqlstm*

sqlcex - Compile and EXecute

Syntax

```
#include <sql.h>

SQLTAPI  sqlcex (cur,dap,dal);

SQLTCUR  cur;   /* Cursor handle */
SQLTDAP  dap;   /* Command buffer */
SQLTDAL  dal;   /* Length of SQL command */
```

Description

This function takes a SQL command or non-stored procedure as input, generates the compiled version of the command/procedure, and executes it. No data is bound.

Use this function to compile and execute a SQL command or procedure that contains no bind variables and only needs to be executed once; examples are data definition and data control commands (CREATE, DROP, GRANT, REVOKE) and data manipulation commands which meet these criteria.

This function also enables server-level commands to create, delete, or alter database areas and storage groups.

All compiled commands for all cursors that the program has connected to the database are destroyed by:

- Commits (explicit or implicit, including implicit by autocommit or by change in isolation level).
- Rollbacks (including rollbacks caused by a deadlock).

Note: You cannot compile and execute a procedure as static before storing it with the *sqlsto* function.

Parameters

cur

The cursor handle associated with this function.

For these SQL commands, use the server handle returned by *sqlcsv* instead:

ALTER DATABASE
ALTER DBAREA
ALTER STOGROUP
CREATE DATABASE
CREATE DBAREA
CREATE STOGROUP
DEINSTALL DATABASE
DROP DATABASE
DROP DBAREA
DROP STOGROUP
INSTALL DATABASE
SET DEFAULT STOGROUP

dap

A pointer to the variable that contains the command or procedure to compile and execute.

dal

The length of the variable pointed to by *dap*. If the value pointed to by *dap* is null-terminated, specify zero and the system will compute the length.

Return value

If this function returns zero, it was successful. If this function returns a non-zero value, it was unsuccessful.

Example

```
char*  name;    /* name */
char   *pswd;   /* password */
short  ret;     /* return code */
char   errmsg [SQLMERR];

static char grant[] =
  "GRANT CONNECT TO %s IDENTIFIED BY %s";

char buf[100]
```

```
sprintf (buf, grant, name, pswd);
if (sqlcex(cur, buf, 0)) /* Compile and execute */
{
  sqlrcd(cur, &ret);     /* Get return code */
  sqlerr(ret, &errmsg);  /* Get error text  */
  printf("%s\n", errmsg);
}
```

Related functions

sqlcom *sqlcsv* *sqlexe*

sqlclf - Change process activity Log File

Syntax

```
#include <sql.h>

SQLTAPI  sqlclf (shandle, logfile, startflag)

SQLTSVH  shandle;    /* Server handle */
SQLTDAP  logfile;    /* Log file name to open */
SQLTFMD  startflag;  /* Start activity log flag */
```

Description

This function opens a new process activity log file for the database server. Use this function to write the messages that appear on the Process Activity server display to a file. This function is useful for multi-user servers.

Instead of using the *sqlclf* function, you could use the *sqlset* function and the SQLPALG parameter.

To turn on logging, specify a file name and set *startflag* to 1. To turn off logging, specify a null filename or set *startflag* to 0.

Note: SQLBase supports filenames up to 256 characters including the terminating null character.

Parameters

shandle

The server handle returned by *sqlcsv.*

logfile

A pointer to the null-terminated string that contains the name of the log file. If null, logging is turned off.

startflag

Indicates whether to start or stop writing to the log file:

0	Stop logging
1	Start logging

Return value

If this function returns zero, it was successful. If this function returns a non-zero value, it was unsuccessful.

Example

```
main()
{
  SQLTSVH   shandle;
  SQLTDAP   srvname;
  char      *password;
  char      *logfile;
  int       startflag;

  srvname = "SERVER1";
  password = 0;
  startflag = 1;
  logfile = "ACTIVITY.LOG";

/* CONNECT TO THE SERVER */

if (rcd = sqlcsv(&shandle,srvname,password))
  apierr("SQLCSV");
else
  printf("Connection Established to Server \n");

/* CHANGE ACTIVITY LOG FILE */
```

```
printf("change activity log file to %s\n", logfile);
if (rcd = sqlclf(shandle,logfile,startflag))
  apierr("SQLCLF");
else
  printf("Successful change and start of server activity log
        file\n");

/* DISCONNECT FROM THE SERVER */

if (rcd = sqldsv(shandle))
  apierr("SQLDSV");
else
  printf("Disconnected from Server \n");
}
```

Related functions

sqlcsv *sqlset*

sqlcmt - CoMmiT

Syntax

```
#include <sql.h>

SQLTAPI  sqlcmt(cur);

SQLTCUR  cur;   /* Cursor handle */
```

Description

This function commits a database transaction and starts a new transaction. All changes to the database since the last commit are made permanent and cannot be undone.

Before a commit, all changes made since the start of the transaction can be rolled back.

A commit releases all locks held by a transaction except when cursor-context preservation is on.

This function commits the work of *all* cursors that an application has connected to the database or connection handle.

Connecting to a database or connection handle causes an implicit commit of a transaction. After establishing this connection to the database, SQLBase issues a COMMIT to establish the starting point of the first transaction in the logging system. However, subsequent connections to other cursors are not specifically database connections, and do not cause SQLBase to issue a COMMIT or activate any transaction control devices. Also, they do not alter the flow of the current transaction and destroy compiled commands.

This function destroys all compiled commands for all cursors and connection handles connected to the database except when cursor-context preservation is on.

The database can also be committed by executing a SQL COMMIT command.

Parameters

cur

The cursor handle associated with this function.

Return value

If this function returns zero, it was successful. If this function returns a non-zero value, it was unsuccessful.

Example

```
ret = sqlcmt(cur);
```

Related functions

sqlrbk

sqlcnc - CoNnect Cursor

Syntax

```
#include <sql.h>

SQLTAPI   sqlcnc (curp, dbnamp, dbnaml)

SQLTCUR   PTR  curp;   /* Cursor handle */
SQLTDAP        dbnamp; /* Connect string */
SQLTDAL        dbnaml; /* Connect string length */
```

Description

This function applies to applications in which you are connecting cursors to a specific database that belong to a single transaction.

This function connects to a database and issues a cursor handle that identifies an implicit connection to a specific database. All cursors that you connect to this database belong to a single transaction and to the same implicit connection handle.

This function can connect to a new database or connect a new cursor to the current database.

Note: To create multiple, independent connections, SQLBase allows you to explicitly create multiple connection handles. You can use connection handles for multiple transactions to the same database within an application, or for creating multi-threaded Win32 applications. For details on creating connection handles, read the section on connection handles in Chapter 3.

Parameters

curp

A pointer to the variable where this function returns the cursor handle.

dbnamp

A pointer to the connect string that contains the database name, username, and password separated by forward slashes:

```
database/username/password
```

These rules are used:

- The characters before the first forward slash are the database name.
- Any characters after the first forward slash and before the second forward slash are the username.
- Any characters after the second forward slash are the password.

If the database name, username, or password is not specified, then the system uses the current default. For example, you can specify the following connect string in which the default database name and username are used:

```
//password
```

The default database name, username, and password are determined by:

- The *defaultdatabase*, *defaultuser*, and *defaultpassword* keywords in *sql.ini*.

- The default of DEMO/SYSADM/SYSADM.

dbnaml

The length of the string pointed to by *dbnamp*. If the string pointed to by *dbnamp* is null-terminated, specify zero and the system will compute the length.

Return value

If this function returns zero, it was successful. If this function returns a non-zero value, it was unsuccessful.

Example

```
SQLTCUR  cur;   /* Cursor handle */
SQLTRCD  rcd;   /* Return code */

if (rcd = sqlcnc(&cur, "PAYROLL/BOSS/SECRET", 0))
  {
    printf("Failure on connect (rcd = %d)\n",rcd);
    exit(0);
  }
else
  printf("Connection established\n");
```

Related functions

sqlcnr *sqldis*

sqlcnr - Connect with No Recovery

Syntax

```
#include <sql.h>

SQLTAPIsqlcnr(curp, dbnamp, dbnaml)

SQLTCUR PTR  curp;   /* Cursor handle */
SQLTDAP      dbnamp; /* Connect string */
SQLTDAL      dbnaml; /* Connect string length */
```

Description

This function applies to applications in which you are connecting cursors to a specific database that belong to a single transaction.

Note: To create multiple, independent connections, SQLBase allows you to explicitly create multiple connection handles. You can use connection handles for multiple transactions to the same database within an application, or for creating multi-threaded Win32 applications. For details, read the section on connection handles in Chapter 3.

This function connects to a database with recovery (transaction logging) turned off and issues a cursor handle that is associated with a single, implicit connection to a database. All cursors that you connect to this database belong to a single transaction and to the same implicit connection handle.

You must understand the implications of this function. When recovery is turned off, transaction logging is not performed and transaction rollbacks are not possible.

This function can connect to a new database or connect a new cursor to the current database.

Turning off recovery has an effect only when it is the first connect to the database. All subsequent connects to this database by *any* user must be done with *sqlcnr*. If a user tries a subsequent connect with *sqlcnc*, they will get an error.

Parameters

curp

A pointer to the variable where this function returns the cursor handle.

dbnamp

A pointer to the connect string that contains the database name, username, and password separated by forward slashes:

```
database/username/password
```

These rules are used:

- The characters before the first forward slash are the database name.
- Any characters after the first forward slash and before the second forward slash are the username.
- Any characters after a second forward slash are the password.

If the database name, username, or password is not specified, then the system uses the current default. For example, you can specify the following connect string in which the default database name and username are used:

```
//password
```

The default database name, username, and password are determined by:

- The *defaultdatabase*, *defaultuser*, and *defaultpassword* keywords in *sql.ini*.
- The default of DEMO/SYSADM/SYSADM.

dbnaml

The length of the string pointed to by *dbnamp*. If the string pointed to by *dbnamp* is null-terminated, specify zero and the system will compute the length.

Return value

If this function returns zero, it was successful. If this function returns a non-zero value, it was unsuccessful.

Example

```
SQLTCUR  cur;/* Cursor handle */
SQLTRCD  rcd;/* Return code */

if (rcd = sqlcnr(&cur, "PAYROLL/BOSS/SECRET", 0))
  {
     printf("Failure on connect (rcd = %d)\n",rcd);
     exit(0);
  }
else
  printf("Connection with recovery turned off\n");
```

Related functions

sqlcnc *sqldis*

sqlcom - COMpile a SQL command/procedure

Syntax

```
#include <sql.h>

SQLTAPI   sqlcom (cur,cmdp,cmdl);

SQLTCUR   cur;     /* Cursor handle */
SQLTDAP   cmdp;    /* SQL command or procedure*/
SQLTDAL   cmdl;    /* Length of SQL command */
```

Description

This function compiles a SQL command or non-stored procedure and stores it in the work space associated with the cursor. No data is bound. After a SQL command or procedure has been compiled, it can be executed or stored.

There are 3 steps in compiling:

1. Parse:
   - Check that the command or procedure is formulated correctly.
   - Break the statement into components for the optimizer.
   - Verify names of columns and tables in the system catalog.
2. Optimize:
   - Replace view column names and table names with real names.
   - Gather statistics on data storage from the system catalog.
   - Identify possible access paths.
   - Calculate the cost of each alternate path.
   - Choose the best path.
3. Generate execution code:
   - Produce an application plan for execution.

All compiled commands for all cursors that the program has connected to the database are destroyed by:

- Commits (explicit or implicit, including implicit by autocommit or by change in isolation level).
- Rollbacks (including rollbacks caused by a deadlock).

Note: You cannot compile a procedure as static before storing it with the *sqlsto* function.

Parameters

cur

The cursor handle associated with this function.

For these SQL commands, use the server handle returned by *sqlcsv* instead:

ALTER DATABASE
ALTER DBAREA
ALTER STOGROUP
CREATE DATABASE
CREATE DBAREA
CREATE STOGROUP
DEINSTALL DATABASE
DROP DATABASE
DROP DBAREA
DROP STOGROUP
INSTALL DATABASE
SET DEFAULT STOGROUP

cmdp

A pointer to the string that contains the SQL command.

cmdl

The length of the string pointed to by *cmdp*. If the string pointed to by *cmdp* is null-terminated, specify zero and the system will compute the length.

Return value

If this function returns zero, it was successful. If this function returns a non-zero value, it was unsuccessful.

Example

```
static char sqlcmd[] =
  "SELECT A, B FROM TAB WHERE C = :1";
ret = sqlcom(cur, sqlcmd, 0);
```

Related functions

sqlcex *sqlexe* *sqlsto*
sqlcsv

sqlcpy - CoPY data from one table to another

Syntax

```
#include <sql.h>

SQLTAPI   sqlcpy (fcur, selp, sell, tcur, isrtp, isrtl)

SQLTCUR  fcur;    /* Cursor handle for SELECT */
SQLTDAP  selp;    /* SELECT command */
SQLTDAL  sell;    /* Length of SELECT command */
SQLTCUR  tcur;    /* Cursor handle for INSERT */
SQLTDAP  isrtp;   /* INSERT command */
SQLTDAL  isrtl;   /* Length of INSERT command */
```

Description

This function copies data from one table to another. The destination table must exist and the data type of the destination columns must be compatible with the data in the corresponding source columns. For example, you cannot copy alphabetic data to a numeric column. The source table and the destination table can be in different databases.

This function needs two cursors: one for the SELECT command that retrieves the data from the source table, and one for an INSERT command that adds rows to the target table.

The application must issue COMMITs following a transaction that uses this function to ensure that changes are made permanent.

Each item in the SELECT statement must correspond on a one-to-one basis with each bind variable in the INSERT command. For example, bind variable *:1* corresponds to the first SELECT list item and bind variable *:2* corresponds to the second SELECT list item.

Parameters

fcur

The cursor handle associated with the SELECT command.

selp

A pointer to the string that contains the SELECT command that retrieves data from the source table.

sell

The length of the string pointed to by *selp*. If the string pointed to by *selp* is null-terminated, specify zero and the system will compute the length.

tcur

The cursor handle associated with the INSERT command.

isrtp

A pointer to the string that contains the INSERT command that adds the selected data to the target table.

isrtl

The length of the string pointed to by *isrtp*. If the string pointed to by *isrtp* is null-terminated, specify zero and the system will compute the length.

Return value

If this function returns zero, it was successful. If this function returns a non-zero value, it was unsuccessful.

The error returned by this function does not indicate the cursor that caused the error. Check the return code for each cursor to establish the source of the error.

Example

```
SQLTCUR  cur1 = 0;   /* select cursor */
SQLTCUR  cur2 = 0;   /* insert cursor */
SQLTRCD  rcd1 = 0;   /* return code (cur1) */
SQLTRCD  rcd2 = 0;   /* return code (cur2) */

main()
{
static  char  select[] =      /* SQL select statement */
  "SELECT EMP_NO,EMP_NAME FROM EMP";
static  char  insert[] =       /* SQL insert statement */
  "INSERT INTO TMP (TMP_NO, TMP_NAME) VALUES (:1, :2)";

/* CONNECT TO BOTH CURSORS */

if (sqlcnc(&cur1, "DEMO", 0))
```

```
  failure("SELECT CURSOR CONNECT");

if (sqlcnc(&cur2, "DEMO", 0))
  failure("INSERT CURSOR CONNECT");

/* PERFORM COPY OPERATION */

if (sqlcpy(cur1,select,0,cur2,insert,0))
  failure("COPY OPERATION");

/* COMMIT BOTH CURSORS */

if (sqlcmt(cur1) || sqlcmt(cur2))
  failure("COMMIT");

/* DISCONNECT BOTH CURSORS */

if (sqldis(cur1))
  failure("DISCONNECT OF SELECT CURSOR");

cur1 = 0;

if (sqldis(cur2))
  failure("DISCONNECT OF INSERT CURSOR");
}

failure(ep)
char*  ep; /* -> failure msg string */
{
  SQLTEPO  epo;                  /* error position */
  char  errmsg[SQLMERR];    /* error msg text buffer */

printf("Failure on %s \n", ep);
sqlrcd(cur1,&rcd1);        /* get return codes */
sqlrcd(cur2,&rcd2);

if (rcd1)                       /* error on cursor 1? */
  {
    sqlerr(rcd1, errmsg);
    sqlepo(cur1, &epo);
    printf("%s(error: %u, position: %u) \n",errmsg,rcd1,epo);
  }
if (rcd2)                      /* error on cursor 2? */
  {
    sqlerr(rcd2, errmsg);
    sqlepo(cur2, &epo);
```

```
   printf("%s(error: %u, position: %u)
\n",errmsg,rcd1,epo);
  }

if (cur1)                    /* cursor 1 exists? */
  sqldis(cur1);

if (cur2)                    /* cursor 2 exists? */
  sqldis(cur2);

exit(1);
}
```

sqlcre - CREate database

Syntax

```
#include <sql.h>

SQLTAPI  sqlcre (shandle, dbname, dbnamel)

SQLTSVH  shandle;  /* Server handle */
SQLTDAP  dbname;   /* Database name */
SQLTDAL  dbnamel;  /* Database name length */
```

Description

This function physically creates a database on the specified server, returns an error if the database already exists, and adds the *dbname* keyword to *sql.ini*.

In SQLBase, a *database* contains a database file placed in a sub-directory. The database file must have the extension *.dbs*, for example, *demo.dbs*. The name of the sub-directory must be the same as the database file name without the extension, for example, *\demo*.

Do not specify an extension for a database name (*demo.xyz* is invalid). SQLBase automatically assigns a database name extension of *.dbs*.

Usually the database sub-directory is placed in the *\Centura* directory. This is the default, but it can be set to any location using the *dbdir* keyword in *sql.ini*.

The maximum length of the database name is 8 characters.

Parameters

shandle

The server handle returned by *sqlcsv*.

dbname

A pointer to the string that contains the database name.

dbnamel

The length of the string pointed to by *dbname*. If the string pointed to by *dbname* is null-terminated, specify zero and the system will compute the length.

Return value

If this function returns zero, it was successful. If this function returns a non-zero value, it was unsuccessful.

Example

```
main()
{
  srvname = "SERVER1";
  password = 0;

/* CONNECT TO THE SERVER */

if (rcd = sqlcsv(&handle,srvname,password))
  apierr("SQLCSV");
else
  printf("Connection Established to Server \n");

/* CREATE DATABASE */

if (rcd = sqlcre(handle,"DEMOX",0))
  apierr("SQLCRE");
else
  printf("Database DEMOX Created \n");

/* DISCONNECT FROM THE SERVER */

if (rcd = sqldsv(handle))
  apierr("SQLDSV");
else
  printf("Disconnected from Server \n");
}
```

Related functions

sqlcsv *sqldel* *sqlind*
sqlded

sqlcrf - Continue RollForward

Syntax

```
#include <sql.h>

SQLTAPI  sqlcrf (shandle, dbname, dbnamel)

SQLTSVH  shandle;  /* Server handle */
SQLTDAP  dbname;   /* Database name */
SQLTDAL  dbnamel;  /* Length of database name */
```

Description

Call this function after a rollforward operation has stopped because it cannot open the next transaction log file.

Ordinarily, the *sqlrlf* function is used to restore the logs and *sqlcrf* is used to continue the rollforward. However, you can also restore the logs directly to the log directory using other means such as a tape backup or optical disk. If this is done, you must call this function explicitly to continue the rollforward process.

Parameters

shandle

The server handle returned by *sqlcsv*.

dbname

A pointer to the string that contains the database name.

dbnamel

The length of the string pointed to by *dbname*. If the string pointed to by *dbname* is null-terminated, specify zero and the system will compute the length.

Return value

If this function returns zero, it was successful. If this function returns a non-zero value, it was unsuccessful.

Example

```
static char dbname1[]=”omed”;
/* RESTORE DATABASE */

if (rcd =
sqlrdb(shandle,dbname1,0,bkpdir,bkpdirl,local,over))
  apierr("SQLRDB");
else
  printf("Restored Database \n");

/* ROLLFORWARD TO END */

sqlrof(shandle,dbname1,0,mode,0,0);

/* RESTORE LOGS USING OPERATING SYSTEM COPY */

system("DEL \\centura\\omed\\*.log");
system("COPY \\backup\\omed\\*.log \\centura\\omed");

/* CONTINUE ROLLFORWARD */

sqlcrf(shandle,dbname1,0);

/* END ROLLFORWARD */

if (rcd = sqlenr(shandle,dbname1,0))
  apierr("SQLENR");
else
  printf("End Rollforward \n");
```

Related functions

sqlbdb	*sqlenr*	*sqlrlf*
sqlblf	*sqlgnl*	*sqlrof*
sqlbss	*sqlrdb*	*sqlrss*
sqlcsv	*sqlrel*	

sqlcrs - Close ReStriction and Result Set modes

Syntax

```
#include <sql.h>

SQLTAPI  sqlcrs (cur,rsp,rsl);

SQLTCUR  cur;   /* Cursor handle */
SQLTDAP  rsp;   /* Result set name */
SQLTDAL  rsl;   /* Result set name length */
```

Description

This functions turns off both result set mode and restriction mode.

This function lets you optionally save the result set by specifying a name in *rsp*. To use a saved result set later, call the *sqlrrs* function and specify the saved result set name. The *sqlrrs* function turns on result set mode and restriction mode.

The *sqldrs* function drops a saved result set.

Be cautious about using saved result sets. Internally, a saved result set is a list of row identifiers (ROWIDs) that is stored in the SYSROWIDLISTS system catalog table. A ROWID changes whenever the row is updated. If one of the rows is updated after you have saved and closed a result set, you will get an error if you open the result set later and try to fetch the row.

Parameters

cur

The cursor handle associated with this function.

rsp

A pointer to the string that contains the name of the result set. Specify a null string (SQLNPTR) to close the result set without saving it.

rsl

The length of the string pointed to by *rsp*. If the string pointed to by *rsp* is null-terminated, specify zero and the system will compute the length. If the result set is not being saved, specify zero.

Return value

If this function returns zero, it was successful. If this function returns a non-zero value, it was unsuccessful.

Example

```
/* Save current result set as "saveres" */

ret = sqlcrs(cur, "saveres", 0);
```

Related functions

sqlcrs	*sqlrrs*	*sqlsrs*
sqldrs	*sqlscn*	*sqlstr*
sqlprs	*sqlspr*	*sqlurs*

sqlcsv - Connect to SerVer

Syntax

```
#include <sql.h>

SQLTAPI       sqlcsv (handlep, serverid, password)

SQLTSVH  PTR  handlep;/* Returned server handle */
SQLTDAP       serverid;/* Null-terminated server identifier */
SQLTDAP       password;/* Null-terminated server password */
```

Description

This function connects a user to a server to perform administrative operations.

This function returns a server handle that is required for the following administrative operations:

- Create or delete a database
- Install or deinstall a database
- Backup or restore a database
- Backup or restore log files
- Initiate rollforward recovery
- Abort a server process

- Terminate the server

Parameters

handlep

A pointer to the variable where this function returns the server handle.

serverid

A pointer to the null-terminated string that contains the name of the server.

The server name is set by the *servername* keyword in *sql.ini*. The maximum length of a server name is 8 characters. The server name must begin with a letter.

password

A pointer to the null-terminated string that contains the server password.

The *password* keyword in *sql.ini* sets a password for a server. This keyword follows a *server* keyword in *sql.ini.*

If the server password is set, a case-insensitive comparison is performed between the server password and the *sqlcsv* password.

The maximum length of a server password is 8 characters.

Return value

If this function returns zero, it was successful. If this function returns a non-zero value, it was unsuccessful.

Example

```
main()
{
  srvname = "SERVER1";
  password = 0;

/* CONNECT TO THE SERVER */

if (rcd = sqlcsv(&handle,srvname,password))
  apierr("SQLCSV");
else
  printf("Connection Established to Server \n");

/* DISCONNECT FROM THE SERVER */
```

```
if (rcd = sqldsv(handle))
  apierr("SQLDSV");
else
  printf("Disconnected from Server \n");
```

Related functions

All of the functions below require the server handle returned by *sqlcsv*.

sqlbdb	*sqldrr*	*sqlmop*
sqlblf	*sqldsv*	*sqlmrd*
sqlbss	*sqlenr*	*sqlmsk*
sqlclf	*sqlfgt*	*sqlmwr*
sqlcre	*sqlfpt*	*sqlrdb*
sqlcrf	*sqlgnl*	*sqlrlf*
sqlded	*sqlgsi*	*sqlrof*
sqldel	*sqlind*	*sqlrss*
sqldrc	*sqlmcl*	*sqlsab*
sqldro	*sqlmdl*	*sqlstm*

sqlcty - Command TYpe

Syntax

```
#include <sql.h>

SQLTAPI      sqlcty (cur, cty);

SQLTCUR      cur;   /* Cursor handle */
SQLTCTY PTR  cty;   /* Variable */
```

Description

This function returns the command type of the SQL command currently being processed. The command type is set after *sqlcom* or *sqlexe*.

Parameters

cur

The cursor handle associated with this function.

cty

A pointer to the variable where this function returns the command type. For example, if the previously-compiled command was an UPDATE, this function returns 4. The command types are defined in *sql.h*.

Note that *sqlcty* returns the SQLTSEL command type for either a SELECT or PROCEDURE command that is compiled and current. Either command can return results to *sqlfet*. To determine the actual command type, use the *sqlget* function in conjunction with the SQLPWFC parameter. See the documentation for *sqlget* for more information.

Identifier in sql.h	Value returned in cty	Operation
SQLTSEL	1	SELECT or PROCEDURE
SQLTINS	2	INSERT
SQLTCTB	3	CREATE TABLE
SQLTUPD	4	UPDATE
SQLTDEL	5	DELETE
SQLTCIN	6	CREATE INDEX
SQLTDIN	7	DROP INDEX
SQLTDTB	8	DROP TABLE
SQLTCMT	9	COMMIT
SQLTRBK	10	ROLLBACK
SQLTACO	11	Add column
SQLTDCO	12	Drop column
SQLTRTB	13	Rename table
SQLTRCO	14	Rename column
SQLTMCO	15	Modify column
SQLTGRP	16	GRANT privilege on table
SQLTGRD	17	GRANT DBA
SQLTGRC	18	GRANT CONNECT

Identifier in sql.h	Value returned in cty	Operation
SQLTGRR	19	GRANT RESOURCE
SQLTREP	20	REVOKE privilege on table
SQLTRED	21	REVOKE DBA
SQLTREC	22	REVOKE CONNECT
SQLTRER	23	REVOKE RESOURCE
SQLTCOM	24	COMMENT ON
SQLTWAI	25	Wait
SQLTPOS	26	Post
SQLTCSY	27	CREATE SYNONYM
SQLTDSY	28	DROP SYNONYM
SQLTCVW	29	CREATE VIEW
SQLTDVW	30	DROP VIEW
SQLTRCT	31	Row count
SQLTAPW	32	ALTER PASSWORD
SQLTLAB	33	LABEL ON
SQLTCHN	34	Chained command
SQLTRPT	35	Repair table
SQLTSVP	36	SAVEPOINT
SQLTRBS	37	ROLLBACK to savepoint
SQLTUDS	38	UPDATE STATISTICS
SQLTCDB	39	CHECK DATABASE
SQLTFRN	40	Non-SQLBase DBMS commands
SQLTAPK	41	Add primary key
SQLTAFK	42	Add foreign key
SQLTDPK	43	Drop primary key

Identifier in sql.h	Value returned in cty	Operation
SQLTDFK	44	Drop foreign key
SQLTCDA	45	CREATE DBAREA
SQLTADA	46	ALTER DBAREA
SQLTDDA	47	DROP DBAREA
SQLTCSG	48	CREATE STOGROUP
SQLTASG	49	ALTER STOGROUP
SQLTDSG	50	DELETE STOGROUP
SQLTCRD	51	CREATE DATABASE
SQLTADB	52	ALTER DATABASE
SQLTDDB	53	DROP DATABASE
SQLTSDS	54	SET DEFAULT STOGROUP
SQLTIND	55	INSTALL DATABASE
SQLTDED	56	DEINSTALL DATABASE
SQLTARU	57	Add referential integrity user error
SQLTDRU	58	Drop referential integrity user error
SQLTMRU	59	Modify referential integrity user error
SQLTSCL	60	Set client
SQLTCKT	61	CHECK TABLE
SQLTCKI	62	CHECK INDEX
SQLTOPL	63	PL/SQL Stored Procedure
SQLTUNL	85	UNLOAD
SQLTLDP	86	LOAD
SQLTPRO	87	Stored procedure
SQLTGEP	88	GRANT EXECUTE ON
SQLTREE	89	REVOKE EXECUTE ON

Identifier in sql.h	Value returned in cty	Operation
SQLTTGC	90	CREATE TRIGGER
SQLTTGD	91	DROP TRIGGER
SQLTVNC	92	CREATE EVENT
SQLTVND	93	DROP EVENT
SQLTSTR	94	START AUDIT
SQLTAUD	95	AUDIT MESSAGE
SQLTSTP	96	STOP AUDIT
SQLTACM	97	ALTER COMMAND
SQLTXDL	98	LOCK DATABASE
SQLTXDU	99	UNLOCK DATABASE
SQLTDBT	102	DBATTRIBUTE
SQLTATG	103	ALTER TRIGGER

Return value

If this function returns zero, it was successful. If this function returns a non-zero value, it was unsuccessful.

Example

```
int    cmnd;  /* command type */
short  ret;   /* return code */

ret = sqlcty(cur, &cmnd);
```

Related functions

sqlcom *sqlexe*

sqldbn - DataBase Names

Syntax

```
#include <sql.h>

SQLTAPI  sqldbn (serverid, buffer, length)

SQLTDAP  serverid;   /* Server identifier */
SQLTDAP  buffer;     /* Directory list */
SQLTDAL  length;     /* Buffer length */
```

Description

This function returns a list of the databases on the specified server.

Use this function instead of *sqldir*.

Parameters

serverid

A pointer to a null-terminated string that contains the name of the server specified in *sql.ini*. Specify a null server name to get a directory of local databases.

buffer

A pointer to the variable where this function returns the database names.

Each name is null-terminated. The end of the list is marked by an extra null-terminator. For example, the database names *demo*, *payables*, and *emp* are returned in this format:

```
demo\0payables\0emp\0\0
```

length

The length of the area pointed to by *buffer*.

Return value

If this function returns zero, it was successful. If this function returns a non-zero value, it was unsuccessful.

Example

```
main()
{
  srvname="SERVER1";

/* DIRECTORY OF DATABASES */

if (rcd = sqldbn(srvname,buffer,len))
  apierr("SQLDBN");
else
    {
      j = 0;
      printf("Directory of Databases : ");
      while ( (buffer[j] != '\n') && (j< 20) )
        {
          if (buffer[j] == '\0')
{
              printf(", ");
}
          else
{
              printf("%c",buffer[j]);
}
          j++;
        }
      printf("\n");
    }
  }
```

Related functions

sqldir

sqldch - Destroy Connection Handle

Syntax

```
#include <sql.h>

SQLTAPI   sqldch (hCon);

SQLTCNH   hCon;   /* Connection handle */
```

Description

This function terminates a specific connection. Before terminating a connection, it is good programming practice to commit the transaction and close all open cursors. This function automatically closes any open cursors before destroying the connection handle.

When terminating a connection, this function commits or rolls back the current transaction before terminating the connection. By default, *sqldch* will COMMIT the transaction for a SQLBase server before terminating the connection. For the default behavior of servers other than SQLBase, read your applicable server documentation.

To modify the default connect closure behavior for both SQLBase and non-SQLBase servers, use the *sqlset*() function call with the SQLPCCB parameter. For details, read information on the *sqlset* function in this chapter.

Parameters

hCon

The handle to the connection to be terminated.

Return value

If this function returns zero, it was successful. If this function returns a non-zero value, it was unsuccessful.

Example

```
if (rcd = sqldch(hCon))
{
  printf("Failure terminating connection (rcd = %d)\n", rcd);
  exit(0);
```

```
}
else printf("Connection terminated\n");
```

Related functions

sqlcch *sqlopc* *sqldis*

sqlded - DEinstall Database

Syntax

```
#include <sql.h>

SQLTAPI   sqlded (shandle, dbname, dbnamel)

SQLTSVH   shandle;    /* Server handle */
SQLTDAP   dbname;     /* Database name */
SQLTDAL   dbnamel;    /* Database name length */
```

Description

This function:

- Deinstalls the specified database from the network.
- Removes the *dbname* keyword from *sql.ini*.

This function does *not* physically delete the database.

You cannot deinstall a database that has a user connected.

Parameters

shandle

The server handle returned by *sqlcsv*.

dbname

A pointer to the string that contains the database name.

dbnamel

The length of the string pointed to by *dbname*. If the string pointed to by *dbname* is null-terminated, specify zero and the system will compute the length.

Return value

If this function returns zero, it was successful. If this function returns a non-zero value, it was unsuccessful.

Example

```
main()
{
   srvname = "SERVER1";
   password = 0;

/* CONNECT TO THE SERVER */

if (rcd = sqlcsv(&handle,srvname,password))
   apierr("SQLCSV");
else
   printf("Connection Established to Server \n");

if (rcd = sqlcre(handle,"DEMOX",0))
   apierr("SQLCRE");
else
   printf("Database DEMOX Created \n");

/* DEINSTALL DATABASE */

if (rcd = sqlded(handle,"DEMOX",0))
   apierr("SQLDED");
else
   printf("Database DEMOX Deinstalled \n");

/* INSTALL DATABASE */

if (rcd = sqlind(handle,"DEMOX",0))
   apierr("SQLIND");
else
   printf("Database DEMOX Installed \n");

/* DISCONNECT FROM THE SERVER */

if (rcd = sqldsv(handle))
   apierr("SQLDSV");
else
   printf("Disconnected from Server \n");
}
```

Related functions

sqlcre *sqldel* *sqlind*
sqlcsv

sqldel - DELete database

Syntax

```
#include <sql.h>

SQLTAPI   sqldel (shandle, dbname, dbnamel)

SQLTSVH   shandle;    /* Server handle */
SQLTDAP   dbname;     /* Database name */
SQLTDAL   dbnamel;    /* Database name length */
```

Description

This function physically deletes the entire database directory for a database *including* all associated transaction log files on the server. If the log is redirected, the log directory for the database is also completely removed.

This function removes the *dbname* keyword from *sql.ini*.

Parameters

shandle

The server handle returned by *sqlcsv*.

dbname

A pointer to the string that contains the database name.

dbnamel

The length of the string pointed to by *dbname*. If the string pointed to by *dbname* is null-terminated, specify zero and the system will compute the length.

Return value

If this function returns zero, it was successful. If this function returns a non-zero value, it was unsuccessful.

Example

```
main()
{
    srvname = "SERVER1";
    password = 0;

/* CONNECT TO THE SERVER */

if (rcd = sqlcsv(&handle,srvname,password))
    apierr("SQLCSV");
else
    printf("Connection Established to Server \n");

/* DELETE DATABASE */

if (rcd = sqldel(handle,"DEMOX",0))
    apierr("SQLDEL");
else
    printf("Database DEMOX Deleted \n");

/* DISCONNECT FROM THE SERVER */

if (rcd = sqldsv(handle))
    apierr("SQLDSV");
else
    printf("Disconnected from Server \n");
}
```

Related functions

sqlcre *sqlded* *sqlind*
sqlcsv

sqldes - DEScribe items in a SELECT list

Syntax

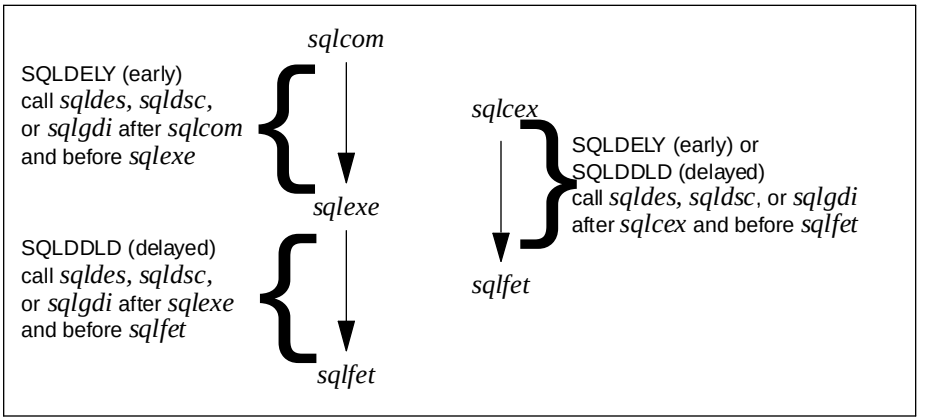
```
#include <sql.h>

SQLTAPI  sqldes (cur, slc, ddt, ddl, chp, chlp, prep, scap)

SQLTCUR  cur;      /* Cursor handle */
SQLTSLC  slc;      /* Select column number */
SQLTDDT  PTRddt;   /* Database data type */
SQLTDDL  PTRddl;   /* Database data length */
SQLTCHP  chp;      /* Column heading buffer */
SQLTCHL  PTRchlp;  /* Column heading length */
SQLTPRE  PTRprep;  /* Numeric precision */
SQLTSCA  PTRscap;  /* Numeric scale */
```

Description

This function returns the *database* data type and length for a column in a SELECT list.

This function differs from *sqldsc* which returns the *external* data type and length. The external data type is defined in the SYSCOLUMNS system catalog table. External data types match program data types in *sql.h*.

The following diagram shows how the value of the SQLPDIS parameter (SQLDELY, SQLDDLD, or SQLDNVR) controls when (and if) describe information for a SELECT statement is available for sending to a client. You can specify the SQLPDIS parameter's value using the *sqlset* function.



When describe information is available, given the different SQLPDIS parameter settings.

This table summarizes the information illustrated above:

SQLPDIS constant	Value	When describe information is available
SQLDELY early (default)	0	The server sends describe information after *sqlcom*; subsequent calls to *sqldes*, *sqldsc*, or *sqlgdi* are legal until after a call to *sqlexe*. The server also sends describe information after *sqlcex*; subsequent calls to *sqldes*, *sqldsc*, or *sqlgdi* are legal until after a call to *sqlfet*.
SQLDDLD delayed	1	The server sends describe information after *sqlexe*. Calling *sqldes*, *sqldsc*, or *sqlgdi* after calling *sqlexe* but before the first *sqlfet* is legal; calling *sqldes*, *sqldsc*, or *sqlgdi* at any other time is illegal. The server also sends describe information after *sqlcex*; subsequent calls to *sqldes*, *sqldsc*, or *sqlgdi* are legal until after a call to *sqlfet*. Use this setting to reduce message traffic for database servers that do not support compile (*sqlcom*) operations (like Microsoft's SQL Server).

SQLPDIS constant	Value	When describe information is available
SQLDNVR never	2	The server *never* sends describe information; any call to *sqldes*, *sqldsc*, or *sqlgdi* is illegal. When SQLPDIS is set to SQLDNVR, *sqlnsi* always returns 0. You must hard code the number of SELECT items so that the application knows how many times to call *sqlssb*. Use this setting to reduce message traffic when the application always knows the number and type of columns in a SELECT statement and never makes calls to *sqldes*, *sqldsc*, or *sqlgdi*.

You can pass null pointers (SQLNPTR) for arguments that you do not want.

You can retrieve the number of columns in the SELECT list with the *sqlnsi* function call.

Parameters

cur

The cursor handle associated with this function.

slc

The column number (starting with 1) in the SELECT list about which to return information.

ddt

A pointer to the variable where this function returns the database data type of the column.

Number	Typedef in sql.h	Data Type
1	SQLDCHR	Character
2	SQLDNUM	Number
3	SQLDDAT	Date/time
4	SQLDLON	Long
5	SQLDDTE	Date (only)
6	SQLDTIM	Time (only)

ddl

A pointer to the variable where this function returns the length of the column.

Data Type	Returns
Character	Size specified when column was defined
Numeric	27 (22 digits of precision plus room for scientific notation)
Date/time	26
Long	0
Date (only)	10
Time (only)	15

Note that the length returned for numeric and date/time columns are for display and printing. Use the *sqldsc* function to get the length as stored in SQLBase's internal format.

chp

A pointer to the variable where this function returns the column heading defined in the SYSCOLUMNS system catalog table.

chlp

A pointer to the variable where this function returns the length of the string pointed to by *chp*.

prep

A pointer to the variable where this function returns the precision of a numeric column.

scap

A pointer to the variable where this function returns the scale, if any, of a numeric column.

Return value

If this function returns zero, it was successful. If this function returns a non-zero value, it was unsuccessful.

Example

```
static  char  select[] = "SELECT * FROM TEST";

char             ddt;          /* Datatype */
char             colnam[50];   /* Column heading buffer */
unsigned char    i;
unsigned char    ddl;          /* Data length */
int              prec, scale;  /* Precision, scale */
int              hdl;          /* Heading length */
uchar            nsi;          /* Number of SELECT items */

sqlnsi(cur, &nsi);
for (i = 1; i <= nsi; i++)
   {
      memset(colnum, '\0', 50);
      if sqldes(cur,i,&ddt,&ddl,colnam,&hdl,&prec,&scale))
            {
.              .. process error
            }
   printf("%d %d %s %d %d %d\n", ddt, ddl, colnam, hdl,
          prec, scale);
   }
```

Related functions

sqlcom *sqldsc* *sqlexe*
sqlgdi *sqlnsi*

sqldii - Describe Into variable

Syntax

```
#include <sql.h>

SQLTAPI  sqldii (cur, ivn, inp, inl);

SQLTCUR  cur;      /* Cursor handle*/
SQLTSLC  ivn;      /* INTO variable position number */
SQLTDAP  inp;      /* INTO variable name */
SQLTCHL  PTR inl;  /* INTO variable name length*/
```

Description

This function describes an INTO variable.

Parameters

cur

The cursor handle associated with this function.

ivn

The relative position of the INTO variable, starting at 1.

inp

A pointer to the string that contains the name of the INTO variable.

inl

A pointer to the variable where this function returns the length of the INTO variable's name.

Return value

If this function returns zero, it was successful. If this function returns a non-zero value, it was unsuccessful.

Example

```
#include "sql32.h"
#include <memory.h>
#include <stdio.h>
#include <stdlib.h>

/*------------------------------------------------------- */
/*                                                       */
/* Example of a simple fetch                             */
/*                                                       */
/* Run EMP.SQL via SQLTALK to initialize tables and data */
/*                                                       */
/*-------------------------------------------------------*/

SQLTCUR cur;                /* SQLBASE cursor number*/
SQLTRCD rcd;                /* error number */
char    errmsg[SQLMERR];    /* error msg text buffer*/
void    failure(char*);     /* error routine*/
```

```
main()
{
    char    name[20];/* employee name buffer     */
    SQLTCHL PTR nii;
    SQLTCHL PTR inl;
    SQLTSLC ivn;
    SQLTDAP inp;
    static  char selcmd [] = /* SQL SELECT statement   */
       "SELECT EMP_NAME into :name FROM EMP  ";
      /*
        CONNECT TO THE DATABASE
      */

      if (rcd = sqlcnc(&cur, "ISLAND", 0))
      {
        sqlerr(rcd, errmsg);/* get error message text */
        printf("%s \n",errmsg);
        return(1);
      }

      /*
        COMPILE SELECT STATEMENT
      */

      if (sqlcom(cur, selcmd, 0))
        failure("SELECT COMPILE");

      /*
        PERFORM sqldii
      */

      if (sqldii(cur,1,name,inl))
        failure ("SQLDII");
      else
        printf("The length of the into variable is
        %d\n",*inl);

      /*
        SET UP SELECT BUFFER
      */

      if (sqlssb(cur, 1, SQLPBUF, name, 20, 0, SQLNPTR,
      SQLNPTR))
        failure("SET SELECT BUFFER");

      /*
        EXECUTE SELECT STATEMENT
```

```
*/

if (sqlexe(cur))
  failure("EXECUTING SELECT");

/*
  FETCH DATA
*/
for (;;)
{
  memset(name,' ',20);   /* clear employe name buf */

  if (rcd = sqlfet(cur)) /* fetch the data         */
    break;
  printf("%s\n", name);  /* print employe name    */
}

if (rcd != 1)           /* failure on fetch */
   failure("FETCH");

/*
  DISCONNECT FROM THE DATABASE
*/

if (rcd = sqldis(cur))
   failure("DISCONNECT");
}

void failure(ep)
char*          ep;    /* failure msg string  */
{
   SQLTEPO   epo;    /* error position   */

   printf("Failure on %s \n", ep);

   sqlrcd(cur, &rcd);    /* get the error        */
   sqlepo(cur, &epo);    /* get error position     */
   sqlerr(rcd, errmsg);  /* get error message text  */

   sqldis(cur);          /* disconnect cursor      */

   printf("%s (error: %u, position: %u)
   \n",errmsg,rcd,epo);
   exit(1);
}
```

Related functions

sqlnii

sqldir - DIRectory of databases

Syntax

```
#include <sql.h>

SQLTAPI  sqldir (svrno, buffer, length)

SQLTSVN  svrno;    /* Server number */
SQLTDAP  buffer;   /* Database names */
SQLTDAL  length;   /* Length of buffer */
```

Description

This function returns a list of database names on the specified server.

This function is provided for backwards compatibility with earlier versions of SQLBase. When creating new applications, do not use this function; use the *sqldbn* function instead.

Parameters

svrno

A numeric literal that specifies the server. The system appends this literal to "server" to form the server name set in *sql.ini*.

Specify a zero to return a list of local databases.

buffer

A pointer to the variable where this function returns the database names. Each name is null-terminated. The end of the list is marked by an extra null-terminator. For example, the database names *demo*, *payables* and *emp* are returned in this format:

```
demo\0payables\0emp\0\0
```

length

The length of the value pointed to by *buffer*. The list of database names is truncated if *buffer* is not large enough.

Return value

If this function returns zero, it was successful. If this function returns a non-zero value, it was unsuccessful.

Example

```
char buf[100];  /* database directory buffer */
short ret;      /* return code */
short srvr;     /* server number */

srvr = 1;

ret = sqldir(srvr, buf, sizeof(buf));
if (ret)
    ... process error
```

Related functions

sqldbn

sqldis - DISconnect from cursor

Syntax

```
#include <sql.h>

SQLTAPI   sqldis (cur);

SQLTCUR   cur;    /* Cursor handle */
```

Description

This function disconnects a cursor. If you are closing the final cursor, note the difference in behavior between cursors connected through implicit, or explicit connections. For details, read the section *Connection Handles*, in Chapter 3, *Using the SQL/API.*

If you are disconnecting a final cursor that is connected implicitly with the *sqlcnc* or the *sqlcnr* function, a COMMIT is performed before the cursor is disconnected. If you are using the *sqlcnc()* function call, you can use the *sqlset()* API function call with the SQLPCCB parameter and specify the ROLLBACK option. When this is set, a roll back is performed before the cursor is disconnected.

If you are disconnecting a final cursor that is connected explicitly with the *sqlopc* function, the cursor remains pending and is not automatically committed. Note that cursors connected with the *sqlopc* function belong to a specific connection handle. Each connection handle represents a single transaction and its connection to a single database. The transaction is either committed or rolled back only when the connection handle is terminated using the *sqldch* function call.

You can specify whether a transaction is rolled back or committed by:

- using the *sqlset*() API function call with the SQLPCCB parameter. By default, the setting is server dependent and in the case of SQLBase the DEFAULT is COMMIT.
- explicitly executing a COMMIT, ROLLBACK, *sqlcmt*(), or *sqlrbk*() when the connection handle is terminated.
- setting the connect closure behavior to ROLLBACK when opening a cursor with the *sqlopc*() function call.

Parameters

cur

A cursor handle of cursor to be closed.

Return value

If this function returns zero, it was successful. If this function returns a non-zero value, it was unsuccessful.

Example

```
if(rcd = sqldis(cur))
{
  printf("Failure closing cursor (rcd = %d)\n", rcd);
  exit(0);
}
else printf("Cursor closed\n");
```

Related functions

sqlcch *sqlopc* *sqldch*

sqldon - DONe

Syntax

```
#include <sql.h>

SQLTAPI   sqldon ( )
```

Description

This function does a rollback and disconnects *all* open cursors.

This function is often used in conjunction with *sqlini*. If *sqlini* was called, *sqldon* must be called before the program exits to free allocated resources.

See *testwin.c* for an example of how to use this function. This online file is provided with your SQLBase shipment.

Return value

If this function returns zero, it was successful. If this function returns a non-zero value, it was unsuccessful.

Example

```
int PASCAL WinMain (hInstance, hPrevInstance, lpszCmdLine,
                cmdShow)

   HANDLE    hInstance;
   HANDLE    hPrevInstance;
   LPSTR     lpszCmdLine;
   int       cmdShow;
{
   short  ret;   /* return code */

   extern int far pascal yieldpgm();
   sqlini(MakeProcInstance(yieldpgm,hInstance));
   ...
   if (ret = sqldis(cur))    /* disconnect */
      ... process error

   sqldon();    /* Disconnect all cursors */
   return;
}
```

Related functions

sqlini

sqldox - Directory Open eXtended

Syntax

```
#include <sql.h>

SQLTAPI  sqldox (shandle, dirname, attribute)

SQLTSVH  shandle;   /* Server Handle                */
SQLTDAP  dirname;   /* Directory name to open       */
SQLTFLG  attribute; /* file attribute to use on read */
```

Description

This function opens the file directory specified by dirname on the database server associated with shandle.

After you open a directory, you use *sqldrr* to read the file names in the directory. Only those file names that match the file attribute (defined in *sql.h*) will be returned.

Use the *sqldrc* function to close the directory.

The *sqldro* function does not return a handle for the directory because a program can only have one directory opened at a time. If you perform *sqldro* when a directory is already open, the current open directory is automatically closed.

Note: SQLBase supports filenames up to 256 characters including the terminating null character.

Parameters

shandle

The server handle returned by *sqlcsv.*

dirname

A pointer to a null-terminated string that contains the name of the directory to open.

attribute

File attribute flags which can be logically ORed to return combinations of files that match the attribute flag.

Flag	Description
SQLARDO	Read Only
SQLAHDN	Hidden Files
SQLASYS	system Files
SQLAVOL	Volume Label
SQLADIR	Subdirectories

Return value

If this function returns zero, it was successful. If this function returns a non-zero value, it was unsuccessful.

Example

```
SQLTSVH   shandle;
SQLTDAP   srvname;
SQLTFLG   fattribute;
char      *password;
char      *dirname;
int       modulo;
char      buffer[3000];

srvname = "SERVER1";
password = 0;
dirname = "\\CENTURA";
fattribute = SQLADIR;
/* CONNECT TO SERVER */

if (rcd = sqlcsv(&shandle, srvname, password))
     apierr("SQLCSV");
else
     printf("Connection Established to Server \n");

/* directory open, read and close */
printf("Directory open, read and close \n");

printf("\nOpen a directory of %s\n", dirname);
```

```
if ((rcd = sqldox(shandle, dirname, fattribute)) != 0)
     apierr("SQLDRO");
else
{
     printf("Directory opened successfully, rcd=%d\n", rcd);
     module = 0;
      while ((rcd = sqldrr(shandle, buffer)) == 0)
     {
            if ((modulo++ % 3) == 0)
               printf("\n");
         printf("%-13s", buffer);
     }
          printf("\n");
          printf("sqldrr() = %u\n", rcd);
          if (rcd = sqldrc(shandle))
            apierr("SQLDRC");
          else
            printf("Directory closed successfully, rcd=
              %d\n", rcd);
}

printf("End of directory open, read, and close\n");

/* DISCONNECT FROM SERVER */

if (rcd = sqldsv(shandle))
  apierr("SQLDSV");
else
  printf("Disconnected from Server \n");
```

Related functions

sqlcsv *sqldrc* *sqldrr*

sqldrc - DiRectory Close

Syntax

```
#include <sql.h>

SQLTAPI   sqldrc(shandle)

SQLTSVH   shandle;  /* Server handle */
```

Description

This function closes the directory on the database server associated with *shandle* that the program opened with the *sqldro* function.

Call this function after *sqldrr* has read the last file name in the directory.

A program can only have one directory opened at a time. If you perform a *sqldro* function when a directory is already open, the current open directory is automatically closed.

Parameters

shandle

The server handle returned by *sqlcsv.*

Return value

If this function returns zero, it was successful. If this function returns a non-zero value, it was unsuccessful.

Example

```
SQLTSVH   shandle;
SQLTDAP   srvname;
char      *password;
char      *dirname;
int       modulo;
char      buffer[3000];

srvname = "SERVER1";
password = 0;
dirname = "\\CENTURA";
```

```
/* CONNECT TO THE SERVER */

if (rcd = sqlcsv(&shandle,srvname,password))
   apierr("SQLCSV");
else
   printf("Connection Established to Server \n");

/* directory open, read and close */

printf("Directory open, read, close \n");

printf("\nOpen a directory of %s\n", dirname);
if ((rcd = sqldro(shandle, dirname)) != 0)
   apierr("SQLDRO");
else
     {
       printf("Directory opened successfully,
            rcd=%d\n",rcd); modulo = 0;
       while ((rcd = sqldrr(shandle, buffer)) == 0)
   {
       if ((modulo++ % 3) == 0)
       printf("\n");
       printf("%-13s", buffer);
   }
       printf("\n");
       printf("sqldrr() = %u\n", rcd);

       if (rcd = sqldrc(shandle))
          apierr("SQLDRC");
       else
       printf("Directory closed successfully, rcd=
            %d\n",rcd);

   }
printf("End of directory open, read, and close\n");

/* DISCONNECT FROM THE SERVER */

if (rcd = sqldsv(shandle))
   apierr("SQLDSV");
else
   printf("Disconnected from Server \n");
```

Related functions

sqlcsv *sqldro* *sqldrr*

sqldro - DiRectory Open

Syntax

```
#include <sql.h>

SQLTAPI   sqldro (shandle, dirname)

SQLTSVH   shandle;    /* Server handle */
SQLTDAP   dirname;    /* Directory name to open */
```

Description

This function opens the file directory specified by *dirname* on the database server associated with *shandle*.

After you open a directory, you use *sqldrr* to read the file names in the directory.

Use the *sqldrc* function to close the directory.

The *sqldro* function does not return a handle for the directory because a program can only have one directory opened at a time. If you perform *sqldro* when a directory is already open, the current open directory is automatically closed.

Note: SQLBase supports filenames up to 256 characters including the terminating null character.

Parameters

shandle

The server handle returned by *sqlcsv*.

dirname

A pointer to a null-terminated string that contains the name of the directory to open.

Return value

If this function returns zero, it was successful. If this function returns a non-zero value, it was unsuccessful.

Example

```
SQLTSVH   shandle;
SQLTDAP   srvname;
char      *password;
char      *dirname;
int       modulo;
char      buffer[3000];

srvname = "SERVER1";
password = 0;
dirname = "\\CENTURA";

/* CONNECT TO THE SERVER */

if (rcd = sqlcsv(&shandle,srvname,password))
   apierr("SQLCSV");
else
   printf("Connection Established to Server \n");

/* directory open, read and close */
printf("Directory open, read, close \n");

printf("\nOpen a directory of %s\n", dirname);
if ((rcd = sqldro(shandle, dirname)) != 0)
   apierr("SQLDRO");
else
   {
      printf("Directory opened successfully, rcd=%d\n",rcd);
      modulo = 0;
      while ((rcd = sqldrr(shandle, buffer)) == 0)
         {
            if ((modulo++ % 3) == 0)
               printf("\n");
               printf("%-13s", buffer);
            }
      printf("\n");
      printf("sqldrr() = %u\n", rcd);
      if (rcd = sqldrc(shandle))
         apierr("SQLDRC");
      else
```

```
            printf("Directory closed successfully, rcd=
              %d\n",rcd);

    }
printf("End of directory open, read, and close\n");

/* DISCONNECT FROM THE SERVER */

if (rcd = sqldsv(shandle))
  apierr("SQLDSV");
else
  printf("Disconnected from Server \n");
```

Related functions

sqlcsv *sqldrc* *sqldrr*

sqldrr - DiRectory Read

Syntax

```
#include <sql.h>

SQLTAPI   sqldrr (shandle, filename)

SQLTSVH   shandle;    /* Server handle */
SQLTDAP   filename;   /* File name buffer */
```

Description

This function reads a file name in the directory on the database server into the variable specified by *filename*.

This function is called after a *sqldro* function.

The *sqldrr* function returns one file name per call. The file name returned is only the base name for the file; the name does not include the directory name prefix.

Parameters

shandle

The server handle returned by *sqlcsv*.

filename

A pointer to the variable where this function returns the file name. The file name is null-terminated.

Return value

If this function returns zero, it was successful. If this function returns a non-zero value, it was unsuccessful.

This function returns an error code after the last file name has been read to indicate that the end of the directory has been reached.

Example

```
SQLTSVH   shandle;
SQLTDAP   srvname;
char      *password;
char      *dirname;
int       modulo;
char      buffer[3000];

srvname = "SERVER1";
password = 0;
dirname = "\\CENTURA";

  /* CONNECT TO THE SERVER */

if (rcd = sqlcsv(&shandle,srvname,password))
   apierr("SQLCSV");
else
   printf("Connection Established to Server \n");

/* directory open, read and close */
printf("Directory open, read, close \n");

printf("\nOpen a directory of %s\n", dirname);
if ((rcd = sqldro(shandle, dirname)) != 0)
   apierr("SQLDRO");
else
   {
       printf("Directory opened successfully,
          rcd=%d\n",rcd);
       modulo = 0;
         while ((rcd = sqldrr(shandle, buffer)) == 0)
           {
```

```
                    if ((modulo++ % 3) == 0)
                        printf("\n");
                    printf("%-13s", buffer);
                }
            printf("\n");
            printf("sqldrr() = %u\n", rcd);

            if (rcd = sqldrc(shandle))
                apierr("SQLDRC");
            else
                printf("Directory closed successfully,
                     rcd= %d\n",rcd);

        }
    printf("End of directory open, read, and close\n");

    /* DISCONNECT FROM THE SERVER */

    if (rcd = sqldsv(shandle))
        apierr("SQLDSV");
    else
        printf("Disconnected from Server \n");
```

Related functions

sqlcsv *sqldrc* *sqldro*

sqldrs - Drop Result Set

Syntax

```
#include <sql.h>

SQLTAPI   sqldrs (cur,rsp,rsl)

SQLTCUR   cur;     /* Cursor handle */
SQLTDAP   rsp;     /* Result set name buffer */
SQLTDAL   rsl;     /* Result set name length */
```

Description

This function drops a saved result set. The result set must have been created by calling *sqlcrs* and specifying a name.

Parameters

cur

The cursor handle associated with this function.

rsp

A pointer to the string that contains the name of the result set.

rsl

The length of the string pointed to by *rsp*. If the string pointed to by *rsp* is null-terminated, specify zero and the system will compute the length.

Return value

If this function returns zero, it was successful. If this function returns a non-zero value, it was unsuccessful.

Example

```
/* Drop result set "saveres" */
ret = sqldrs(cur, "saveres", 0);
```

Related functions

sqlcrs	*sqlrrs*	*sqlstr*
sqldrs	*sqlspr*	*sqlurs*
sqlprs	*sqlsrs*	

sqldsc - DeSCribe item in a SELECT command

Syntax

```
#include <sql.h>

SQLTAPI   sqldsc (cur, slc, edt, edl, chp, chlp, prep, scap)

SQLTCUR     cur;        /* Cursor handle */
SQLTSLC     slc;        /* Select column number */
SQLTDDT     PTR  edt;   /* External data type */
SQLTDDL     PTR  edl;   /* External data length */
SQLTCHP     chp;        /* Column heading buffer */
SQLTPTR     PTR  chlp;  /* Column heading length */
```

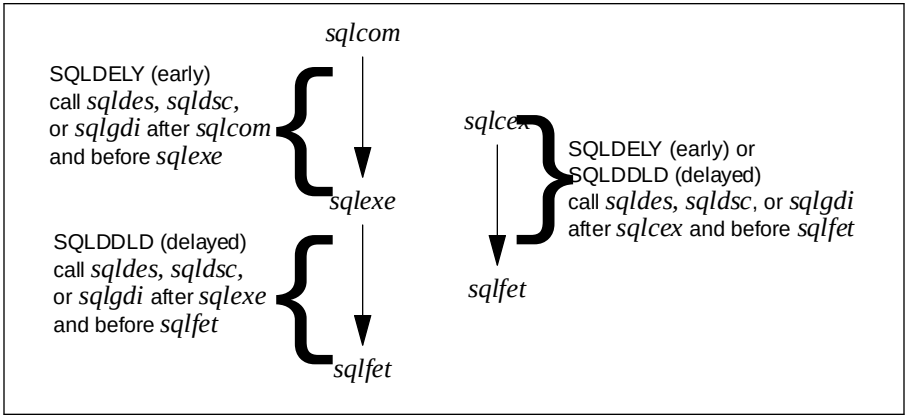
```
SQLTPRE     PTR  prep;  /* Numeric precision */
SQLTSCA     PTR  scap;  /* Numeric scale */
```

Description

This function returns *external* data type and length for a column in a SELECT list.

The external data type is defined in the SYSCOLUMNS system catalog table. External data types match program data types in *sql.h*. This function differs from *sqldes* which returns the *database* data type and length.

The following diagram shows how the value of the SQLPDIS parameter (SQLDELY, SQLDDLD, or SQLDNVR) controls when (and if) describe information for a SELECT statement is available for sending to a client. You can specify the SQLPDIS parameter's value using the *sqlset* function.



When describe information is available, given the different SQLPDIS parameter settings

This table summarizes the information illustrated above:

SQLPDIS constant	Value	When describe information is available
SQLDELY early (default)	0	The server sends describe information after *sqlcom*; subsequent calls to *sqldes*, *sqldsc*, or *sqlgdi* are legal until after a call to *sqlexe*. The server also sends describe information after *sqlcex*; subsequent calls to *sqldes*, *sqldsc*, or *sqlgdi* are legal until after a call to *sqlfet*.

SQLPDIS constant	Value	When describe information is available
SQLDDLD delayed	1	The server sends describe information after *sqlexe*. Calling *sqldes*, *sqldsc*, or *sqlgdi* after calling *sqlexe* but before the first *sqlfet* is legal; calling *sqldes*, *sqldsc*, or *sqlgdi* at any other time is illegal. The server also sends describe information after *sqlcex*; subsequent calls to *sqldes*, *sqldsc*, or *sqlgdi* are legal until after a call to *sqlfet*. Use this setting to reduce message traffic for database servers that do not support compile *(sqlcom)* operations (like Microsoft's SQL Server).
SQLDNVR never	2	The server *never* sends describe information; any call to *sqldes*, *sqldsc*, or *sqlgdi* is illegal. When SQLPDIS is set to SQLDNVR, *sqlnsi* always returns 0. You must hard code the number of SELECT items so that the application knows how many times to call *sqlssb.* Use this setting to reduce message traffic when the application always knows the number and type of columns in a SELECT statement and never makes calls to *sqldes*, *sqldsc*, or *sqlgdi*.

Specify null pointers (SQLNPTR) for arguments that you do not want.

You can retrieve the number of columns in the SELECT lists with the *sqlnsi* function.

Parameters

cur

The cursor handle associated with this function.

slc

The column number (starting with 1) in the SELECT list to get information about. You can use the column number to set up a loop and call *sqldsc* for each column in the SELECT list.

edt

A pointer to the variable where this function returns the external data type of the column.

Number	Typdef in sql.h	Data type
1	SQLEINT	INTEGER
2	SQLESMA	SMALLINT
3	SQLEFLO	FLOAT
4	SQLECHR	CHAR
5	SQLEVAR	VARCHAR
6	SQLELON	LONGVAR
7	SQLEDEC	DECIMAL
8	SQLEDAT	DATE
9	SQLETIM	TIME
10	SQLETMS	TIMESTAMP
11	SQLEMON	MONEY
12	SQLEDOU	DOUBLE
13	SQLEGPH	GRAPHIC
14	SQLEVGP	VARGRAPHIC
15	SQLELGP	LONG VARGRAPHIC
16	SQLEBIN	BINARY
17	SQLEVBI	VAR BINARY
18	SQLELBI	LONG BINARY
19	SQLEBOO	BOOLEAN
20	SQLELCH	CHAR >254
21	SQLELVR	VARCHAR >254

edl

A pointer to the variable where this function returns the external data length of the column:

Data type	Returns
INTEGER	4
SMALLINT	2
FLOAT	4 or 8
CHAR	Size specified when column was defined.
VARCHAR	Size specified when column was defined.
LONGVAR	0
DECIMAL	8
DATE	4
TIME	3
TIMESTAMP	10

Note that the length returned for numeric and datetime columns are as stored in SQLBase's internal format. Use the *sqldes* function to get the length for printing and display.

chp

A pointer to the variable where this function returns the column heading.

chlp

A pointer to the variable where this function returns the column heading length.

prep

A pointer to the variable where this function returns the precision of a numeric column.

scap

A pointer to the variable where this function returns the scale, if any, of a of a numeric column.

Return value

If this function returns zero, it was successful. If this function returns a non-zero value, it was unsuccessful.

Example

```
main()
{
    SQLTNSI     nsi; /* number of select items
    SQLTSLC     i;   /* column number to describe
    SQLTDDT     edt; /* external data type
    SQLTDDL     edl; /* external data length
    char        buf[19];/* buffer for column name
    SQLTPTR     chl; /* column header length
    SQLTPRE     prec;/* precision
    SQLTSCA     scale;/* scale

static  char  dbnam[] = "demox";
static  char  selcom[] = "SELECT * FROM TEST";

...

/* COMPILE THE SELECT COMMAND */

if (rcd = sqlcom(cur, selcom, 0))
    apierr("SQLCOM");

if (rcd = sqlnsi(cur,&nsi))
    apierr("SQLNSI");
/* DESCRIBE */

for (i = 1; i <= nsi; i++)
    {
      memset(buf, '\0', sizeof(buf)); /* fill the buffer with
                        nulls */
      if (rcd = sqldsc(cur,i,&edt,&edl,buf,&chl,&prec,&scale))
          apierr("SQLDSC");
      printf("i=%d, edt=%d, edl=%d, colname=%s, chl=%d, prec=%d,
          scale=%d\n", i,edt,edl,buf,chl,prec,scale);
    }

if (rcd = sqldis(cur))
    apierr("SQLDIS");

}
```

Related functions

sqldes *sqlgdi* *sqlnsi*

sqldst - Drop STored command/procedure

Syntax

```
#include <sql.h>

short     sqldst (cur, cnp, cnl);

SQLTCUR   cur;     /* Cursor handle */
SQLTDAP   cnp;     /* Command/procedure name buffer */
SQLTDAL   cnl;     /* Command/procedure name length */
```

Description

This function drops a stored command or stored procedure.

Parameters

cur

The cursor handle associated with this function.

cnp

A pointer to a string that contains the name of the SQL command or procedure to drop.

cnl

The length of the string pointed to by *cnp*. If the string pointed to by *cnp* is null-terminated, specify zero and the system will compute the length.

Return value

If this function returns zero, it was successful. If this function returns a non-zero value, it was unsuccessful.

Example

```
ret = sqldst(cur, "myquery", 0);
```

Related functions

sqlsto

sqldsv - Disconnect from SerVer

Syntax

```
#include <sql.h>

SQLTAPI   sqldsv (handle)

SQLTSVH   handle;   /* Server handle */
```

Description

This function disconnects from a server.

After the server connection is broken, you will not be able to perform administrative functions.

Parameters

shandle

The server handle returned by *sqlcsv.*

Return value

If this function returns zero, it was successful. If this function returns a non-zero value, it was unsuccessful.

Example

```
main()
{
   srvname = "SERVER1";
   password = 0;

/* CONNECT TO THE SERVER */

if (rcd = sqlcsv(&handle,srvname,password))
   apierr("SQLCSV");
else
```

```
    printf("Connection Established to Server \n");

/* DISCONNECT FROM THE SERVER */

if (rcd = sqldsv(handle))
    apierr("SQLDSV");
else
    printf("Disconnected from Server \n");
}
```

Related functions

sqlcsv

sqlelo - End Long Operation

Syntax

```
#include <sql.h>

SQLTAPI   sqlelo (cur)

SQLTCUR   cur;    /* Cursor handle */
```

Description

This function ends a LONG VARCHAR operation. This function removes the overhead necessary for handling LONG VARCHAR columns.

Parameters

cur

The cursor handle associated with this function.

Return value

If this function returns zero, it was successful. If this function returns a non-zero value, it was unsuccessful.

Example

```
ret=sqlelo(cur);
```

Related functions

sqlbld	*sqlgls*	*sqlrlo*
sqlbln	*sqllsk*	*sqlwlo*

sqlenr - ENd Rollforward

Syntax

```
#include <sql.h>

SQLTAPI   sqlenr (shandle, dbname, dbnamel)

SQLTSVH   shandle;    /* Server handle */
SQLTDAP   dbname;     /* Database name */
SQLTDAL   dbnamel;    /* Length of database name */
```

Description

Call this function after a rollforward operation has stopped because it cannot open the next transaction log file. If the next log file is not available, call this function to finish the rollforward recovery based on the logs processed up to that point.

Parameters

shandle

The server handle returned by *sqlcsv*.

dbname

A pointer to the string that contains the database name.

dbnamel

The length of the string pointed to by *dbname*. If the string pointed to by *dbname* is null-terminated, specify zero and the system will compute the length.

Return value

If this function returns zero, it was successful. If this function returns a non-zero value, it was unsuccessful.

Example

```
SQLTSVH     shandle;
char*       password;
SQLTDPV     lbmset;
SQLTFNP     bkpdir;
SQLTFNL     bkpdirl;
SQLTRFM     mode=SQLMEOL;
SQLTLNG     lognum;
SQLTBOO     local,over;

static  char  dbname1[] = "omed";

password = 0;
bkpdir = "\\BACKUP\\OMED";
bkpdirl = strlen(bkpdir);
printf("value of bkpdir = %s \n",bkpdir);

local=1;
over=1;

/* CONNECT TO SERVER */

if (rcd = sqlcsv(&shandle,srvname,password))
   apierr("SQLCSV");

/* RESTORE DATABASE */

if (rcd =
sqlrdb(shandle,dbname1,0,bkpdir,bkpdirl,local,over))
   apierr("SQLRDB");
else
   printf("Restored Database \n");

/* ROLLFORWARD TO END */

sqlrof(shandle,dbname1,0,mode,0,0);

lognum=0;

/* The loop below assumes that all log file backups */
/* are on disk.*/
/* If a log file backup is not on disk, lognum is set */
/*to a */
/* non-zero value which causes the loop to terminate. */

while (lognum == 0)
```

```
    {
        /* GET NEXT LOG */
        sqlgnl(shandle,dbname1,0,&lognum);

        /* RESTORE LOG FILES */
        sqlrlf(shandle,dbname1,0,bkpdir,bkpdirl,local,over);
    }
/* END ROLLFORWARD */

if (rcd = sqlenr(shandle,dbname1,0))
    apierr("SQLENR");
else
    printf("End Rollforward \n");
```

Related functions

sqlbdb	*sqlcsv*	*sqlrlf*
sqlblf	*sqlgnl*	*sqlrof*
sqlbss	*sqlrdb*	*sqlrss*
sqlcrf	*sqlrel*	

sqlepo - Error POsition

Syntax

```
#include <sql.h>

SQLTAPI  sqlepo (cur, epo)

SQLTCUR          cur;     /* Cursor handle */
SQLTEPO   PTR    epo;     /* Error position */
```

Description

This function returns the error position in the SQL command now being processed by the specified cursor. The error position is set after *sqlcom* or *sqlcex*.

When a SQL/API function returns an error, the offset of the error in the SQL command is set. The error position is meaningful after a compile or an execute because it points to the position in a SQL command where a syntax error was detected.

Parameters

cur

The cursor handle associated with this function.

epo

A pointer to a variable where this function returns the error position offset. The first character in the SQL command is position zero.

Return value

If this function returns zero, it was successful. If this function returns a non-zero value, it was unsuccessful.

Example

```
SQLTEPO errpos;  /* error position */
short ret;       /* return code */

if (!sqlcom(cur, sqlcmd, 0))
   ret = sqlepo(cur, &errpos);
```

Related functions

sqlcom *sqlcex*

sqlerr - ERRor message

Syntax

```
#include <sql.h>

SQLTAPI     sqlerr (error, msg)

SQLTRCD     error;    /* Error code */
SQLTDAP     msg;      /* Message text */
```

Description

This function returns the text of the error message associated with the error code. The text comes from the file *error.sql*.

Each SQL/API function returns a code. You can retrieve the most recent code with the function *sqlrcd* function.

The file *error.sql* contains message text for every return code. Each entry in *error.sql* contains the error code, mnemonic, message text, and the message reason and remedy for that code.

When a program detects an error condition, it uses the error code to look up the error message. Use the *sqlerr* function to retrieve the error message text (without the mnemonic) associated with a return code. Use the *sqlfer* function to retrieve the error message text and the mnemonic associated with a return code.

Parameters

error

The error code to retrieve the message text for.

msg

A pointer to the variable where this function returns the error message text. The error message text is a null-terminated string. SQLMERR is a constant in *sql.h* that indicates the size of the error message text buffer. This function *always* returns error message text.

Return value

This function returns zero if the value specified in *error* exists in *error.sql*. If this function returns a non-zero value, it means that the value in *error* does not exist in *error.sql*. The text returned in *msg* will also indicate this.

Example

```
char   errmsg [SQLMERR];   /* buffer for error msg */
short  ret;                /* return code */

if (ret = sqlexe(cur))
{
   sqlerr(ret, errmsg);    /* get error message */
   printf("%s \n", errmsg);/* print error message */
}
```

Related functions

sqletx *sqlrcd* *sqlxer* *sqlfer*

sqletx - Error message TeXt

Syntax

```
#include <sql.h>

SQLTAPI   sqletx (rcd, msgtyp, bfp, bfl, txtlen)

SQLTRCD           rcd;    /* Error code to get text for */
SQLTPTY           msgtyp; /* Message text type */
SQLTDAP           bfp;    /* Ptr to receiving buffer */
SQLTDAL           bfl;    /* Length of receiving buffer */
SQLTDAL     PTR   txtlen; /* Length of retrieved text */
```

Description

This function retrieves one or more of the following from the *error.sql* file for the specified error code:

- Error message
- Error reason
- Error remedy

Each API function call returns a code. You can retrieve the most recent error code with the *sqlrcd* function. When an application program detects an error condition, it can use the error code to look up the error message, error reason, and error remedy.

Parameters

rcd

The error code to retrieve information for.

msgtyp

You can specify the following message types individually or together by adding the constants together. For example, a value of seven indicates that you want the error message text, reason, and remedy all returned in the buffer that *bfp* points to.

Constant name	Value	Explanation
SQLXMSG	1	Retrieve error message text. The *sqlerr* function does the same thing.

Constant name	Value	Explanation
SQLXREA	2	Retrieve error message reason.
SQLXREM	4	Retrieve error message remedy.

bfp

A pointer to the buffer where this function copies the error message text, reason, or remedy.

bfl

Length of the buffer pointed at by *bfp*.

If you are retrieving the error message text, reason, and remedy, you can specify the *sql.h* constant SQLMETX for this argument. SQLMETX is always set to a value that is large enough to hold the error message text, reason, and remedy.

If you are only retrieving the error message text, you can specify the *sql.h* constant SQLMERR for this argument. SQLMERR is always set to a value that is large enough to hold the error message text.

txtlen

A pointer to the variable where this function returns the number of bytes retrieved.

For example, if the buffer is 100 bytes and requested text is 500 bytes, this function returns 100 bytes in *bfp* and a value of 500 in *txtlen*. The application program could then allocate a larger buffer to retrieve the entire text string.

Specify a null pointer if you do not want the total length of the text.

Return value

If this function returns zero, it was successful. If this function returns a non-zero value, it was unsuccessful.

Example

This example retrieves the error message text, reason, and remedy after calling *sqlcom*.

```
SQLTCUR   cur;        /* cursor value*/
SQLTRCD   rcd;        /* error code to get text for */
char      buf[1000];  /* buffer to receive the text */
SQLTDAL   txtlen;     /* length of returned text */
```

```
if (rcd = sqlcom(cur, "CREATE TABLE EMP (LASTNAME
          CHAR(20))", 0))
{
   sqletx(rcd, SQLXMSG + SQLXREA + SQLXREM, buf,
          sizeof(buf), &txtlen)
   printf("Error Explanation:\n%s\n", buf);
}
```

If you only wanted the remedy text, you would call the *sqletx* function as follows:

```
sqletx(rcd, SQLXREM, buf, sizeof(buf), &txtlen)
```

Related functions

sqlerr *sqlrcd* *sqlxer*
sqlfer

sqlexe - EXEcute a SQL command/procedure

Syntax

```
#include <sql.h>

SQLTAPI   sqlexe (cur)

SQLTCUR   cur;    /* Cursor handle */
```

Description

This function executes a previously-compiled command or procedure.

The command or procedure executed can be one compiled earlier in the current application or one that was stored and retrieved.

If the command or procedure contains bind variables, data must be bound before execution.

Parameters

cur

The cursor handle associated with this function.

To execute the following SQL commands, use the server handle returned by *sqlcsv* instead:

ALTER DATABASE
ALTER DBAREA
ALTER STOGROUP
CREATE DATABASE
CREATE DBAREA
CREATE STOGROUP
DEINSTALL DATABASE
DROP DATABASE
DROP DBAREA
DROP STOGROUP
INSTALL DATABASE
SET DEFAULT STOGROUP

Return value

If this function returns zero, it was successful. If this function returns a non-zero value, it was unsuccessful.

Example

```
if (ret = sqlexe(cur))
   {
      ... process error
   }
```

Related functions

sqlcex *sqlcom* *sqlcsv*

sqlexp - EXecution Plan

Syntax

```
# include <sql.h>

SQLTAPI   sqlexp (cur, buffer, length)

SQLTCUR   cur;        /* Cursor handle */
SQLTDAP   buffer;     /* Execution plan buffer */
SQLTDAL   length;     /* Length of buffer */
```

Description

This function returns the execution plan for a compiled SQL command. The execution plan shows the tables, views, indexes, and optimizations for the SQL command. Each line in the plan represents one table or view needed to process the SQL command.

Table and views for the SQL command are listed in the order in which they will be processed.

The SELECT column contains a number that identifies all the tables or views for a given SELECT.

The TABLE column contains the name of the table or view. System generated temporary tables are identified in the TABLE column as TEMP TABLE. For views and temporary tables, the table identifier is followed by the number of the SELECT which will be processed to produce the rows for the table or view.

The INDEX column contains the name of the index to use for the table. TEMP INDEX indicates a system-generated temporary index.

OPTIONS shows the processing options which have been selected.

ANTI JOIN	An optimization of the NOT IN operator.
INDEX MERGE	Optimize joining of tables where appropriate indexes are available in each table.
OR LIST	OR LIST optimization which occurs with an OR operator or an IN operator with a list of values.
OUTJOIN	Outer join has been specified.
QUICK TERM	IN optimization. When doing a join for purposes of satisfying an IN with a subselect, "quickly terminate" on the first satisfaction of the IN condition.

Parameters

cur

The cursor handle associated with this function.

buffer

A pointer to the variable where this function returns the execution plan for the command. Each line of the execution plan is terminated with a linefeed character. The end of the execution plan is terminated with a null.

length

The length of the value pointed to by *buffer*.

Return value

If this function returns zero, it was successful. If this function returns a non-zero value, it was unsuccessful.

Example

First, set up an area to receive the execution plan:

```
char buf [2000]
```

Then, compile a SQL command such as the one shown below:

```
SELECT DISTINCT S#, P#, QTY FROM SPJ
WHERE QTY =
  (SELECT MAX(QTY) FROM SPJ SPJY, S
  WHERE SPJY.P# = SPJ.P#
  AND SPJY.S# = S.S#
  AND S.CITY = 'ATHENS')
```

Call the *sqlexp* function:

```
ret = sqlexp(cur, buf, sizeof (buf));
```

The area *buf* will contain an execution plan as shown below.

```
EXECUTION PLAN:

SELECT    TABLE              INDEX         OPTIONS
========= ================== ============ =========
1         SPJ
1         TEMP TABLE-SEL     TEMP INDEX
2         S
2         SPJ                SPJX          INDEXMERGE
```

sqlfer - Full ERror message

Syntax

```
#include <sql.h>

SQLTAPI   sqlfer (error, msg)

SQLTRCD   error;  /* Error code */
SQLTDAP   msg;    /* Message buffer */
```

Description

This function returns the full text of the error message associated with the error code specified by *error*. The text that this function returns comes from *error.sql*.

Each SQL/API function returns a code. You can retrieve the most recent code with the function *sqlrcd* function.

The file *error.sql* contains message text for every return code. Each entry in *error.sql* contains the error code, the mnemonic, and the message text for that code.

When a program detects an error condition, it uses the error code to look up the error message. Use the *sqlerr* function to retrieve the error message text (without the mnemonic) associated to a return code. Use the *sqlfer* function to retrieve the error message text and the mnemonic associated to a return code

Parameters

error

The error code to retrieve the message text for.

msg

A pointer to the variable where this function returns the full error message text. The error message text is a null-terminated string. SQLMERR is a constant in *sql.h* that indicates the size of the error message text. This function *always* returns error message text.

Return value

This function returns zero if the value in *error* exists in *error.sql*. If this function returns a non-zero value, it means that the value in *error* does not exist in *error.sql*. The text returned in *msg* will also indicate this.

Example

```
#include "sql.h"
#include "stdio.h"

#define ERR_NUMS 12

main()
{
   SQLTRCDerror;       /* error code */
   introw_num;
   charmsg_buf[200];

staticintmsg_line[ERR_NUMS] =
   {
      1, 4, 2104, 9001, 9100, 9286, 9287, 9288, 9289, 9301,
           171, 3001
   };

for (row_num=0;row_num<ERR_NUMS;row_num++)
   {
      sqlfer(msg_line[row_num],msg_buf);
      printf("Output from SQLFER():  %s\n",msg_buf);
   }

}
```

Related functions

sqlerr *sqlrcd* *sqlxer*
sqletx

sqlfet - FETch next row from result set

Syntax

```
#include <sql.h>

SQLTAPI   sqlfet (cur)

SQLTCUR   cur;     /* Cursor handle */
```

Description

This function fetches the next row resulting from a query. A successful *sqlexe* or *sqlcex* must come before this function. This function returns an end of fetch value (1) when there are no more rows to fetch.

This function is associated with fetchable commands. In SQLBase, a fetchable command is one that can return a result through *sqlfet*. The SELECT and PROCEDURE commands are fetchable commands. This means that you can fetch results from a SELECT or PROCEDURE command until you reach the end of output.

Retrieve LONG VARCHAR columns with the *sqlrlo* function.

If there is an error, the return code will not indicate the column that caused the problem. Check the *pfc* variable (set up with *sqlssb*) or use *sqlgfi* to determine the column in error.

Parameters

cur

The cursor handle associated with this function.

Return value

This function returns the values shown in the table below during normal operation. Any other value returned means that an error occurred.

Returned Value	Meaning
0	Row was fetched.
1	End of fetch (last row has been fetched).
2	Update performed since last fetch.

Returned Value	Meaning
3	Delete performed since last fetch.

Example

```
ret = sqlfet(cur);
```

Related functions

sqlcex *sqlgfi* *sqlssb*
sqlexe

sqlfgt - GeT File from server

Syntax

```
#include <sql.h>

SQLTAPI   sqlfgt(shandle, srvfile, lclfile)

SQLTSVH   shandle;    /* Server handle */
SQLTDAP   srvfile;    /* Server filename */
SQLTDAP   lclfile;    /* Local file name */
```

Description

This function copies the file specified by *srvfile* on the database server associated to *shandle* to the file *lclfile* on the client computer.

Note: SQLBase supports filenames up to 256 characters including the terminating null character.

Parameters

shandle

The server handle returned by *sqlcsv*.

srvfile

A pointer to the null-terminated string that contains the name of the file on the database server to copy.

lclfile

A pointer to the null-terminated string that contains the name of the file on the client computer where the server file is copied.

Return value

If this function returns zero, it was successful. If this function returns a non-zero value, it was unsuccessful.

Example

```
main()
{
   SQLTSVH     shandle;
   SQLTDAP     srvname;
   char        *password;
   char        *srvfile;
   char        *lclfile;

   srvname = "SERVER1";
   password = 0;
   srvfile = "sql.h";
   lclfile = "localsql.h";

/* CONNECT TO THE SERVER */

if (rcd = sqlcsv(&shandle,srvname,password))
   apierr("SQLCSV");
else
   printf("Connection Established to Server \n");

if (rcd = sqlfgt(shandle, srvfile, lclfile))
   apierr("SQLFGT");
else
   printf("Successful Get File from Server \n");

srvfile = "srvsqlfl.h";

if (rcd = sqlfpt(shandle, srvfile, lclfile))
   apierr("SQLFPT");
else
```

```
    printf("Successful Put File to Server \n");

/* DISCONNECT FROM THE SERVER */

if (rcd = sqldsv(shandle))
    apierr("SQLDSV");
else
    printf("Disconnected from Server \n");
}
```

Related functions

sqlcsv *sqlfpt*

sqlfpt - PuT File to server

Syntax

```
#include <sql.h>

SQLTAPI   sqlfpt (shandle, srvfile, lclfile)

SQLTSVH   shandle;    /* Server handle */
SQLTDAP   srvfile;    /* Server file name */
SQLTDAP   lclfile;    /* Local file name */
```

Description

This function copies the file specified by *lclfile* on the client computer to the file *srvfile* on the database server associated to *shandle*.

Note: SQLBase supports filenames up to 256 characters including the terminating null character.

Parameters

shandle

The server handle returned by *sqlcsv*.

srvfile

A pointer to the null-terminated string that contains the name of the file on the database server where the client file is copied.

lclfile

A pointer to the null-terminated string that contains the name of the file on the client computer to copy.

Return value

If this function returns zero, it was successful. If this function returns a non-zero value, it was unsuccessful.

Example

```
main()
{
   SQLTSVH   shandle;
   SQLTDAP   srvname;
   char      *password;
   char      *srvfile;
   char      *lclfile;

   srvname = "SERVER1";
   password = 0;
   srvfile = "sql.h";
   lclfile = "localsql.h";

/* CONNECT TO THE SERVER */

if (rcd = sqlcsv(&shandle,srvname,password))
   apierr("SQLCSV");
else
   printf("Connection Established to Server \n");

if (rcd = sqlfgt(shandle, srvfile, lclfile))
   apierr("SQLFGT");
else
   printf("Successful Get File from Server \n");

srvfile = "srvsqlfl.h";

if (rcd = sqlfpt(shandle, srvfile, lclfile))
   apierr("SQLFPT");
else
   printf("Successful Put File to Server \n");

/* DISCONNECT FROM THE SERVER */
```

```
if (rcd = sqldsv(shandle))
   apierr("SQLDSV");
else
   printf("Disconnected from Server \n");
}
```

Related functions

sqlcsv *sqlfgt*

sqlfqn - Fully-Qualified column Name

Syntax

```
#include <sql.h>

SQLTAPI  sqlfqn (cur, col, nameptr, namelen)

SQLTCUR   cur;              /* Cursor handle */
SQLTFLD   field;            /* Field number */
SQLTDAP   nameptr;          /* Column name */
SQLTDAL   PTR     namelen;  /* Length of column name */
```

Description

This function returns the fully-qualified name of a column in a SELECT list. The function can be called only after a SELECT command has been compiled or retrieved because this is the only time the information is available.

An attempt to get a SELECT list element that is not a database column name causes an error. This can happen when a SELECT list item is an expression, a view column name derived from an expression, or a constant.

This function is faster than a query on the SYSCOLUMNS system catalog table.

This function differs from *sqldes* and *sqldsc* because it returns the fully-qualified name of the underlying table of a column in a SELECT list. The *sqldes* and *sqldsc* functions only return the column heading.

Parameters

cur

The cursor handle associated with this function.

field

The column number that indicates the sequence number (starting with 1) of the item in the SELECT list for which the fully-qualified name is wanted.

nameptr

A pointer to the variable where this function returns the name. The fully-qualified name of a column has this form:

```
username.columnname.tablename
```

namelen

A pointer to the variable where this function returns the length of the name.

Return value

If this function returns zero, it was successful. If this function returns a non-zero value, it was unsuccessful.

Example

```
#define NOTCOL 5131 /* select list element not a
                    column name error */

static char select[] = "select name, phone, from empview; ";
char colname[50];
uint cvl;
uint col;
short ret;/* return code */

/* get fully qualified name */

memset(colname, ' ', sizeof(colname));/*initialize */
for (col=1, col <= 2, col++)
{
   if (ret = sqlfqn(cur, col, colname, &cvl ))
{
      if (ret == NOTCOL)
            continue;/* not a real column */
         else
            ... process error
   }
   ProcessName (colname, cvl);
}
```

Related functions

sqldsc *sqldes*

sqlgbc - Get Backend Cursor

Syntax

```
#include <sql.h>

SQLTAPI   sqlgbc (cursor, curp)

SQLTCUR          cursor; /* Cursor Handle */
SQLTCUR   PTR    curp;   /* Cursor Handle */
```

Description

This function retrieves the backend cursor handle for the supplied cursor handle.

Parameters

cursor

A cursor handle returned by *sqlcnc*.

curp

A pointer to the variable where this function returns the backend cursor handle.

Return value

If this function returns zero, it was successful. If this function returns a non-zero value, it was unsuccessful.

Example

```
SQLTCUR   cur;    /* Cursor handle */
SQLTCUR   curp;   /* Cursor handle */
SQLTRCD   rcd;    /* Return code */


if (rcd = sqlcnc( &cur, "PAYROLL/BOSS/SECRET", 0))
{
    printf("Failure on connect (rcd = %d \n", rcd);
    exit(0);
```

```
}
else
{
     if ((rcd = sqlgbc( cur, &curp)) != 0)
     {
     apierr("SQLGBC");
     }
     else
     {
        printf("Backend Cursor: %d \n", curp);
     }
}
```

sqlgbi - Get Backend Information

Syntax

```
#include <sql.h>

  SQLTAPI  sqlgbi (cursor, curp, pnmp)

  SQLTCUR      cursor; /* Cursor Handle                */
  SQLTCUR PTR  curp;   /* Backend cursor handle ptr    */
  SQLTPNM PTR  pnmp;   /* Backend process number ptr   */
```

Description

This function retrieves the backend cursor handle and process number for the supplied cursor handle.

Parameters

cursor

A cursor handle returned by sqlcnc.

curp

A pointer to the variable where this function returns the backend cursor handle.

pnmp

A pointer to the variable where this function returns the backend process number.

Return value

If this function returns zero, it was successful. If this function returns a non-zero value, it was unsuccessful.

Example

```
SQLTCUR   cur;      /* Cursor handle   */
SQLTCUR   curp;     /* Backend cursor handle   */
SQLTPNM   pnmp;     /* Backend process number  */
SQLTRCD   rcd;      /* Return code   */


if (rcd = sqlcnc( &cur, "PAYROLL/BOSS/SECRET", 0))
{
   printf("Failure on connect (rcd = %d \n", rcd);
   exit(0);
}
else
{
   if ((rcd = sqlgbi( cur, &curp, &pnmp)) != 0)
   {
   apierr("SQLGBC");
    }
  else
  {
     printf("Backend Cursor: %d  Backend Process: %d \n",
curp, pnmp);
  }
}
```

sqlgdi - Get Describe Information

Syntax

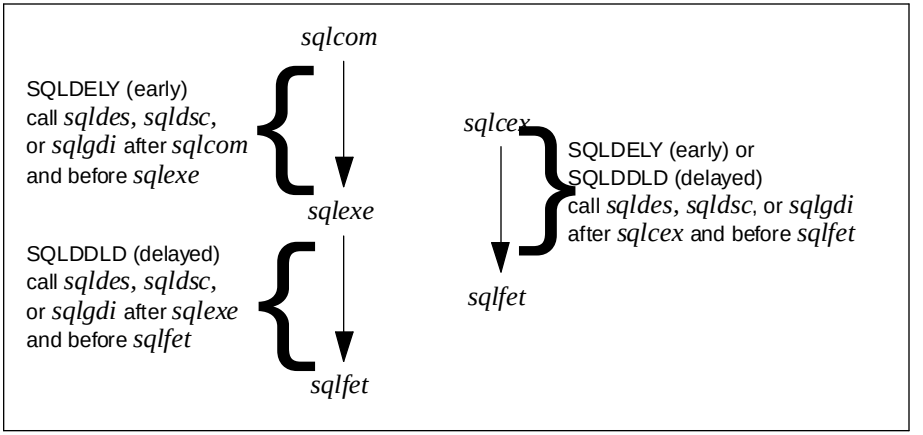
```
#include <sql.h>

SQLTAPI   sqlgdi (cur, gdidef);

SQLTCUR   cur;      /* Cursor handle */
SQLTPGD   gdidef;   /* Describe structure */
```

Description

This function returns descriptive information about a column in a SELECT list.

This function returns all the descriptive information that *sqldes* and *sqldsc* return as well as the column label and the null indicator.

The diagram below shows how the SQLPDIS settings (SQLDELY, SQLDDLD, and SQLDNVR) control when describe information is available. You can specify the SQLPDIS parameter's value by calling the *sqlset* function.



The following table explains how the setting of the SQLPDIS parameter controls when you can call *sqlgdi*. The SQLPDIS parameter controls when (and if) describe information for a SELECT statement is sent to a client.

SQLPDIS setting (constant)	Value	When you can call sqlgdi
SQLDELY early (default)	0	The server sends describe information after *sqlcom*; subsequent calls to *sqldes*, *sqldsc*, or *sqlgdi* are legal until after a call to *sqlexe.* The server also sends describe information after *sqlcex*; subsequent calls to *sqldes*, *sqldsc*, or *sqlgdi* are legal until after a call to *sqlfet.*

SQLDDLD delayed	1	The server sends describe information after *sqlexe*. Calling *sqldes*, *sqldsc*, or *sqlgdi* after calling *sqlexe* but before the first *sqlfet* is legal; calling *sqldes*, *sqldsc*, or *sqlgdi* at any other time is illegal. The server also sends describe information after *sqlcex*; subsequent calls to *sqldes*, *sqldsc*, or *sqlgdi* are legal until after a call to sqlfet. Use this setting to reduce message traffic for database servers that do not support compile *(sqlcom)* operations (like Microsoft's SQL Server).
SQLDNVR never	2	The server *never* sends describe information; any call to *sqldes*, *sqldsc*, or *sqlgdi* is illegal. When SQLPDIS is set to SQLDNVR, *sqlnsi* always returns 0. You must hard code the number of SELECT items so that the application knows how many times to call *sqlssb*. Use this setting to reduce message traffic when the application always knows the number and type of columns in a SELECT statement and never makes calls to *sqldes*, *sqldsc*, or *sqlgdi*.

You can retrieve the number of columns in the SELECT list with the *sqlnsi* function and then use the number of columns in a loop that calls *sqlgdi* for each column.

Parameters

cur

The cursor handle associated with this function.

gdidef

This is a structure that you define in the program where this function returns information about a column. The structure and typedefs below are defined in *sql.h*:

```
struct    gdidefx
  {
  ubyte1      gdichb[31];   /* Column heading */
  SQLTCHL     gdichl;       /* Column heading length */
  ubyte1      gdilbb[31];   /* Label */
  SQLTLBL     gdilbl;       /* Label length */
  SQLTSLC     gdicol;       /* SELECT column number */
  SQLTDDT     gdiddt;       /* Database data type */
  SQLTDDL     gdiddl;       /* Database data length */
  byte2       gdiedt;       /* External data type */
```

```
SQLTDDT     gdiedl;      /* External data length */
SQLTPRE     gdipre;      /* Decimal precision */
SQLTSCA     gdisca;      /* Decimal scale */
byte2       gdinul;      /* Null indicator */
byte1       gdifil[50];  /* Reserved */
};
typedef     struct       gdidefx gdidef;
typedef     struct     gdidefx SQLTGDI;
typedef     struct     gdidefx* SQLTPGD;
#define     GDISIZ     sizeof(gdidef)
```

The table below explains the elements in the structure. You only need to fill-in *gdicol* before calling *sqlgdi.*

Element	Explanation
gdichb	The column heading (name) defined in the SYSCOLUMNS system catalog table.
gdichl	The length of the column heading.
gdilbb	The label defined in the SYSCOLUMNS system catalog table.
gdilbl	The length of the column label.
gdicol	The column number (starting with 1) in the SELECT list.
gdiddt	A pointer to the variable where this function returns the database data type of the column: **Typedef in *sql.h*** — **Number** — **Data type** SQLDCHR — 1 — Character SQLDNUM — 2 — Numeric SQLDDAT — 3 — Date-time SQLDLON — 4 — Long SQLDDTE — 5 — Date (only) SQLDTIM — 6 — Time (only)

<table>
<tr><th>Element</th><th>Explanation</th></tr>
<tr><td>gdiddl</td><td>The database length of the column:

<u>Data type</u> <u>Length</u>

Character Size specified when column was defined.
Numeric 27 (22 digits of precision plus room for scientific notation).
Date-time 26
Long 0
Date (only) 10
Time (only) 15</td></tr>
<tr><td>gdiedt</td><td>The external data type of the column:

<u>Typdef in sql.h</u> <u>Number</u> <u>Data type</u>
SQLEINT 1 INTEGER
SQLESMA 2 SMALLINT
SQLEFLO 3 FLOAT
SQLECHR 4 CHAR
SQLEVAR 5 VARCHAR
SQLELON 6 LONGVAR
SQLEDEC 7 DECIMAL
SQLEDAT 8 DATE
SQLETIM 9 TIME
SQLETMS 10 TIMESTAMP
SQLEMON 11 MONEY
SQLEDOU 12 DOUBLE
SQLEGPH 13 GRAPHIC
SQLEVGP 14 VARGRAPHIC
SQLELGP 15 LONG VARGRAPHIC
SQLEBIN 16 BINARY
SQLEVBI 17 VAR BINARY
SQLELBI 18 LONG BINARY
SQLEBOO 19 BOOLEAN</td></tr>
</table>

Element	Explanation
gdiedl	The external length of the column: **Data type** — **Length** INTEGER — 4 SMALLINT — 2 FLOAT — 4 or 8 CHAR — Size specified when column was defined. VARCHAR — Size specified when column was defined. LONGVAR — 0 DECIMAL — 8 DATE — 4 TIME — 3 TIMESTAMP — 10
gdipre	Precision for a numeric column.
gdisca	Scale, if any, for a numeric column.
gdinul	Null indicator: -1 — Column can contain null value. 0 — Column cannot contain null value.
gdifil	Reserved.

Return value

If this function returns zero, it was successful. If this function returns a non-zero value, it was unsuccessful.

Example

```
sqlgdi(cur, gdidef);
```

Related functions

sqldes *sqllab* *sqlnsi*
sqldsc

sqlget - GET parameter

Syntax

```
#include <sql.h>

SQLTAPI  sqlget (cur/shandle, parm, pbuf, len)

SQLTCUR        cur/shandle;/* Database cursor or server
                              handle */
SQLTPTY        parm;       /* Parameter type */
SQLTDAP        pbuf;       /* Information buffer */
SQLTDAL  PTR   len;        /* Information length */
```

Description

This function retrieves individual database parameters. Pass a parameter type and retrieve a corresponding value and length.

Note: SQLBase supports filenames up to 256 characters including the terminating null character.

Parameter	Description
SQLPAID	**Adapter Identifier.** This parameter allows the setting of an network adapter identification string. If you call *sqlset* and specify the SQLPAID parameter, it changes the setting of the *adapter_id* keyword in *win.ini*.
SQLPALG	**Process Activity file name.** The file to which SQLBase writes the messages displayed on a multi-user servers Process Activity screen.
SQLPANL	**Apply net log.** This parameter disables internal condition checking while a netlog is being applied. This keyword is useful to Centura technical support and development personnel only. If you call *sqlset* and specify the SQLPANL parameter, it changes the setting of the *applynetlog* keyword in *sql.ini*. 0 = Off 1 = On

Parameter	Description
SQLPAPT	**Activate process timing.** When this parameter is on (1), activation times are accumulated for prepares, executes and fetches. Activation times are accumulated at three different levels; system, process, and cursor. By default, this parameter is turned off. 0 = Off 1 = On Note that if you are using the *sqlset* function to set the SQLPCTL (command time limit) parameter, parameter settings for the SQLPAPT (activate process timing) and SQLPSTA (statistics for server) parameters can be affected in the following ways: • When you enable a command time limit (by specifying a non-zero value in either the *cmdtimeout* keyword of the server's *sql.ini* file or with the SQLPCTL parameter), SQLPSTA (statistics for server) and SQLPAPT (process timing) are automatically turned on. • If you turn off a command time limit, SQLPSTA (statistics for server) and SQLPAPT (process timing) are automatically turned off, unless you explicitly turned on either parameter after you enabled a command time limit. • If you explicitly turn off either SQLPSTA (statistics for server) or SQLPAPT (process timing), your command time limit (if you enabled on) is turned off and *sql.ini* is updated to reflect *cmdtimeout*=0. It is recommended that if you set a value for any of these three parameters, you should set the same value for the other two. For example, if you set SQLPAPT parameter On (1), you should also set SQLPCTL and SQLPSTA parameters On (1).
SQLPAUT	**Autocommit.** Commits the database automatically after each SQL command. By default this parameter is Off (0) and SQLBase commits the database only when you issue a COMMIT command. Autocommit is cursor-specific. When you set autocommit On (1) for a cursor and then perform an operation with that cursor, SQLBase commits *all* of the transaction's cursors. Performing operations with cursors that do not have autocommit set on does not affect the rest of the transaction's cursors. You cannot have autocommit and bulk execute on simultaneously.
SQLPAWS	**OS averaging window size.** This parameter specifies the number of samples of the CPU % Utilization value to keep for determining the average value. You can specify a window size of 1 to 255. The default setting is one (1). If you call *sqlset* and specify the SQLPAWS parameter, it changes the setting of the *osavgwindow* keyword in *sql.ini*. 0 = Off 1 = 255 units

Parameter	Description
SQLPBLK	**Bulk execute mode**. Reduces the network traffic for multi-row inserts, deletes, and updates. In bulk execute mode, data values are buffered so that many rows can be sent to the server in one message. Increasing the size of the output message buffer (with the *sqloms* function) increases the number of operations that can be buffered in one message to the server, thereby improving performance. This setting is cursor specific. If this is On (1), as many operations are buffered in the output message buffer as possible. By default, bulk execute mode is Off (0). Bulk execute mode cannot be on at the same time as the autocommit (SQLPAUT) option.
SQLPBRN	**Database brand**. SQLBALB - HP Allbase SQLBAPP - SQLHost Application Services SQLBAS4 - IBM AS/400 SQL/400 SQLBDB2 - IBM DB2 SQLBDBC - Teradata DBC Machines SQLBIGW - Informix SQLBIOL - Informix On-Line SQLBNTW - NetWare SQL SQLBORA - Oracle SQLBSQB - Centura SQLBase SQLBSHR - Teradata ShareBase
SQLPBRS	**Backend result sets**. If the database server supports backend result sets, this parameter's value is 1 (Yes); otherwise, its value is 0 (No).
SQLPCAC	**Size of database cache (in KBytes)**. This parameter sets the cache which buffers database pages in memory. The larger the cache, the less the disk input and output. In other words, as you increase the value of the cache setting, disk access is reduced. The default cache size for Windows is 500K; for all other platforms, the default is 2M. The minimum is 15K and the maximum is 32767K. If you call *sqlset* and specify the SQLPCAC parameter, it changes the setting of the *cache* keyword in *sql.ini*, but the new setting does not take effect until SQLBase is restarted.

Parameter	Description
SQLPCCB	**Connect Closure Behavior.** This parameter specifies the connect closure behavior that occurs when you terminate a connection using the *sqldch* function. Valid options are COMMIT, ROLLBACK, or DEFAULT. The default is 0 which means that connect closure behavior is dependent on the database server to which the user is connected. In the case of SQLBase, the DEFAULT setting (0) issues a COMMIT before a connection handle is terminated. To determine the DEFAULT behavior for other servers, read the applicable server documentation. Setting this parameter on (1) instructs the server to issue a COMMIT before a connection handle is terminated, while a setting of (2) issues a ROLLBACK. This option also specifies whether a COMMIT or ROLLBACK is issued before disconnecting to a cursor with an implicit connection using the *sqlcnc* function.
SQLPCCK	**Client check.** This parameter tells SQLBase to send the client a RECEIVE upon receipt of a request. By default, clientcheck is off (0). When SQLBase has finished executing a command, it issues a SEND request to the client with the results of the command. If successful, the server then issues a RECEIVE request and waits to receive another command. Setting this parameter on (1) instructs SQLBase to issue a RECEIVE request before beginning execution of the command, not after it finishes executing the command. Doing so allows SQLBase to detect a situation where the client session is dropped or a cancel request is made during command processing. If you call *sqlset* and specify the SQLPCCK parameter, it changes the setting of the *clientcheck* keyword in *sql.ini*. 0 = Off 1 = On
SQLPCGR	**Contiguous cache pages in cache group.** This parameter specifies the number of contiguous cache pages to allocate. For example if you set cache at 3000, and cachegroup at 30, SQLBase allocates 100 cache groups, consisting of 30 pages each. To set the number of cache pages per group to 50: cachegroup = 50 The default is 30. If you call *sqlset* and specify the SQLPCGR parameter, it changes the setting of the *cachegroup* keyword in *sql.ini*.
SQLPCHS	**Retrieved chained command contains a SELECT command**. 0 = Chained command does *not* contain a SELECT command. 1 = Chained command *does* contain a SELECT command. This setting is cursor-specific.

Parameter	Description
SQLPCIS	**Client identifier.** This parameter returns a client identification string. The client identification string will consist of: MAIL_ID\NETWORK_ID\ADAPTER_ID\APP_ID\CLIENT_NAME Each of these identification strings can be returned separately by calling sqlget with the appropriate parameter.
SQLPCLG	**Commit logging.** When this parameter is On (1), SQLBase causes every database transaction in which data was modifed to log a row of data. The data that is logged contains the transaction's identifier (Transaction ID) and a unique sequence number. When the COMMIT operation is executed for a transaction that is modified, the data is logged in the system utility table SYSCOMMITORDER. The SYSCOMMITORDER table lists transactions that operated on the database in the order in which they were committed. For details on the SYSCOMMITORDER table, see "Appendix C," in the *Database Administrator's Guide*. Turning the SQLPCLG parameter Off (0), which is the default, stops commit logging. Turning the SQLPCLG parameter Off (0), which is the default, stops commit logging. You must have DBA privileges to set the SQLPCLG parameter and to use DDL commands with the SQLPCLG parameter. Note that commit logging is also supported for replication with Centura Ranger.
SQLPCLI	**LOAD/UNLOAD Client Value.** The load/unload's ON CLIENT clause value. 0 = Off (file is on the server) 1 = On (file is on the server) This parameter indicates where the load/unload file will reside. Before using this parameter, compile the load/unload statement first.
SQLPCLN	**Client name.** The name of a client computer.
SQLPCMP	**Message compression**. When message compression is On (1), messages sent between a client and the database server or gateway are compressed. This means that messages are shorter, and more rows can be packed into a single message during bulk insert and fetch operations. The compression algorithm collapses repeating characters (run-length encoding). SQLBase performs the compression incrementally as each component of a message is posted. By default, message compression is Off (0) because it incurs a CPU cost on both the client and server machines. This parameter is cursor-specific.
SQLPCSV	**Commit server status.** Indicates whether commit service is enabled for the server. 0 = Off 1 = On

Parameter	Description
SQLPCTF	**LOAD/UNLOAD control file indicator.** Indicates whether a file is a load/unload control file. 0 = Not a control file 1 = Is control file You can use this parameter in conjunction with the SQLPCTF parameter (control filename) to obtain information about a file after you compile the load/unload statement.
SQLPCTI	**Checkpoint time interval**. How often SQLBase should perform a recovery checkpoint operation. SQLBase's automatic crash recovery mechanism requires that recovery checkpoints be done. The default checkpoint time interval is one minute. This should yield a crash recovery time of less than a minute. If your site can tolerate a longer crash recovery time, you can increase this interval to up to 30 minutes. Depending on the applications running against the database server, a checkpoint operation can affect performance. If this happens, you can increase the checkpoint interval until you attain the desired performance.
SQLPCTL	**Command time limit.** The amount of time (in seconds) to wait for a SELECT, INSERT, UPDATE, or DELETE statement to complete execution. After the specified time has elapsed, SQLBase rolls back the command. Valid values range from 1 to 43,200 seconds (12 hours maximum), and include 0 (zero) which indicates an infinite time limit. Note that if you are using the *sqlset* function to set the SQLPCTL (command time limit) parameter, settings for the SQLPAPT (activate process timing) and SQLPSTA (statistics for server) parameters can be affected in the following ways: • When you enable a command time limit (by specifying a non-zero value in either the *cmdtimeout* keyword of the server's *sql.ini* file or with the SQLPCTL parameter), SQLPSTA (statistics for server) and SQLPAPT (process timing) are automatically turned on. • If you turn off a command time limit, SQLPSTA (statistics for server) and SQLPAPT (process timing) are automatically turned off, unless you explicitly turned on either parameter after you enabled a command time limit. • If you explicitly turn off either SQLPSTA (statistics for server) or SQLPAPT (process timing), your command time limit (if you enabled on) is turned off and *sql.ini* is updated to reflect *cmdtimeout*=0. It is recommended that if you set a value for any of these three parameters, you should set the same value for the other two. For example, if you set SQLPCTL parameter On (1), you should also set SQLPAPT and SQLPSTA parameters On (1).

Parameter	Description
SQLPCTS	**Character set file name.** This parameter identifies a file that specifies different values for the ASCII character set. This is useful for non-English speaking countries where characters in the ASCII character set have different hex values than those same characters in the U.S. ASCII character set. If you call *sqlset* and specify the SQLPCTS parameter, it changes the setting of the *characterset* keyword in *sql.ini*.
SQLPCTY	**Country file section (for example, France).** This parameter tells SQLBase to use the settings in the specified section of the *country.sql* file. SQLBase supports English as the standard language, but it also supports many national languages including those spoken in Europe and Asia. You specify information that enables SQLBase to support another language in the *country.sql* file. If you call *sqlset* and specify the SQLPCTY parameter, it changes the setting of the *country* keyword in *sql.ini*.
SQLPCXP	**Execution plan cost.** SQLBase uses a cost-based optimizer to determine the most efficient way to access data based on the available indexes, system catalog statistics, and the composition of a SQL command. The access plan SQLBase chooses is the one with the lowest estimated cost.
SQLPDBD	**DBDIR keyword information**. The drive, path, and database directory name information specified for the *sql.ini's* DBDIR keyword.
SQLPDBM	**Database mode**. Indicates whether the database is local or remote. SQLMDBL = local SQLMRTR = remote
SQLPDBN	**Database name**. The name of the database that you are accessing.
SQLPDDB	**Default database name**. This overrides the SQLBase default database name of DEMO.
SQLPDDR	**Database directory**. The drive, path, and directory name where the database you are connected to resides.
SQLPDIS	**Describe information control**. When (and if) SQLBase sends describe information for a SELECT command to a client. This parameter is cursor-specific. **SQLDELY** (0) means early and is the default value. The server sends describe information after a call to *sqlcom*. Call *sqldes*, *sqldsc*, or *sqlgdi* after *sqlcom* and before *sqlexe*. The server also sends describe information after a call to *sqlcex*. Call *sqldes*, *sqldsc*, or *sqlgdi* after *sqlcex* and before *sqlfet*. **SQLDDLD** (1) means delayed. The server sends describe information after a call to *sqlexe*. Call *sqldes*, *sqldsc*, or *sqlgdi* after *sqlexe*, but before the first *sqlfet*. Calling *sqldes*, *sqldsc*, or *sqlgdi* at any other time is illegal. The server also sends describe information after *sqlcex*. Call *sqldes*, *sqldsc*, or *sqlgdi* after *sqlcex* and before *sqlfet*. Use this setting to reduce message traffic for database servers that do not support compile (*sqlcom*) operations.

Parameter	Description
	SQLDNVR (2) means never. The server never sends describe information. Any call to *sqldes*, *sqldsc*, or *sqlgdi* is illegal. When you set SQLPDIS to SQLDNVR, *sqlnsi* always returns zero (0). You must hard-code the number of columns in the SELECT command so that the application knows how many times to call *sqlssb*. Use this setting to reduce message traffic when the application always knows the number and type of columns in a SELECT command and never makes calls to *sqldes*, *sqldsc*, or *sqlgdi*.
SQLPDLK	**Deadlocks.** The number of deadlocks that have occurred since the server was started.
SQLPDMO	Demo version of database. 0 = No 1 = Yes
SQLPDPW	**Default password**.
SQLPDTL	**Database command time limit.** This parameter sets the amount of time (in seconds) to wait for a SELECT, INSERT, UPDATE or DELETE command to complete execution. This only includes the time to prepare and execute, not the time to fetch. After the specified time has elapsed, SQLBase rolls back the command. The time limit is valid only for the database requested. A global server command time limit is available by using SQLPCTL. 0 = no time limit 1 = 43,000 secs
SQLPDTR	**Set distributed transaction mode.** If this parameter is on (1), all subsequent CONNECTs and SQL statements will be part of a distributed transaction. Currently, you can have one distributed transaction per session. The default for this parameter is off (0). 0 = Off 1 = On
SQLPDUS	Default username.
SQLPEMT	**Error message tokens.** One or more object names (tokens) returned in an error message.

Parameter	Description
SQLPERF	**Error filename.** Specifies a file that contains entries to translate standard SQLBase return codes into user-defined return codes: errorfile=filename The file contains entries for error code translation in the form: sbrcd,udrcd where *sbrcd* is a SQLBase return code found in *error.sql*, and *udrcd* is a user-defined return code. The *sbrcd* value must be a positive integer; the *udrcd* can be a positive or negative integer. There can be no white space between the values or after the comma. The client application converts the *sbrcd* value to the *udrcd* value using the *sqltec* API function. For example, SQLBase returns a value of '1' to indicate an end-of-fetch condition, while DB2 returns a value of '100'. If you want an application to convert all SQLBase return codes of '1' to '100', the entry in the errorfile would look like this: 1,100 When your application calls the *sqltec* function, if the SQLBase return code doesn't exist, SQLBase returns a non-zero return code that means that the translation did not occur. To force translation to occur, you can create a global translation entry using the asterisk (*) character and a generic return code (like '999'). For example, assume an errorfile of SQLBase return codes and corresponding DB2 return codes. For those SQLBase return codes that have no corresponding DB2 return code, you can force the application to return the generic return code '999' with the following entry: *,999 If you call *sqlset* and specify the SQLPERF parameter, it changes the setting of the *errorfile* keyword in *sql.ini*.
SQLPEXE	**Database server program name**.
SQLPEXP	**Execution plan.** Retrieves the execution plan of the last SQL statement that SQLBase compiled.
SQLPEXS	**Extension size** (in MBytes for partitioned databases, and in KBytes for non-partitioned databases). SQLBase databases grow dynamically as data is added, and expand in units called *extensions*. When a database becomes full, SQLBase must add another extension (or *extent*) to the database.
SQLPFNM	**LOAD/UNLOAD filename**. The name of the load/unload file. This can also be the name of the load/unload control filename. The client application uses this parameter to obtain the filename after the load/unload statement is compiled at the back end. You can also use this in conjunction with the SQLPCTF (control file value parameter).

Parameter	Description
SQLPFRS	**Frontend result sets**. SQLBase supports backend result sets, but for those database servers that do not, Centura offers frontend result sets (maintained on the client computer). For SQLBase, SQLPFRS is Off (0). For database servers that don't support backend end result sets, like DB2, SQLPFRS is On (1). This parameter is cursor-specific.
SQLPFT	**Fetchthrough mode**. If fetchthrough is On (1), rows are fetched from the database server even if they are available from the client's input message buffer. Since data could have been updated since you last fetched it (into the input message buffer), using the fetchthrough feature ensures that you see the most up-to-date data. If fetchthrough is Off (0), rows are fetched from the client's input message buffer when possible. In fetchthrough mode, rows are fetched from the backend one at a time; there is no multi-row buffering. Because of this, and the network traffic involved, fetchthrough increases response time. Note for procedures, if you want the On Procedure Fetch section to execute exactly once for every fetch call from the client, returning one row at a time, set fetchthrough mode On at the client (the default is Off). If the result set you are fetching was created by a SELECT command that included an aggregate function, defined a complex view, or included a DISTINCT, GROUP BY, HAVING, UNION, or ORDER BY clause, then SQLBase creates a virtual table. The rows of this virtual table *cannot* be mapped to the rows in the database. For this reason, if a row in the result set is UPDATEd, when you fetch it, it will *not* reflect the UPDATE even if fetchthrough is On. This parameter is cursor-specific.
SQLPGBC	**Global cursor**. The COBOL SQLPrecompiler uses this parameter. Note that COBOL SQLPrecompiler is not released with the standard SQLBase 6.0. This parameter is listed here for the sake of completeness.
SQLPGCD	Group commit delay ticks.
SQLPGCM	Group commit count.
SQLPHEP	**Heap size of DOS TSR executables.** For a single-user database server, the heap is the space available for sorting, cursor workspace, and cache. For *dbrouter.exe*, the heap is the memory used by message buffers for communicating with the server. Note that the DOS platform is not released with the standard SQLBase 6.0. This parameter is listed here for the sake of completeness.
SQLPHFS	**Read-only history file size** (in KBytes). If read-only mode is enabled, this parameter limits the size of the read-only history file. The default size is 1 MByte (1000 KBytes).

Parameter	Description
SQLPISO	**Isolation level** of all the cursors that the program connects to the database. See the *sqlsil* function for an explanation of the isolation levels. SQLILRR = Repeatable Read SQLILCS = Cursor Stability SQLILRO = Read-Only SQLILRL = Release Locks
SQLPLBM	**Transaction log backup mode**. By default, this parameter is not enabled (0) and SQLBase deletes log files as soon as they are not needed to perform transaction rollback or crash recovery. This is done so that log files do not accumulate and fill up the disk. If SQLPLBM is Off (0), you are not able to recover the database if it is damaged by user error or a media failure. This parameter must be On (1) when you back up databases (*sqlbdb*) and log files (*sqlblf*), but does not need to be On when you back up snapshots (*sqlbss*).
SQLPLCK	**Lock limit allocations.** This parameter specifies the maximum number of lock entries to allocate. SQLBase allocates lock entries dynamically (in groups of 100) on an as-needed basis. The default setting is 0, which means that there is no limit on the number of locks allocated; as many lock entries can be allocated as memory permits. If you call *sqlset* and specify the SQLPLCK parameter, it changes the setting of the *locks* keyword in *sql.ini*.
SQLPLDR	**Transaction log directory**. The disk drive and directory that contains the log files. SQLBase creates log files in the home database directory by default, but you can redirect them to a different drive and directory with the *sql.ini*'s *lodgir* keyword.
SQLPLDV	**Load version**. Retrieves the load version you set when you called *sqlset* with this parameter. This parameter is cursor-specific.
SQLPLFF	**Support long data with front-end result sets**. Lets (1) you or prevents (0) you from reading and writing long data when using front end result sets with SQLNetwork routers and gateways. This parameter is cursor-specific.
SQLPLFS	**Transaction log file size** (in KBytes). The default log file size is 1 MByte (1000 KBytes) and the smallest size is 100,000 bytes.
SQLPLGF	Get log file offset. You can use this parameter to see how much of a log file has been written.

Parameter	Description
SQLPLOC	**Local/remote database server**. Specifies whether the database being accessed is local or remote. 0 = Remote 1 = Local engine
SQLPLSS	**Last SQL statement.** Retrieves the last SQL statement that SQLBase compiled.
SQLPLRD	**Local result set directory**. If the database server does not support backend result sets, this parameter retrieves the name of the directory on the client computer that contains the frontend result set file. By default, this is the current working directory.
SQLPMID	**E-Mail Identifier.** This parameter allows the setting of an E-Mail identification string. If you call *sqlset* and specify the SQLPMID parameter, it changes the setting of the *mail_id* keyword in *win.ini*.
SQLPMUL	**Multi-user version of SQLBase**. Specifies whether the database server you are accessing is multi-user (1) or single-user (0).
SQLPNCK	**Check network transmission errors.** This parameter enables and disables a checksum feature that detects transmission errors between the client and the server. To use this feature, both the client and the server must enable netcheck. The default is off (0). If you call *sqlset* and specify the SQLPNCK parameter, it changes the setting of the *netcheck* keyword *sql.ini*. 0 = Off 1 = On
SQLPNCT	**Netcheck algorithm.** This parameter specifies the algorithm SQLBase uses when netcheck is enabled. Configure this keyword only when you enable netcheck. By default, checksum(0) is enabled. To switch to CRC/16: netchecktype = 1 If you call *sqlset* and specify the SQLPNCT parameter, it changes the setting of the *netchecktype* statement in *sql.ini*. 0 = Checksum 1 = CRC/16
SQLPNDB	**Mark as brand new database.** Used in conjunction with COUNTRY.DBS. 0 = False 1 = True
SQLPNID	**Network identifier.** This parameter allows the setting of an Network identification string. If you call *sqlset* and specify the SQLPNID parameter, it changes the setting of the *network_id* keyword in *win.ini*.

Parameter	Description
SQLPNIE	**Null indicator error**. Controls what *sqlfet* returns in *sqlssb*'s *pfc* parameter when the value is null: 0 = *sqlfet* returns zero (default). 1 = *sqlfet* returns FETRNUL (7).
SQLPNLB	**Next transaction log file to back up.** An integer that specifies the number of the next log file to back up.
SQLPNLG	**Net log file.** This parameter invokes a diagnostic server utility that records database messages to a specified log file. This utility logs all messages that pass between a server and clients on a network. Do not use the netlog utility unless instructed to do by Centura's Technical Support staff. By default, the netlog utility is off. If you call *sqlset* and specify the SQLPNLG parameter, it changes the setting of the *netlog* keyword in *sql.ini*.
SQLPNPB	**Do not prebuild result sets**. If SQLPNPB is Off (0), SQLBase prebuilds result sets. The database server releases shared locks before returning control to the client. The client application must wait until the entire result set is built before it can fetch the first row. If SQLPNPB is On (1), SQLBase doesn't prebuild result sets if the client is in result set mode and Release Locks (RL) isolation level. The advantage of having SQLPNPB on is that the client does not have to wait very long before fetching the first row. SQLBase builds the result set as the client fetches data. By default, SQLPNPB is On (1) for single-user engines and Off (0) for multi-user servers. This parameter is cursor-specific.
SQLPNPF	**Net prefix character.** This parameter allows SQLBase to distinguish a database on one server from an identically-named database on another server and to circumvent the network's requirement of name uniqueness. You can specify a value with which SQLBase prefaces each database name on the server. If you have a netprefix entry in the server's *sql.ini* file, all clients connecting to databases on that server must specify the same netprefix value in their configuration files. If you call *sqlset* and specify the SQLPNPF parameter, it changes the setting of the *netprefix* keyword in *sql.ini*.

Parameter	Description
SQLPOBL	**Optimized bulk execute mode**. This is similar to, but even faster than, bulk execute mode (SQLPBLK) which reduces the network traffic for multi-row inserts, deletes, and updates. The difference is that if an error occurs, SQLBase rolls back the entire transaction. In bulk execute mode, data values are buffered so that many rows can be sent to the server in one message. Increasing the size of the output message buffer (with the *sqloms* function) increases the number of operations that can be buffered in one message to the server, thereby improving performance. This setting is cursor specific. If this is On (1), as many operations are buffered in the output message buffer as possible. By default, bulk execute mode is Off (0). Bulk execute mode cannot be on at the same time as the autocommit (SQLPAUT) option.
SQLPOFF	**Optimize first fetch.** This parameter lets you set the optimization mode for a particular cursor. All queries that are compiled or stored in this cursor inherit the optimization mode in effect. 0 = optimizes the time it takes to return the entire result set. 1 = optimize the time it takes to fetch the first row of the result set. If you call *sqlget* and specify the SQLPOFF parameter, it overrides the setting for *optimizefirstfetch* in *sql.ini* for the particular cursor. If you do not specify this parameter, the optimization mode for the cursor is determined by the setting of the *optimizefirstfetch* value of the server. If *sql.ini* does not have an *optimizefirstfetch* keyword, the default setting is 0 (optimize the time it takes to return the entire result set). Note that a parameter that was earlier stored, retrieved, and executed will continue to use the execution plan with which it was compiled.
SQLPOMB	**Output buffer message size.** This parameter sets the size (in bytes) of the output message buffer. The output message buffer is allocated on both the client computer and on the database server. The client builds an output message in this buffer and sends it to a buffer of the same size on the database server. It is called an *output* message buffer because it is output from the client's point of view. The most important messages sent from the client to the database server are SQL commands to compile or a row of data to insert. A larger output message buffer does *not* reduce network traffic unless bulk execute is on. SQLBase automatically maintains an output message buffer large enough to hold any SQL command or a row to insert of any length (given available memory). Despite the specified output message buffer size, SQLBase dynamically allocates more space for the output message buffer if needed. A large output message buffer can help performance when writing LONG VARCHAR columns.

Parameter	Description

SQLPOOJ	**Oracle outer join.** This parameter enables and disables Oracle-style join processing. Oracle's outer join implementation differs from the ANSI and industry standard implementation. To paraphrase the ANSI standard, the correct semantics of an outer join are to display all the rows of one table that meet the specified constraints on that table, regardless of the constraints on the other table. For example, assume two tables (A and B) with the following rows:
	`Table A (a int)` `Table B (b int)`
	`1` `1`
	`2` `2`
	`3` `3`
	`4`
	`5`
	If you issue the following SQL command:
	SELECT a, b
	FROM A, B
	WHERE A.a = B.b (+)
	AND B.b IS NULL;
	the ANSI result is:
	`Table A (a int)` `Table B (b int)`
	`1`
	`2`
	`3`
	`4`
	`5`
	Assuming the same two tables and the same SQL command, the correct result for Oracle is:
	`Table A (a int)` `Table B (b int)`
	`4`
	`5`
	If you set *oracleouterjoin*=1; you receive the Oracle result shown directly above. If you call *sqlset* and specify the SQLPOOJ parameter, it changes the setting of the *oracleouterjoin* keyword in *sql.ini*.
	0 = Off
	1 = On

SQLPORID	**Oracle row ID.** Retrieves the Oracle row ID affected by the most recent operation. Use this parameter in applications that access an Oracle database through SQLRouter/Oracle or SQLGateway/Oracle.

Parameter	Description
SQLPOPL	**Optimizer techniques**. Tells you which optimizing techniques that SQLBase is using for all clients that connect to a server. You can fall back on old optimizing techniques after upgrading to newer versions of SQLBase by using the *sqlset* function to set this value to 1. If you discover better performance of a query when this parameter is set to 1, you should report it to Centura's Technical Support team. Be sure not to include compilation time in the comparison of settings 1 and 2. 1 = SQLBase is using old optimizing techniques. 2 = SQLBase is using current optimizing techniques (default).
SQLPOSR	**OS statistics sample rate.** This parameter specifies the frequency at which operating system statistics (CPU % Utilization) are gathered. You can specify a setting of 0 to 255 seconds. The default setting is zero (0), which disables the gathering of CPU statistics. If you call *sqlset* and specify the SQLPOSR parameter, it changes the setting of the *ossamplerate* keyword in *sql.ini*. 0 = Off 1 = 255 secs
SQLPOVR	**LOAD/UNLOAD overwrite value.** Indicates whether the unload command contained an OVERWRITE clause. 0 = No OVERWRITE clause 1 = OVERWRITE clause specified
SQLPPAR	**Partitioned database.** Indicates the database is partitioned. 0 = No 1 = Yes
SQLPPCX	**Cursor context preservation**. If cursor context preservation is On (1), SQLBase prevents a COMMIT from destroying an active result set, thereby enabling an application to maintain its position after a COMMIT, INSERT, or UPDATE. Locks are kept on pages required to maintain the fetch position. Be aware that this can block other applications trying to access the same data. Also, locks can prevent other applications from doing DDL operations. By default, cursor context preservation is Off (0). A COMMIT destroys a cursor's result set or compiled command. SQLBase does *not* preserve cursor context after an isolation level change or a system-initiated ROLLBACK, such as a deadlock, timeout, etc. SQLBase *does* preserve cursor context after a user-initiated ROLLBACK if both of the following are true: 1) The application is in Release Locks (RL) isolation level. 2) A data definition language (DDL) statement was not performed.

Parameter	Description
	If the result set you are fetching was created by a SELECT command that included an aggregate function, defined a complex view, or included a DISTINCT, GROUP BY, HAVING, UNION, or ORDER BY clause, then SQLBase creates a virtual table. The rows of this virtual table *cannot* be mapped to the rows in the database. For this reason, if a row in the result set is UPDATEd, when you fetch it, it will *not* reflect the UPDATE even if fetchthrough is On. This parameter is cursor-specific.
SQLPPDB	**Access to partitioned databases**. While this parameter is TRUE, users can access partitioned databases; when FALSE (0), user access to partitioned databases is disabled, allowing you to restore MAIN.DBS.
SQLPPLF	**Preallocate transaction log files**. By default, this parameter is Off (0) and a log files grows in increments of 10% of its current size. This uses space conservatively, but can lead to a fragmented log file which can affect performance. If this parameter is On (1), log files are created full size (preallocated).
SQLPPLV	**Level of Process Activity display**. The level (0 - 4) of detail of the messages on a multi-user server's Process Activity display.
SQLPPTH	**Path separator on server machine.** This is useful for remote file operations.
SQLPREC	**Recovery**. If this parameter is On (1), SQLBase performs transaction logging. Transaction logging enables SQLBase to roll back changes made to a database before a COMMIT, and to recover from a system failure. If this parameter is Off (0), SQLBase does not perform transaction logging.
SQLPRES	**Restriction mode**. If this parameter is On (1), SQLBase uses the result of one query as the basis for the next query. Each subsequent query further restricts the result set. If this parameter is Off (0), each successive query overwrites the result set created by the previous query.
SQLPRID	**Retrieve current row ID**. This parameter retrieves a row's current ROWID. This is useful to see if a row's ROWID has changed as a result of an UPDATE command.
SQLPROD	**Read-only database**. Makes a database accessible on a read-only basis. SQLBase disallows you from executing data definition language (DDL) or data manipulation language (DML) commands. If this parameter is On (1), SQLBase disables both the Read-Only isolation level and transaction logging.

Parameter	Description
SQLPROM	**Read-only transaction mode**. Allows users connecting to any of the databases on the server to use the RO (read-only) isolation level. The RO isolation level allows users to have a consistent view of data during their session. If this parameter is On (1), SQLBase allows users to use the RO isolation level. All future server sessions for all databases on the server are started with RO transactions enabled; SQLBase maintains a read-only history file that contains multiple copies of modified database pages; when users try to access pages changed by other users, SQLBase retries a copy of the original page from the history file. Read-only transactions can affect performance, so, by default, this parameter is Off (0), prohibiting users from setting the RO isolation level. If you call *sqlset* and specify the SQLPROM parameter, it changes the setting of the *readonly* keyword in *sql.ini*, but the new setting does not take effect until you restart SQLBase. 0 = Off 1 = On NOTE: To turn on RO transaction mode for a single database and the current session, use the SQLPROT parameter.
SQLPROT	**Read-only transaction mode**. If this parameter is On (SQLVON), SQLBase allows applications to set the read-only (RO) isolation level on for a single database and the current server session. SQLBase maintains a read-only history file that contains one or more copies of pages that have been modified. Read-only transactions can affect performance, so, by default, this parameter is Off (SQLVOFF), prohibiting use of the RO isolation level. If this parameter is set to the default (SQLVDFL), SQLBase uses the *readonly* keyword setting in the *sql.ini* file to determine whether to allow read-only transactions. If you do not provide a value for this keyword, SQLBase uses the internal default (SQLVOFF). NOTE: To turn on RO transaction mode for a single database and the current server session, use the SQLPROM parameter.
SQLPRTO	**Rollback on lock timeout**. This parameter is On (1) by default and SQLBase rolls back an entire transaction when there is a lock timeout. If this parameter is Off (0), SQLBase rolls back only the current command. This parameter is cursor-specific.
SQLPSCR	**Scroll mode**. Otherwise known as result set mode, scroll mode lets you scroll back and forth through a result set and retrieve any row in the result set without sequentially fetching forward. Once you have positioned the cursor on a row, later fetches start from that row. Scroll mode is On if this parameter is 1, and Off if it is 0. This parameter is cursor-specific.

Parameter	Description
SQLPSIL	**Silent mode.** This parameter turns the display for multi-user server on (0) and off (1). To set the display of the server screens off: silent = 1 By default, multi-user server displays are on (0). If you call *sqlset* and specify the SQLPSIL parameter, it changes the setting of the *silent* statement in *sql.ini*. 0 = On 1 = Off
SQLPSTA	**Statistics for server.** This parameter collects the following timer and counter information: Timers: Compile. Execute. Fetch. Counters: Physical disk writes. Physical disk reads. Virtual disk writes Virtual disk reads. Total number of disconnects. Total number of connects. Hash joins - number of joins that have occurred. Sorts - number of sorts that have been performed Deadlocks - number of deadlocks that have occurred. Process switches - number of process switches. Full table scan - number of times a full table scan occurred. Index use - number of times an index has been used. Transactions - number of completed transactions. Command type executed - one counter for each command type. The default for this parameter is off (0). 0 = off 1 = on Note that if you are using the *sqlset* function to set the SQLPCTL (command time limit) parameter, settings for the SQLPAPT (activate process timing) and SQLPSTA (statistics for server) parameters can be affected in the following ways: • When you enable a command time limit (by specifying a non-zero value in either the *cmdtimeout* keyword of the server's *sql.ini* file or with the SQLPCTL parameter), SQLPSTA (statistics for server) and SQLPAPT (process timing) are automatically turned on.

Parameter	Description
	• If you turn off a command time limit, SQLPSTA (statistics for server) and SQLPAPT (process timing) are automatically turned off, unless you explicitly turned on either parameter after you enabled a command time limit. • If you explicitly turn off either SQLPSTA (statistics for server) or SQLPAPT (process timing), your command time limit (if you enabled on) is turned off and *sql.ini* is updated to reflect *cmdtimeout*=0. It is recommended that if you set a value for any of these three parameters, you should set the same value for the other two. For example, if you set SQLPSTA parameter On (1), you should also set SQLPCTL and SQLPSTA parameters On (1).
SQLPSVN	**Name of server.** This parameter shows the name of the server you are connected to. Setting this parameter will only change the setting in *sql.ini*. To activate the new setting, the server must be restarted. You must have DBA authority to set this parameter.
SQLPSWR	**Write defaults.** Changes to defaultdatabase, defaultuser, or defaultpassword are written to *sql.ini*. 0 = No 1 = Yes
SQLPTCO	**Time colon only.** This parameter configures SQLBase to recognize when a delimiter other than a colon(:) is being used to separate the hours, minutes, and seconds portions of a time value. The default is off (0). If you call *sqlset* and specify the SQLPTCO parameter, it changes the setting of the *timecolononly* keyword in *sql.ini*. 0 = No 1 = Yes
SQLPTHM	**Thread mode.** This parameter specifies whether to use native threads or SQLBase threads. A value of 1 indicates SQLBase threads and a value of 2 indicates native threads. Note for Windows 95, SQLBase now uses Windows 95 native threads only. By default, *threadmode* is 1, except on Windows 95 where the default is 2. On NetWare platforms, if you are running in Ring 0, Centura recommends using SQLBase threads which invoke stack switching. This should yield better performance. Novell disallows stack switching in Ring 3, so be sure to set threadmode to 2 when in Ring 3. If you call *sqlset* and specify the SQLPTHM parameter, it changes the setting of the *threadmode* keyword in *sql.ini*.
SQLPTMS	**Timestamp**. If this parameter is TRUE (1), SQLBase timestamps the messages on a multi-user server's Process Activity display; if FALSE (0), SQLBase does not.

Parameter	Description
SQLPTMO	**Client request time out.** This parameter specifies the time period that the server waits for a client to make a request. If the client does not make a request within the specified period, SQLBase rolls back the client session, processes, and transactions. The time-out clock restarts each time the client makes a request. The time-out value is 0 (infinite by default, and the maximum value is 200 minutes. If you call *sqlset* and specify the SQLPTMO parameter, it changes the setting of the *timeout* statement in *sql.ini*.
SQLPTMZ	**Time zone.** This parameter sets the value of SYSTIMEZONE, a SQLBase keyword that returns the time zone as an interval of Greenwich Mean Time. SYSTIMEZONE uses the expression (SYSTIME - TIMEZONE) to return the current time in Greenwich Mean Time. By default, timezone is 0. If you call *sqlset* and specify the SQLPTMZ parameter, it changes the setting of the *timezone* keyword in *sql.ini*.
SQLPTPD	**Temp directory.** This parameter specifies the directory where SQLBase places temporary files. In the course of processing, SQLBase can create several kinds of temporary files: sort files, read-only history files, and general-use files. To specify *d:\tmp* as the temporary directory: tempdir = d:\tmp You must set tempdir for read-only databases. If you call *sqlset* and specify the SQLPTPD parameter, it changes the setting of the *tempdir* keyword in *sql.ini*.
SQLPTRC	**Trace stored procedures.** Enables or disables statement tracing for procedures. 0 = Off 1 = On
SQLPTRF	**Tracefile name**. Directs statement output to a file on the server. If you do not set this parameter to a file name, the statement output goes to the server's Process Activity screen.
SQLPTSL	**Transaction span limit**. The number of log files that SQLBase allows an active transaction to span. When SQLBase creates a new log file, it checks this limit for all active transactions and rolls back any transaction that violates the limit. By default, the transaction span limit is zero (0) which disables the limit checking.

Parameter	Description
SQLPTSS	**Thread stack size.** This parameter specifies the stack size. By default, threadstacksize is 10 kilobytes and the minimum value is 8192 bytes. You should not decrease the default value. Running complex queries when threadstacksize is set to 8192 can result in a stack overflow error. If you receive stack overflow errors, increase the value of threadstacksize by 512 bytes at a time. If you call *sqlset* and specify the SQLPTSS parameter, it changes the setting of the *threadstacksize* keyword in *sql.ini*.
SQLPUID	**Application identifier.** This parameter allows the setting of an user identification string. If you call *sqlset* and specify the SQLPUID parameter, it changes the setting of the *app_id* keyword *win.ini*.
SQLPUSR	**Number of users.** This parameter specifies the maximum number of client applications that can connect to the server simultaneously. This means, for example, that a server configured with users=5 could support five clients running one application each, or one client running five applications, or two clients with one running two applications and the other running three applications, and so on. The default value of users is 128, and the maximum is 800. If you call *sqlset* and specify the SQLPUSR parameter, it changes the setting of the *users* keyword in *sql.ini*.
SQLPVER	**Release version**. The version number of the SQLBase server program.
SQLPWFC	**Which Fetchable Command**. The type of fetchable command: • **SQLTSEL (1)** is a SELECT command. • **SQLTPRO (87)** is a PROCEDURE command. • **0** is returned for all other commands (such as INSERT or UPDATE).
SQLPWKA	**Work space allocation unit.** This parameter specifies the basic allocation unit of a work space. For example, if a SQL command requires 5000 bytes and the default value of 1000 is in effect, SQLBase makes 5 memory allocation requests to the operating system (5 * 100 = 5000). The default is 1000 bytes. If you call *sqlset* and specify the SQLPWKA parameter, it changes the setting of the *workalloc* keyword in *sql.ini*.

Parameter	Description
SQLPWKL	**Maximum work space limit.** This parameter specifies a maximum memory limitation for SQL commands. For example, if you specify: worklimit = 4000 SQLBase cannot execute SQL commands requiring more than 4000 bytes of memory. The default is NULL, meaning that no memory limitation exists. If you call *sqlset* and specify the SQLPWKL parameter, it changes the setting of the *worklimit* statement in *sql.ini*.
SQLPWTO	**Lock wait timeout**. The number of seconds for SQLBase to wait for a database lock to be acquired. After the specified time has elapsed, SQLBase rolls back the command or transaction. The default is 300 seconds. Valid timeout values are: 1 - 1800 inclusive (1 second to 30 minutes) 0 = never wait; return error immediately 1 = wait forever This parameter is only relevant for multi-user servers and it is transaction-specific.

Parameters

cur

A cursor handle if the parameter is associated with a cursor or database. A server handle if the parameter is associated with a server. A value of 'No' in the table on the next page indicates that a cursor handle and a server handle is not needed to retrieve the information for the parameter.

parm

The name of the parameter to retrieve. The parameter types are defined in *sql.h* and are shown in the table that begins on the next page.

pbuf

A pointer to the variable where this function returns the parameter. The data type and size of the variable depends on the parameter. For strings like the database directory (SQLPDDR), the variable must be at least SQLMFNL bytes long. SQLMFNL is defined in *sql.h* under "maximum sizes".

len

Specify an address or a pointer to the length. After making this call, *len* is the number of bytes in the value pointed to by *pbuf*. The following table shows whether you need to specify a length for a parameter. If it is not necessary to designate a parameter length, specify zero (0).

Parameter Types

The following table lists:

- *parm* - the parameter type.
- *cur* - whether the parameter requires a cursor handle.
- *pbuf* - the size of the variable pointed to by *pbuf*.
- *len* - whether you need to specify a length for the parameter.

The parameter types and *pbuf* types and sizes are defined in *sql.h*.

parm	*cur*	*pbuf*	*len*
SQLPAID	Yes	SQLMFNL	Yes
SQLPAIO	No	SQLTDPV	No
SQLPALG	Yes	SQLMFNL	Yes
SQLPANL	No	SQLTDPV	No
SQLPAPT	Yes	SQLTDPV	No
SQLPAUT	Yes	SQLTDPV	No
SQLPAWS	Yes	SQLTDPV	No
SQLPBLK	Yes	SQLTDPV	No
SQLPBRN	Yes	SQLTDPV	No
SQLPBRS	Yes	SQLTDPV	No
SQLPCAC	Yes	SQLTDPV	No
SQLPCCK	No	SQLTDPV	No
SQLPCGR	No	SQLTDPV	No
SQLPCHS	Yes	SQLTDPV	No
SQLPCIS	Yes	SQLMFNL	Yes
SQLPCLG	Yes	SQLTDPV	Yes
SQLPCLI	Yes	SQLTDPV	No
SQLPCLN	Yes	SQLMFNL	Yes
SQLPCMP	Yes	SQLTDPV	No

parm	cur	pbuf	len
SQLPCSV	Yes	SQLTDPV	No
SQLPCTF	Yes	SQLTDPV	No
SQLPCTI	Yes	SQLTDPV	No
SQLPCTL	Yes	SQLTDPV	No
SQLPCTS	No	SQLMNPL	Yes
SQLPCTY	No	SQLMFNL	Yes
SQLPCXP	Yes	SQLMFNL	Yes
SQLPDBM	No	SQLTDPV	No
SQLPDBN	Yes	SQLMFNL	Yes
SQLPDDB	No	Character field of size SQLMDNM + 1	Yes
SQLPDDR	Yes	SQLMFNL	Yes
SQLPDIS	Yes	SQLTDPV	No
SQLPDLK	Yes	SQLTDPV	Yes
SQLPDMO	Yes	SQLTDPV	No
SQLPDTL	Yes	SQLTDPV	No
SQLPDTR	No	SQLTDPV	No
SQLPDUS	No	Character field of size SQLMSID + 1	Yes
SQLPEMT	Yes	SQLMXER	Yes
SQLPERF	No	SQLMFNL	Yes
SQLPEXE	Yes	SQLMFNL	Yes
SQLPEXP	Yes	SQLMFNL	Yes
SQLPEXS	No	SQLMFNL	No
SQLPFNM	Yes	SQLMFNL	No
SQLPFRS	Yes	SQLTDPV	No
SQLPFT	Yes	SQLTDPV	No

parm	*cur*	*pbuf*	*len*
SQLPGBC	No	Pass the address of a cursor	No
SQLPGCD	Yes	SQLTDPV	No
SQLPGCM	Yes	SQLTDPV	No
SQLPHEP	Yes	SQLTDPV	No
SQLPHFS	Yes	SQLTDPV	No
SQLPISO	Yes	SQLMFNL	Yes
SQLPLBM	Yes	SQLTDPV	No
SQLPLCK	No	SQLTDPV	No
SQLPLDR	Yes	SQLMFNL	Yes
SQLPLDV	Yes	SQLMFNL	Yes
SQLPLFF	Yes	SQLTDPV	No
SQLPLFS	Yes	SQLTDPV	No
SQLPLGF	Yes	SQLTDPV	No
SQLPLOC	Yes	SQLTDPV	No
SQLPLRD	Yes	SQLMFNL	Yes
SQLPLSS	Yes	SQLMFNL	Yes
SQLPMID	Yes	SQLMFNL	Yes
SQLPMUL	Yes	SQLTDPV	No
SQLPNCK	No	SQLTDPV	No
SQLPNCT	No	SQLTDPV	No
SQLPNDB	Yes	SQLTDPV	No
SQLPNID	Yes	SQLMFNL	Yes
SQLPNIE	No	SQLTDPV	No
SQLPNLB	Yes	SQLTDPV	No
SQLPNLG	No	SQLMFNL	Yes

parm	cur	pbuf	len
SQLPNPB	Yes	SQLTDPV	No
SQLPNPF	No	SQLMFNL	Yes
SQLPOFF	Yes	SQLTDPV	No
SQLPOMB	Yes	SQLTDPV	No
SQLPOOJ	No	SQLTDPV	No
SQLPORID	Yes	SQLMFNL	No
SQLPOPL	Yes	SQLTDPV	No
SQLPOSR	Yes	SQLTDPV	No
SQLPOVR	Yes	SQLTDPV	No
SQLPPAR	Yes	SQLTDPV	No
SQLPPCX	Yes	SQLTDPV	No
SQLPPDB	No	SQLTDPV	No
SQLPPLF	Yes	SQLTDPV	No
SQLPPLV	Yes	SQLTDPV	No
SQLPPTH	Yes	SQLTDPV	No
SQLPREC	Yes	SQLTDPV	No
SQLPRES	Yes	SQLTDPV	No
SQLPRID	Yes	Character field of size RIDSIZ * 2 + 1	Yes
SQLPROD	Yes	SQLTDPV	No
SQLPROT	Yes	SQLTDPV	No
SQLPRTO	Yes	SQLTDPV	No
SQLPSCR	Yes	SQLTDPV	No
SQLPSIL	No	SQLTDPV	No
SQLPSTA	Yes	SQLTDPV	No
SQLPSTC	No	PQLTDPV	No

parm	cur	pbuf	len
SQLPSVN	Yes	SQLMFNL	Yes
SQLPSWR	No	SQLTDPV	No
SQLPTCO	No	SQLTDPV	No
SQLPTHM	No	SQLTDPV	No
SQLPTMS	Yes	SQLTDPV	No
SQLPTMO	No	SQLTDPV	No
SQLPTMZ	No	SQLDPV	No
SQLPTPD	No	SQLMFNL	Yes
SQLPTRC	Yes	SQLTDPV	No
SQLPTRF	Yes	Character field size of SQLMFNL + 1	Yes
SQLPTSL	Yes	SQLTDPV	No
SQLPTSS	No	SQTDPV	No
SQLPUID	Yes	SQKMFNL	Yes
SQLPUSR	No	SQLTDPV	No
SQLPVER	Yes	SQLMFNL	Yes
SQLPWFC	Yes	SQLTDPV	Yes
SQLWKA	No	SQLTDPV	No
SQLWKL	No	SQLTDPV	No
SQLPWTO	Yes	SQLTDPV	No

Return value

If this function returns zero, it was successful. If this function returns a non-zero value, it was unsuccessful.

Example

```
chardbn[SQLMDNM + 1]; /* database name buffer */
SQLTDAL  dbl;         /* database name length */
SQLTRCD  rcd;         /* return code*/

if (rcd = sqlget(0, SQLPDDB, dbn, &dbl))/* get dbname */
   printf("Failure Getting Database Name (rcd = %d)\n",rcd);
dbn[dbl] = 0;  /* concatenate null terminator */
printf("Default Database Name: %s\n", dbn);
```

Related functions

sqlset

sqlgfi - Get Fetch Information

Syntax

```
#include <sql.h>

SQLTAPI  sqlgfi (cur,slc, cvl, fsc)

SQLTCUR           cur;    /* Cursor handle */
SQLTSLC           slc;    /* Select column */
SQLTCDL     PTR   cvl;    /* Value length */
SQLTFSC     PTR   fsc;    /* Fetch status code */
```

Description

This function returns information about a column fetched by the most-recent *sqlfet*. The length of the column data in the SELECT buffer and the fetch return code for a specific column value are returned.

Parameters

cur

The cursor handle associated with this function.

slc

The sequence number of the column in the SELECT list (starting with 1) to get information about.

cvl

A pointer to the variable where this function returns the length of the data received into the select buffer from the previous *sqlfet*. If the column contains null values, this function returns zero.

If the size of the buffer where the data is fetched is smaller than the data received, the data is truncated and an error is returned in *fsc*.

If the data received is less than the size of the buffer where the data is fetched, then *cvl* is set to the actual length received. For example, if the string "TEST" is received into a 20 character variable, *cvl* is set to 4.

You can pass a null pointer (SQLNPTR) if this information is not wanted by the application.

fsc

A pointer to the variable where this function returns the fetch status code for the column retrieved by the previous *sqlfet*.

You can pass a null pointer (SQLNPTR) if this information is not wanted by the application.

The following is a list of the fetch status codes which can be returned. These codes are defined in *sql.h*.

Status Code	Value	Explanation
FETRTRU	1	Data was truncated.
FETRSIN	2	Signed number fetched into unsigned field.
FETRDNN	3	Data is not numeric.
FETRNOF	4	Numeric overflow.
FETRDTN	5	Data type not supported.
FETRDND	6	Data is not a date.

Return value

If this function returns zero, it was successful. If this function returns a non-zero value, it was unsuccessful.

Example

```
static char sqlsel[] = "select name, phone, apt from
                        tenants";
char          fsc;     /* fetch status code */
unsigned char cvl;     /* column value length */
char          col = 1; /* first column*/
short         ret;     /* return code*/
uchar         nsi;     /* number of select items */

sqlnsi(cur, &nsi);/* get # of select items */
while (!(ret = sqlfet(cur)))/* fetch each row */
   {          /* get fetch info for each column */
      while (col++ <= nsi)
      {
             if (sqlgfi(cur, col, &cvl, &fsc))
                break; /* error */
              if (fsc)
                do something/* Process fetch status */
         }
           if (ret) break;
   }
```

Related functions

sqlfet

sqlgls - Get Long Size

Syntax

```
#includes <sql.h>

SQLTAPI  sqlgls (cur, slc , size)

SQLTCUR          cur;      /* Cursor handle */
SQLTSLC          slc;      /* Select number */
SQLTLSI   PTR    size;     /* Size of long column */
```

Description

This function returns the length of the data in a LONG VARCHAR column. This function is called after *sqlfet* to determine the size to read. The returned size can be passed to *sqlrlo*.

Parameters

cur

The cursor handle associated with this function.

slc

The column sequence number (starting with 1) of the column in the SELECT list.

size

A pointer to the variable where this function returns the number of bytes in the LONG VARCHAR column.

Note: Be sure to return this value into an unsigned long variable to accommodate numbers greater than 32K.

Return value

If this function returns zero, it was successful. If this function returns a non-zero value, it was unsuccessful.

Example

```
static char select[] = 'select name, biography from people
          where birthplace = :1";

/* Get length of biography column */

long size;
ret = sqlgls(cur, 2, &size);
```

Related functions

sqlelo *sqllsk* *sqlrlo*

sqlgnl - Get Next Log

Syntax

```
#include <sql.h>

SQLTAPI  sqlgnl (shandle, dbname, dbnamel, lognum)

SQLTSVH           shandle;  /* Server handle */
SQLTDAP           dbname;   /* Database name */
SQLTDAL           dbnamel;  /* Database name length */
SQLTLNG   PTR     lognum;   /* Returned log number */
```

Description

This function returns the name of the next transaction log file needed for recovery.

If the specified transaction log file is not available, you should call the *sqlenr* function to finish the recovery of the database.

Parameters

shandle

The server handle returned by *sqlcsv.*

dbname

A pointer to the string that contains the database name.

dbnamel

The length of the string pointed to by *dbname*. If the string pointed to by *dbname* is null-terminated, specify zero and the system will compute the length.

lognum

A variable where this function returns the number of the next log file. This function returns zero in this variable if the next log file needed is already on disk.

Return value

If this function returns zero, it was successful. If this function returns a non-zero value, it was unsuccessful.

Example

```
SQLTSVH     shandle;
char*       password;
SQLTDPV     lbmset;
SQLTFNP     bkpdir;
SQLTFNL     bkpdirl;
SQLTRFM     mode=SQLMEOL;
SQLTLNG     lognum;
SQLTBOO     local,over;

static chardbname1[] = "omed";

password = 0;
bkpdir = "\\BACKUP\\OMED";
bkpdirl = strlen(bkpdir);
printf("value of bkpdir = %s \n",bkpdir);

local=1;
over=1;

/* CONNECT TO SERVER */

if (rcd = sqlcsv(&shandle,srvname,password))
   apierr("SQLCSV");

/* RESTORE DATABASE */

if (rcd =
sqlrdb(shandle,dbname1,0,bkpdir,bkpdirl,local,over))
     apierr("SQLRDB");
else
```

```
   printf("Restored Database \n");
/* ROLLFORWARD TO END */

sqlrof(shandle,dbname1,0,mode,0,0);

lognum=0;
/*
   The loop below assumes that all log file backups are on
   disk.
   If a log file backup is not on disk, lognum is set to a
non-
   zero value which causes the loop to terminate.
*/
while (lognum == 0)
   {
      /* GET NEXT LOG */
      sqlgnl(shandle,dbname1,0,&lognum);

        /* RESTORE LOG FILES */

sqlrlf(shandle,dbname1,0,bkpdir,bkpdirl,local,over);
   }

/* END ROLLFORWARD */

if (rcd = sqlenr(shandle,dbname1,0))
   apierr("SQLENR");
else
   printf("End Rollforward \n");
```

Related functions

sqlbdb	*sqlcsv*	*sqlrel*
sqlblf	*sqlenr*	*sqlrlf*
sqlbss	*sqlrdb*	*sqlrof*
sqlcrf		

sqlgnr - Get Number of Rows

Syntax

```
#include <sql.h>

SQLTAPI  sqlgnr (cur, tbname, tbnaml, rows)

SQLTCUR        cur;      /* Cursor handle */
SQLTDAP        tbname;   /* Table name */
SQLTDAL        tbnaml;   /* Table name length */
SQLTROW  PTR   rows;     /* Total number of rows */
```

Description

This function returns the number of rows in the specified table from the system catalog. It is faster than executing a SELECT COUNT(*) command without a WHERE clause. You can only use this function for SQLBase databases.

Parameters

cur

The cursor handle associated with this function.

tbname

A pointer to the string that contains the table name.

tbnaml

The length of the string pointed to by *tbname*. If the string pointed to by *tbname* is null-terminated, specify zero and the system will compute the length.

rows

A pointer to the variable where this function returns the number of rows in the table.

Return value

The return value is zero (0) if the function succeeds and non-zero if it fails.

Example

```
/* Get the number of rows in the CUSTOMER table */

long   custcnt;
short  ret; /* return code */

ret = sqlgnr(cur, "CUSTOMER", 0, &custcnt);
```

Related functions

sqlepo *sqlnrr* *sqlrow*

sqlgsi - Get Server Information

Syntax

```
#include <sql.h>
#include <sqlsrv.h>
#include <gsiext.h>

SQLTAPI  sqlgsi(shandle, infoflags, buffer, buflen, rbuflen)

SQLTSVH       shandle;    /* Server handle */
SQLTFLG       infoflags;  /* Information flags */
SQLTDAP       buffer;     /* Buffer to read into */
SQLTDAL       buflen;     /* Size of data buffer */
SQLTDAL PTR   rbuflen;    /* Length of read */
```

Description

This function returns server information.

The format of the information returned by this function is defined in *sqlsrv.h* and *gsiext.h*.

Parameters

shandle

The server handle returned by *sqlcsv*.

infoflags

Server information flags which can be logically OR'd to return combinations of information.

The actual length of data returned for any information type is determined by the extended information flag (SQLXGSI). If you OR the extended information flag with other server information flags, additional information follows the default information structure.

Flag	SQLXGSI flag	sqlsrv.h structure	gsiext.h structure	Information
SQLGCFG	No	cfgdef		Configuration information
SQLGCFG	Yes	cfgdef	cfgdefi	Extended configuration information
SQLGCUR	No	curdef		Cursor information
SQLGCUR	Yes	curdef	curdefi	Extended cursor information
SQLGDBS	No	dbsdef		Database information
SQLGLCK	Yes	lckdef		Lock information
SQLGOSS	No	---	ostdef	Operating system statistics
SQLGPRC	No	prcdef		Process information
SQLGPRC	Yes	prcdef	prcdefi	Extended process information
SQLGPWD	No	---		Send password
SQLGSTT	No	sttdef		Statistics
SQLRCLN	Yes	fgidef		Filter by client name
SQLRDBN	Yes	fgidef		Filter by database name
SQLRPNM	Yes	fgidef		Filter by process number
SQLRUSN	Yes	fgidef		Filter by user name
SQLXGSI	No	---	---	Return extended information.

Note: SQLGDBS only returns information on databases that the server is listening on because that is the only time the information is available. Use the *sqldbn* function to find the databases that the server is listening on.

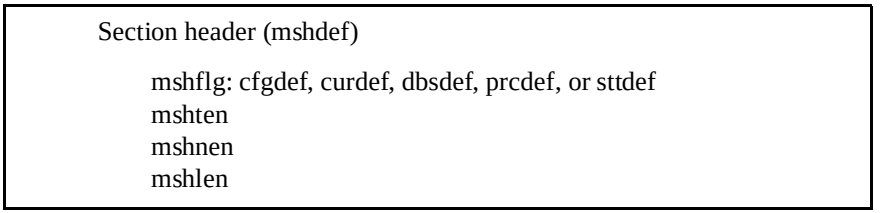
buffer

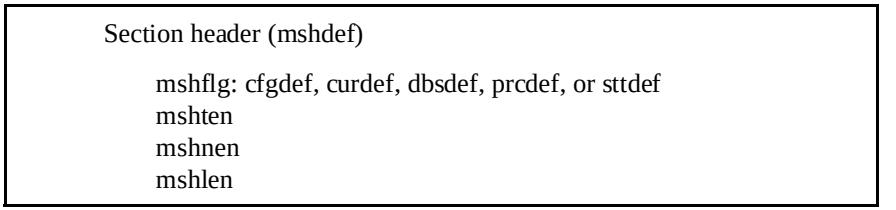
A pointer to the variable where this function returns the server information.

Using the filter flags to filter the amount of returned information requires the fgidef structure to be placed at the beginning of the buffer and filled with the filter information. The fgidef structure will be sent to the server. The returned information will be restricted to the process number, client name, user name, or database name, depending on the filter flags set.

As defined in *sqlsrv.h*, the information returned has a message header (*hdrdef*) that contains the length (*hdrlen*) of the entire message including the message header.

The message header is followed by a separate section for each type of information requested. These sections start with a section header (*mshdef*) that contains the information type (*mshflag*) contained in the section, the total number of entries (*mshten*), the number of entries in the message (*mshnen*), and the number of bytes in that section (including the section header). Finally, each section contains the requested information.





The sqlgsi function will not overflow the message buffer. The mshten and mshnen fields are equal if the message buffer contains all the entries; otherwise mshnen indicates how many entries were actually placed in the message buffer.

If not all the information is present, you can pass a larger buffer size or use the filter flags to break the request into multiple requests.

buflen

The length of the value pointed to by *buffer*.

rbuflen

A pointer to the variable where this function returns the length of the server information.

Return value

The return value is zero (0) if the function succeeds and non-zero if it fails.

Example

See the example program *ex22.c* for a comprehensive example.

```
SQLTSVH     handle;
char        buf[4000];
SQLTDAL     blen;
SQLTRCD     rcd;

if ((rcd = sqlcsv(&handle, srvname, password)) != 0)
   {
```

```
        sqlgsi(handle, SQLGDBS | SQLGSTT, buf, sizeof(buf),
               &blen);
        sqldsv(handle);
    }
```

Related functions

sqlcsv *sqlsta*

sqlims - Input Message Size

Syntax

```
#include <sql.h>

SQLTAPI   sqlims(cur, insize)

SQLTCUR   cur;      /* Cursor number */
SQLTDAL   insize;   /* Input message buffer size */
```

Description

This function changes the maximum size (in bytes) of the input message buffer. The input message buffer is allocated on both the client computer and on the database server. The database server builds an input message in this buffer on the database server computer and sends it across the network to a buffer of the same size on the client. It is called an *input* message buffer because it is input from the client's point of view.

There is one input message buffer per connected cursor on the client computer. The server maintains an input message buffer that is the size of the largest input message buffer on the client computer.

The input message buffer can receive a return code indicating that the specified operation was successful, the data that is being fetched, and other information. While fetching data from the database, SQLBase compacts as many rows as possible into one input message buffer.

Each *sqlfet* call reads the next row from the input message buffer until they are exhausted. At this instant, SQLBase transparently fetches the next input buffer of rows depending on the isolation level.

A large input message buffer can help performance while fetching data from the database because it reduces the number of network messages. Note that a large input message buffer can affect system throughput because of concurrency. Any row

currently in the input message buffer can have a shared lock on it (depending on the isolation level) preventing other users from changing that row. Therefore, a large input message buffer can cause more shared locks to remain than are necessary.

See the explanation of *sqlsil* for more information about how each isolation level uses the input message buffer.

SQLBase automatically maintains an input message buffer large enough to hold at least one row of data. Despite the specified input message size, SQLBase dynamically allocates more space if necessary.

A large input message buffer helps performance when reading LONG VARCHAR columns.

This function can also improve overall system performance by decreasing the size of the input message buffer when an application does not need to fetch data.

Parameters

cur

The cursor handle associated with this function. Each cursor has one input message buffer associated with it on the client.

insize

The size of the input message buffer in bytes. Specify a zero to indicate that you want to use the default input message buffer size in *sql.h* (2000).

Return value

The return value is zero (0) if the function succeeds and non-zero if it fails.

Example

```
main()
{

   SQLTDAL   insize=500;
   SQLTDAL   outsize=500;

staticchardbnam[] = "demox";/* database name */

/* CONNECT TO THE DATABASE */

cur = 0;
```

```
if (rcd = sqlcnc(&cur, dbnam, 0))/* perform connect
operation */
   apierr("SQLCNC");

if (rcd = sqlims(cur,insize))
   apierr("SQLIMS");
else
   printf("Input Message Size set to = %d \n", insize);

if (rcd = sqloms(cur,outsize))
   apierr("SQLOMS");
else
   printf("Output Message Size set to = %d \n", outsize);

/* DISCONNECT FROM THE DATABASE */

if (rcd = sqldis(cur))/* failure on disconnect? */
   apierr("SQLDIS");
}
```

Related functions

sqloms *sqlsil*

sqlind - INstall Database

Syntax

```
#include <sql.h>

SQLTAPI   sqlind (shandle, dbname, dbnamel)

SQLTSVH   shandle;    /* Server handle */
SQLTDAP   dbname;     /* Database name */
SQLTDAL   dbnamel;    /* Database name length */
```

Description

This function installs a database on the network and adds a *dbname* keyword to *sql.ini*.

This function does not physically create a database. Call *sqlcre* to create a database.

Parameters

shandle

The server handle returned by *sqlcsv*.

dbname

A pointer to the string that contains the database name.

dbnamel

The length of the string pointed to by *dbname*. If the string pointed to by *dbname* is null-terminated, specify zero and the system will compute the length.

Return value

The return value is zero (0) if the function succeeds and non-zero if it fails.

Example

```
main()
{
   srvname = "SERVER1";
   password = 0;

/* CONNECT TO THE SERVER */

if (rcd = sqlcsv(&handle,srvname,password))
   apierr("SQLCSV");
else
   printf("Connection Established to Server \n");

if (rcd = sqlcre(handle,"DEMOX",0))
   apierr("SQLCRE");
else
   printf("Database DEMOX Created \n");

/* DEINSTALL DATABASE */

if (rcd = sqlded(handle,"DEMOX",0))
   apierr("SQLDED");
else
   printf("Database DEMOX Deinstalled \n");

/* INSTALL DATABASE */

if (rcd = sqlind(handle,"DEMOX",0))
```

```
    apierr("SQLIND");
else
    printf("Database DEMOX Installed \n");

/* DISCONNECT FROM THE SERVER */

if (rcd = sqldsv(handle))
    apierr("SQLDSV");
else
    printf("Disconnected from Server \n");
}
```

Related functions

sqlcre *sqlded* *sqldel*
sqlcsv

sqlini - INItialize

Syntax

```
#include  <sql.h>

SQLTAPI  sqlini (callback)

SQLTPFP  callback; /* Callback yield function */
```

Description

This function initializes the dynamic library used for a Windows 3.x, Windows 95, or Windows NT application.

This function also sets up a callback function so control can pass to the operating system while a function is executing on the server. Although this callback function is not necessary in single-user mode, you should use it to maintain portability to a multi-user environment. Callback is only needed by Windows 3.x, not Windows NT nor Windows 95.

For a Windows 3.x application, this function initializes the dynamic library and sets up a callback function so the application will yield control while it is waiting for a response form the database server. This callback function is necessary to allow smooth multi-tasking of other Windows applications. Although this callback function is not necessary in single-user mode, you should use it to maintain portability to a multi-user environment.

For Windows, you might call the *sqlini* function as follows:

```
sqlini(MakeProcInstance(YieldProc, hInstance));
```

Windows NT and Windows 95 do not use a callback function. However, the application must still call the sqlini function so that other initialization can take place. On these platforms, call the *sqlini* function in your program as follows:

```
sqlini((SQLTPFP) (0));
```

Call this function before the first database connect.

You must ensure that SQLBase is not called in the yield function. If SQLBase is called while in the yield function, the results are unpredictable.

See *testwin.c* for an example of how to use this function.

Parameters

callback

A far pointer to a callback function that is called to yield to other applications when the server is processing a request. If this argument is null, control remains with the application. If you specify a non-null callback function, it is ignored.

Return value

The return value is zero (0) if the function succeeds and non-zero if it fails.

Example

```
int PASCAL WinMain(hInstance, hPrevInstance, lpszCmdLine,
                   cmdShow)

HANDLE    hInstance;
HANDLE    hPrevInstance;
LPSTR     lpszCmdLine;
int       cmdShow;

{

shortrcd;

if (rcd = sqlini(MakeProcInstance(YieldProc, hInstance)))
  {
     prints("Cannot initialize API interface - %u\n",rcd);
     return FALSE;
  }
```

```
int FAR PASCAL YieldProc()
  {

    MSG msg;

    while( PeekMessage( &msg, NULL, 0, 0, PM_REMOVE ) )
       {
         TranslateMessage( &msg );
         DispatchMessage( &msg );
       }
     return TRUE;/* return successfully*/
    }
}
```

Related functions

sqldon

sqllab - LABel information

Syntax

```
#include<sql.h>

SQLTAPI   sqllab(cur, slc, lpb, lblp)

SQLTCUR   cur;      /* Cursor number */
SQLTSLC   slc;      /* Select column number */
SQLTCHP   lbp;      /* Buffer to retrieve label */
SQLTCHL   lblp;     /* Label name length */
```

Description

This function returns label information for the specified column in a SELECT command.

Labels are text strings that document table columns. Labels are stored in the system catalog table SYSTABLES in the LABEL column. LABELs can be up to 30 characters in length.

A successful compile of a SELECT command must come immediately before this function.

An application can loop through all the columns to get the label information column by column. The *sqlnsi* function returns the number of columns in a SELECT list.

Parameters

cur

The cursor handle associated with this function.

slc

The column number (starting with 1) in the SELECT list to get information about. The column number can be used to set up a loop to describe all columns in the SELECT list.

lbp

A pointer to the variable where this function returns the label.

lblp

A pointer to the variable where this function returns the length of the label.

Return value

The return value is zero (0) if the function succeeds and non-zero if it fails.

Example

```
SQLTCUR  cur1 = 0;   /* SQLBASE cursor */

main()

SQLTSLC slc;         /* Select list column */
SQLTCHL chl;         /* Column header length */
uchar chbuf[300];    /* Column header buffer */

if (sqllab(curlab, slc, chbuf, &chl))
  ... process error

chbuf[chl] = '\0';
printf("Label header = %s\n" chbuf);
printf("Label header length = %d\n", chl);
```

Related functions

sqlgdi *sqlnsi*

sqlldp - LoaD oPeration

Syntax

```
#include <sql.h>

SQLTAPI   sqlldp (cur, cmdp, cmdl)

SQLTCUR   cur;      /* cursor number    */
SQLTDAP   cmdp;     /* -> command buffer    */
SQLTDAL   cmdl;     /* command length    */
```

Description

This function processes the LOAD command and sends it to the backend for compilation and execution. If the load source file resides on the server, the execution is handled completely at the server. If it is on the client, this function handles the retrieval of load data and sends it to the server, in chunks.

Parameters

cur

The cursor handle associated with this function.

cmdp

A pointer to the string that contains the LOAD command.

cmdl

The length of the string pointed to by *cmdp*. If the string pointed to by *cmdp* is null-terminated, specify zero and the system will compute the length.

Return Value

If this function returns zero, it was successful. If this function returns a non-zero value, it was unsuccessful, and returns an error code.

Example

The following sample program calls the LOAD command and inputs a file name that exists online:

```
static char loadcmd[] =
"LOAD SQL db.unl ON SERVER";
```

```
ret = sqlldp(cur, loadcmd, 0);
```

You can also create a customized program to manipulate the load input buffer in the client. For an example, see the *Loading and unloading databases* section in the chapter *Using the SQL/API*.

Related functions

sqlunl

sqllsk - Long SeeK

Synopsis

```
#include <sql.h>

SQLTAPI   sqllsk (cur,slc,pos)

SQLTCUR   cur;      /* Cursor handle */
SQLTSLC   slc;      /* Select column */
SQLTLSI   pos;      /* Desired byte position */
```

Description

This function sets the position to start reading within a LONG VARCHAR column. In other words, you do not have to start *sqlrlo* reading a LONG VARCHAR at the first byte.

You *cannot* seek to a position within a LONG VARCHAR to start writing with *sqlwlo*. The *sqlwlo* function must write the entire LONG VARCHAR column.

You must call this function after *sqlfet* and before *sqlrlo*.

If the requested byte position is beyond the end of the data, this function returns an error.

Parameters

cur

The cursor handle associated with this function.

slc

The column number in the SELECT list. The first column is column 1.

pos

The byte position within the LONG VARCHAR column to start reading. Byte position 1 is the first byte in the LONG VARCHAR column.

Return value

The return value is zero (0) if the function succeeds and non-zero if it fails.

Example

```
static char sqlsel[] = "select biography from people where
          name = :1"

/* position read to last 80 bytes of the long */
long size;

/* Get size of long */
if (!(ret = sqlgls(cur, 1, &size)))
   {
     if (!(ret = sqllsk(cur, 1, size-80))/* set position */
     ... process error (sqllsk)
   }
else
   process error
```

Related functions

sqlrlo *sqlfet* *sqlelo*

sqlmcl - reMote CLose server file

Syntax

```
#include <sql.h>

SQLTAPI   sqlmcl (shandle, fd)

SQLTSVH   shandle;    /* Server handle */
SQLTFLH   fd;         /* File handle */
```

Description

This function closes a file on the server.

You must first open the server file using *sqlmop*.

Parameters

shandle

The server handle returned by *sqlcsv*.

fd

The file handle returned by *sqlmop*.

Return value

The return value is zero (0) if the function succeeds and non-zero if it fails.

Example

```
{
  unsigned    char buffer[1024];
  SQLTSVH     handle;
  SQLTSVN     srvno;
  SQLTDAP     password;
  SQLTFLH     fdin;
  SQLTFLH     fdout;
  SQLTDAL     len;
  SQLTDAL     rlen;
...
if ((ret = sqlcsv(&handle, srvno, password)) == 0)
  {
  if ((rcd = sqlmop(handle, &fdin, "infile",SQLORDONLY |
            SQLOBINARY)) == 0)
    {
      if ((rcd = sqlmop(handle, &fdout, "outfile",SQLOCREAT
                | SQLOTRUNC | SQLOWRONLY | SQLOBINARY)) == 0)
        {
        for (;;)
            {
          rcd = sqlmrd(handle, fdin, buffer, sizeof(buffer),
            &len);
          if (rcd != 0 || rlen == 0)
            break;
          rcd = sqlmwr(handle, fdout, buffer, len, &rlen);
          if (rcd != 0 || len != rlen)
            break;
          }
```

```
            rcd = sqlmcl(handle, fdout);
            }
            rcd = sqlmcl(handle, fdin);
          }
        sqldsv(handle);
        }
    }
```

Related functions

sqlcsv	*sqlmop*	*sqlmsk*
sqlmdl	*sqlmrd*	*sqlmwr*

sqlmdl - reMote DeLete server file

Syntax

```
#include <sql.h>

SQLTAPI   sqlmdl (shandle, filename)

SQLTSVH   shandle;    /* Server handle */
SQLTDAP   filename;   /* File name to delete */
```

Description

This function deletes a file on the server.

Note: SQLBase supports filenames up to 256 characters including the terminating null character.

Parameters

shandle

The server handle returned by *sqlcsv*.

filename

A pointer to the null-terminated string that contains the name of the file to delete.

Return value

The return value is zero (0) if the function succeeds and non-zero if it fails.

Example

```
sqlmdl(shandle, filename);
```

Related functions

sqlcsv	*sqlmop*	*sqlmsk*
sqlmcl	*sqlmrd*	*sqlmwr*

sqlmop - reMote OPen server file

Syntax

```
#include <sql.h>
#include <sqlsrv.h>

SQLTAPI     sqlmop (shandle, fdp, filename, openmode)

SQLTSVH     shandle;    /* Server handle */
SQLTFLH PTR fdp;        /* File handle */
SQLTDAP     filename;   / * File name to open or create */
SQLTFMD     openmode;   /* File open mode */
```

Description

This function opens or creates a file on the server.

There is a limit of four file handles open per each server connect.

Note: SQLBase supports filenames up to 256 characters including the terminating null character.

Parameters

shandle

The server handle returned by *sqlcsv*.

fdp

The file handle returned by *sqlmop*.

filename

A pointer to the null-terminated string that contains the name of the file to open or create.

openmode

The type of operations allowed. This argument is formed by combining one or more of the constants in the following table.

When more than one constant is specified, the constants are joined with the bitwise OR operator (|). These constants are defined in *sqlsrv.h* and are listed in the table below.

Constant	Description
SQLOAPPEND	Reposition the file pointer at the end of the file before every write.
SQLOCREAT	Create and open a new file for writing. Has no effect if *filename* exists.
SQLOEXCL	Return an error value if *filename* exists. Used only with SQLOCREAT.
SQLORDONLY	Open file for reading only. If this is specified, neither SQLORDWR nor SQLOWRONLY can be given.
SQLORDWR	Open file for both reading and writing. If this is specified, neither SQLORDONLY nor SQLOWRONLY can be specified.
SQLOTRUNC	Open and truncate an existing file to zero length. The file must have write permission. The file contents are destroyed.
SQLOWRONLY	Open file for writing only. If this is given, neither SQLORDONLY nor SQLORDWR can be given.
SQLOBINARY	Open file in binary mode.
SQLOTEXT	Open file in text mode.
SQLODIRCREA	Create directory.

Return value

The return value is zero (0) if the function succeeds and non-zero if it fails.

Example

This example copies *infile* on the server to *outfile* on the server.

```
{
  unsigned    char buffer[1024];
  SQLTSVH     handle;
  SQLTSVN     srvno;
  SQLTDAP     password;
  SQLTFLH     fdin;
  SQLTFLH     fdout;
  SQLTDAL     len;
  SQLTDAL     rlen;

...
if ((ret = sqlcsv(&handle, srvno, password)) == 0)
  {
  if ((rcd = sqlmop(handle, &fdin, "infile", SQLORDONLY |
            SQLOBINARY)) == 0)
      {
      if ((rcd = sqlmop(handle, &fdout, "outfile", SQLOCREAT
             | SQLOTRUNC | SQLOWRONLY | SQLOBINARY)) == 0)
        {
        for (;;)
          {
          rcd = sqlmrd(handle, fdin, buffer, sizeof(buffer),
                       &len);
            if (rcd != 0 || rlen == 0)
            break;
          rcd = sqlmwr(handle, fdout, buffer, len, &rlen);
          if (rcd != 0 || len != rlen)
            break;
          }
        rcd = sqlmcl(handle, fdout);
        }
        rcd = sqlmcl(handle, fdin);
      }
    sqldsv(handle);
    }
}
```

Related functions

sqlcsv *sqlmop* *sqlmsk*
sqlmcl *sqlmrd* *sqlmwr*
sqlmdl

sqlmrd - reMote ReaD server file

Syntax

```
#include <sql.h>

SQLTAPI       sqlmrd (shandle, fd, buffer, len, rlen)

SQLTSVH       shandle;  /* Server handle */
SQLTFLH       fd;       /* File handle */
SQLTDAP       buffer;   /* Read buffer */
SQLTDAL       len;      /* Read length */
SQLTDAL  PTR  rlen;     /* Number of bytes read */
```

Description

This function reads *len* bytes from the file associated with *fd* into *buffer*. The read operation begins at the current position of the file pointer associated with the file. After the read operation, the file pointer is positioned at the next unread character.

Parameters

shandle

The server handle returned by *sqlcsv*.

fd

The file handle returned by *sqlmop*.

buffer

A pointer to the variable where this function returns the data that is read.

len

The number of bytes to read.

rlen

A pointer to the variable where this function returns the number of bytes read into *buffer*.

When this function returns zero in *rlen*, it has reached the end of file.

Return value

The return value is zero (0) if the function succeeds and non-zero if it fails.

Example

```
{
  unsigned char buffer[1024];
  SQLTSVH     handle;
  SQLTSVN     srvno;
  SQLTDAP     password;
  SQLTFLH     fdin;
  SQLTFLH     fdout;
  SQLTDAL     len;
  SQLTDAL     rlen;
...
if ((ret = sqlcsv(&handle, srvno, password)) == 0)
    {
    if ((rcd = sqlmop(handle, &fdin, "infile", SQLORDONLY |
            SQLOBINARY)) == 0)
      {
      if ((rcd = sqlmop(handle, &fdout, "outfile", SQLOCREAT
           | SQLOTRUNC | SQLOWRONLY | SQLOBINARY)) == 0)
        {
        for (;;)
          {
          rcd = sqlmrd(handle, fdin, buffer, sizeof(buffer),
rlen);
          if (rcd != 0 || rlen == 0)
            break;
          rcd = sqlmwr(handle, fdout, buffer, len, &rlen);
          If (rcd != 0 || len != rlen)
            break;
          }
        rcd = sqlmcl(handle, fdout);
        }
        rcd = sqlmcl(handle, fdin);
      }
    sqldsv(handle);
    }
}
```

Related functions

sqlcsv *sqlmop* *sqlmwr*
sqlmdl *sqlmsk*

sqlmsk - reMote SeeK server file

Syntax

```
#include <sql.h>

SQLTAPI     sqlmsk (shandle, fd, offset, whence, roffset);

SQLTSVH     shandle;  /* Server handle */
SQLTFLH     fd;       /* File handle */
SQLTLNG     offset;   /* Seek offset */
SQLTWNC     whence;   /* Seek origin */
SQLTLNGPTR  roffset;  /* Resulting seek address */
```

Description

This function moves the file pointer for *fd* to a new location that is *offset* bytes from *whence*. This function returns the new location in *roffset*. The next operation on the file occurs at the new *roffset* location.

Parameters

shandle

The server handle returned by *sqlcsv*.

fd

The file handle returned by *sqlmop*.

offset

The number of bytes from *whence*.

whence

The position where the seek begins:

0	Seek relative to beginning of file.
1	Seek relative to current position.

2	Seek relative to end of file.

roffset

The resulting offset of the new position from the beginning of the file.

Return value

The return value is zero (0) if the function succeeds and non-zero if it fails.

Example

```
sqlmsk(shandle, fhandle, offset, whence, roffset);
```

Related functions

sqlcsv *sqlmdl* *sqlmrd*
sqlmcl *sqlmop* *sqlmwr*

sqlmwr - reMote WRite server file

Syntax

```
#include <sql.h>

SQLTAPI   sqlmwr (shandle, fd, buffer, len, rlen);

SQLTSVH       shandle;  /* Server handle */
SQLTFLH       fd;       /* File handle */
SQLTDAP       buffer;   /* Data to write */
SQLTDAL       len;      /* Length of buffer */
SQLTDAL  PTR  rlen;     /* Number of bytes written */
```

Description

This function writes *len* bytes from the *buffer* into the file associated with *fd*. The write operation begins at the current position of the file pointer associated with the given file. If the file is opened for appending, the operation begins at the current end of the file. After the write operation, the file pointer is incremented by the number of bytes actually written.

Parameters

shandle

The server handle returned by *sqlcsv*.

fd

The file handle returned by *sqlmop*.

buffer

A pointer to the variable that contains the data to write.

len

The number of bytes to write from *buffer*.

rlen

A pointer to the variable where this function returns the number of bytes actually written.

Return value

The return value is zero (0) if the function succeeds and non-zero if it fails.

Example

```
{
    unsigned    char buffer[1024];
    SQLTSVH     handle;
    SQLTSVN     srvno;
    SQLTDAP     password;
    SQLTFLH     fdin;
    SQLTFLH     fdout;
    SQLTDAL     len;
    SQLTDAL     rlen;
...
if ((ret = sqlcsv(&handle, srvno, password)) == 0)
    {
    if ((rcd = sqlmop(handle, &fdin, "infile", SQLORDONLY |
              SQLOBINARY)) == 0)
      {
      if ((rcd = sqlmop(handle, &fdout, "outfile", SQLOCREAT
               SQLOTRUNC | SQLOWRONLY | SQLOBINARY)) == 0)
        {
        for (;;)
          {
```

```
            rcd = sqlmrd(handle, fdin, buffer, sizeof(buffer),
rlen);
            if (rcd != 0 || rlen == 0)
              break;
            rcd = sqlmwr(handle, fdout, buffer, len, &rlen);
            if (rcd != 0 || len != rlen)
              break;
            }
          rcd = sqlmcl(handle, fdout);
          }
          rcd = sqlmcl(handle, fdin);
        }
      sqldsv(handle);
      }
}
```

Related functions

sqlcsv	*sqlmdl*	*sqlmrd*
sqlmcl	*sqlmop*	*sqlmsk*

sqlnbv - Number of Bind Variables

Syntax

```
#include <sql.h>

SQLTAPI  sqlnbv (cur, nbv)

SQLTCUR          cur      /* Cursor handle */
SQLTNBV   PTR    nbv;     /* Variable */
```

Description

This function returns the number of bind variables in the current SQL command being processed for the specified cursor. The number of bind variables is set after the compile.

Parameters

cur

The cursor handle associated with this function.

nbv

A pointer to the variable where this function returns the number of bind variables.

Return value

The return value is zero (0) if the function succeeds and non-zero if it fails.

Example

```
unsigned char nbv;  /* number of bind variables */
short ret;          /* return code */

ret = sqlnbv(cur, &nbv);
```

Related functions

sqlbld	*sqlbnd*	*sqlbnu*
sqlbln	*sqlbnn*	*sqlcbv*
sqlbna	*sqlbss*	

sqlnii - get the Number of Into variables

Syntax

```
#include <sql.h>

SQLTAPI   sqlnii (cur,nii)

SQLTCUR   cur;      /* Cursor handle*/
SQLTCHL   PTR nii;  /* INTO variable name length*/
```

Description

This function retrieves the number of INTO variables.

Parameters

cur

The cursor handle associated with this function.

nii

A pointer to the variable where this function returns the number of INTO variables.

Return value

If this function returns zero, it was successful. If this function returns a non-zero value, it was unsuccessful.

Example

```
#include "sql32.h"
#include <memory.h>
#include <stdio.h>
#include <stdlib.h>
/*------------------------------------------------------ */
/*                                                       */
/*  Example of a simple fetch                            */
/*                                                       */
/*  Run EMP.SQL via SQLTALK to initialize tables and data */
/*                                                       */
/*------------------------------------------------------ */

SQLTCUR   cur;                  /* SQLBASE cursor number*/
SQLTRCD   rcd;                  /* error number */
char      errmsg[SQLMERR];      /* error msg text buffer*/
void      failure(char*);       /* error routine */
main()
{
   char    name[20];            /* employee name buffer */
   SQLTCHL PTR    nii;
   static  char   selcmd [] = /* SQL SELECT statement   */
      "SELECT EMP_NAME into :name FROM EMP  ";

   /*
    CONNECT TO THE DATABASE
   */

   if (rcd = sqlcnc(&cur, "ISLAND", 0))
   {
     sqlerr(rcd, errmsg);   /* get error message text */
    printf("%s \n",errmsg);
    return(1);
   }
```

```
/*
 COMPILE SELECT STATEMENT
*/

if (sqlcom(cur, selcmd, 0))
 failure("SELECT COMPILE");

/*
 PERFORM sqlnii
*/

if (sqlnii(cur,nii))
     failure ("SQLNII");
else
     printf("Number of select items is %d\n",*nii);

/*
 SET UP SELECT BUFFER
*/

if (sqlssb(cur, 1, SQLPBUF, name, 20, 0, SQLNPTR,
SQLNPTR))
 failure("SET SELECT BUFFER");
/*
 EXECUTE SELECT STATEMENT
*/

if (sqlexe(cur))
 failure("EXECUTING SELECT");

/*
 FETCH DATA
*/

for (;;)
{
  memset(name,' ',20);/* clear employe name buf */

  if (rcd = sqlfet(cur))/* fetch the data         */
   break;

  printf("%s\n", name);/* print employe name  */
}

if (rcd != 1)                /* failure on fetch      */
 failure("FETCH");
```

```
  /*
   DISCONNECT FROM THE DATABASE
  */

  if (rcd = sqldis(cur))
   failure("DISCONNECT");
}
void failure(ep)
char*       ep;                 /* failure msg string  */
{
  SQLTEPO   epo;                /* error position   */

  printf("Failure on %s \n", ep);

  sqlrcd(cur, &rcd);           /* get the error        */
  sqlepo(cur, &epo);           /* get error position */
  sqlerr(rcd, errmsg);         /* get error message text */

  sqldis(cur);                 /* disconnect cursor*/

  printf("%s (error: %u, position: %u) \n",errmsg,rcd,epo);
  exit(1);
  }
```

sqlnrr - Number of Rows in Result set

Syntax

```
#include <sql.h>

SQLTAPI   sqlnrr(cur, rcountp)

SQLTCUR          cur;      /* Cursor handle */
SQLTROW   PTR    rcountp;  /* Number of rows */
```

Description

This function retrieves the number of rows in a result set.

INSERTs into the result set increase the row count but DELETEs, which appear as blanked-out rows in result set mode, do not decrease the row count. However, the deleted rows disappear on the next SELECT.

The program must be in result set mode (enabled with the *sqlsrs* function).

Parameters

cur

The cursor handle associated with this function.

rcountp

A pointer to the variable where this function returns the number of rows in the result set.

Return value

The return value is zero (0) if the function succeeds and non-zero if it fails.

Example

```
main()
{
   char*       p;          /* misc. pointer */
   FILE*       in;         /* input file */
   SQLTDAP     cp;         /* character pointer */

   SQLTDAL     length;     /* length */
   SQLTPDL     pdl;        /* program buffer length */
   int         rows=0;     /* number of rows */
   SQLTDDT     ddt;        /* database data type */
   SQLTPDT     pdt;        /* program data type */
   SQLTBNN     bnn;        /* bind number */
   SQLTSLC     slc;        /* select list column */
   SQLTNBV     nbv;        /* number of bind variables */
   SQLTNSI     nsi;        /* number of select items */
   SQLTROW     nrows=0;
   char        line[200];  /* I/O line */
   nt          i;

...

static    char   selcom[] =    /* SELECT command */
  "SELECT A FROM X WHERE A < 1000";

if (rcd = sqlcnc(&cur, dbnam, 0))
   apierr("SQLCNC");
else
   printf("Connection Established to Database DEMO \n");

...
```

```
if (rcd = sqlcom(cur, selcom, 0))
   apierr("SQLCOM");
cp = line;   /* set pointer to input line */
if (rcd = sqlnsi(cur, &nsi))/* get # select items */
   apierr("SQLNSI");
for (slc = 1; slc <= nsi; slc++)/* get information on each
column */
   {
     if (rcd = sqldes(cur, slc, &ddt, &pdl,SQLNPTR, SQLNPTR,
SQLNPTR, SQLNPTR))
       apierr("SQLDES");
   if (rcd = sqlssb(cur, slc, pdt, cp, pdl, 0, SQLNPTR,
SQLNPTR))
       apierr("SQLSSB");
     cp += (pdl + 1);/* locate next area */
   }

if (rcd = sqlexe(cur))/* failure on select execute? */
   apierr("SQLEXE");

if (rcd = sqlnrr(cur, &nrows))
   apierr("SQLNRR");
else
   printf("Number of rows in Result Set = %d\n",nrows);

length = cp - line;   /* compute the length */
*cp = 0;              /* concatenate a zero to the string */
printf("data: \n");
for (i = 0; i < nrows; i++)
   {
     memset(line, ' ', length);/* fill the line with spaces
*/
     if (rcd = sqlfet(cur))     /* failure or end of file?*/
       break;
     printf("%s\n", line);      /* print the line */
   }
printf("Number of rows fetched = %d \n", i);

if (rcd = sqldis(cur))
   apierr("SQLDIS");

}
```

Related functions

sqlgnr *sqlrow* *sqlsrs*

sqlnsi - Number of Select Items

Syntax

```
#include <sql.h>

SQLTAPI   sqlnsi (cur, nsi)

SQLTCUR     cur;     /* Cursor handle */
SQLTNSI PTR nsi;     /* Number of SELECT items */
```

Description

This function returns the number of items in the SELECT list of a SQL command now being processed by the specified cursor. The number of SELECT items is set after *sqlcom* or *sqlcex*. For example, if you compiled and executed the SQL command SELECT * FROM EMP and the columns in EMP are ID, NAME, and DEPT, then *sqlnsi* returns 3.

Parameters

cur

The cursor handle associated with this function.

nsi

A pointer to the variable where this function returns the number of SELECT items.

Return value

The return value is zero (0) if the function succeeds and non-zero if it fails.

Example

```
unsigned char nsi;  /* command type */
short ret;          /* return code */

ret = sqlnsi(cur, &nsi);
```

Related functions

sqlcom *sqlexe*

sqloms - Output Message Size

Syntax

```
#include <sql.h>

SQLTAPI   sqloms (cur,outsize)

SQLTCUR   cur;          /* Cursor number */
SQLTDAL   outsize;      /* Output message buffer size */
```

Description

This function sets the size (in bytes) of the output message buffer.

The output message buffer is allocated on both the client computer and on the database server. The client builds an output message in this buffer and sends it to a buffer of the same size on the database server. It is called an *output* message buffer because it is output from the client's point of view.

The most important messages sent from the client to the database server are SQL commands to compile or a row of data to insert.

A large output message buffer does not necessarily increase performance because it only needs to be large enough to hold the largest SQL command to compile, or large enough to hold the largest row of data to insert. A large output message buffer can allocate space unnecessarily on the both the client and the server. Rows are always inserted and sent one row at a time (except in bulk execute mode). A larger output message buffer does *not* reduce network traffic unless bulk execute is on.

SQLBase automatically maintains an output message buffer large enough to hold any SQL command or a row to insert of any length (given available memory). Despite the specified output message buffer size, SQLBase dynamically allocates more space for the output message buffer if needed.

A large output message buffer can help performance when writing LONG VARCHAR columns.

Parameters

cur

The cursor handle associated with this function. Each cursor has one output message buffer associated with it on the client.

outsize

The size of the output message buffer in bytes. Specify zero to use the default message output buffer size in *sql.h* (1000).

Return value

The return value is zero (0) if the function succeeds and non-zero if it fails.

Example

```
main()
{

   SQLTDAL   insize=500;
   SQLTDAL   outsize=500;

static  char   dbnam[] = "demox";/* database name */

/* CONNECT TO THE DATABASE */

cur = 0;
if (rcd = sqlcnc(&cur, dbnam, 0))/* perform connect
                              operation */
   apierr("SQLCNC");

if (rcd = sqlims(cur,insize))
   apierr("SQLIMS");
else
   printf("Input Message Size set to = %d \n", insize);

if (rcd = sqloms(cur,outsize))
   apierr("SQLOMS");
else
   printf("Output Message Size set to = %d \n", outsize);

/* DISCONNECT FROM THE DATABASE */

if (rcd = sqldis(cur))/* failure on disconnect? */
   apierr("SQLDIS");
}
```

Related functions

sqlims

sqlopc - OPen Cursor

Syntax

```
#include <sql.h>

SQLTAPI   sqlopc (curp, hCon, flag)

SQLTCUR  PTR curp;   /* Cursor handle */
SQLTCHN     hCon;    /* Connection handle */
SQLTMOD     flag;    /* future flag */
```

Description

This function opens a new cursor for a specific connection. You can open 256 cursors per connection handle.

Parameters

curp

A pointer to a cursor handle where this function returns a cursor handle

hCon

The newly created cursor is associated with this connection handle.

flag

Future flag. Currently not defined. You can specify zero for this parameter.

Return value

The return value is zero (0) if the function succeeds and non-zero if it fails.

Example

```
if(rcd = sqlopc(&cur, hCon, 0))
{
  printf("Failure on cursor open (rcd = %d)\n", rcd);
  exit(0);
}
else printf("New cursor opened\n");
```

Related functions

sqlcch *sqldis* *sqldch*

sqlprs - Position in Result Set

Syntax

```
#include <sql.h>

SQLTAPI   sqlprs (cur, row)

SQLTCUR   cur;     /* Cursor handle */
SQLTROW   row;     /* Row number wanted */
```

Description

When in result set mode, this function sets a row position in the current result set. A later *sqlfet* returns the row at the position indicated by *row*. The first row is row zero.

In result set mode, once a result set has been created, you can get any row in the result set with the *sqlprs* function without sequentially fetching forward. Once the cursor is positioned, later fetches start from that row.

Parameters

cur

The cursor handle associated with this function.

row

The position (starting with 0) of the row to return in a later *sqlfet*. If the row is not in the result set, this function returns an error.

Return value

The return value is zero (0) if the function succeeds and non-zero if it fails.

Example

```
/* Set the position to a row in the result set of an array of
strings */

char     *rows[100];
```

```
short   ret;
long    i;

i = getrow();          /* routine to get value of i */
if (ret = sqlprs(cur, i))
   {
     ... process error
   }
else
   ...
```

Related functions

sqlcrs *sqlspr* *sqlstr*
sqldrs *sqlsrs* *sqlurs*
sqlrrs

sqlrbf - Roll Back Flag

Syntax

```
#include <sql.h>

SQLTAPI   sqlrbf (cur, rbf)

SQLTCUR        cur;   /* Cursor handle */
SQLTRBF  PTR   rbf;   /* Rollback flag */
```

Description

This function returns the system rollback flag for the current transaction.

A rollback can happen automatically because of a deadlock or system failure.

The rollback flag is *not* set for a user-initiated rollback.

If the rollback flag is set, the work for all cursors that the program has connected to the database has been rolled back and all compiled commands have been destroyed unless cursor-context preservation is on.

Parameters

cur

The cursor handle associated with this function (transaction).

rbf

A pointer to the variable where this function returns the rollback flag. This function returns a 1 if a server-initiated rollback occurred; otherwise, the value is 0.

Return value

The return value is zero (0) if the function succeeds and non-zero if it fails.

Example

```
char    rbkflag;    /* rollback flag */
short   ret;        /* return code */

ret = sqlrbf(cur, &rbkflag);
```

Related functions

sqlerr *sqlfer* *sqlrcd*

sqlrbk - RollBacK

Syntax

```
#include <sql.h>

SQLTAPI  sqlrbk (cur);

SQLTCUR  cur;   /* Cursor handle */
```

Description

This function rolls back the database to the state it was in at the completion of the last implicit or explicit COMMIT. All uncommitted work is undone. This function also establishes the starting point of the next transaction.

This function rolls back all work done since the last commit for *all* cursors that the application has connected to the database.

If cursor-context preservation is off, this function destroys all compiled commands for all cursors that the program has connected to the database. If cursor-context preservation is on, this function does not destroy compiled commands if both of the following are true:

- The application is in Release Locks (RL) isolation level.
- A DDL operation was not performed.

Parameters

cur

The cursor handle associated with this function.

Return value

The return value is zero (0) if the function succeeds and non-zero if it fails.

Example

```
ret = sqlrbk(cur);
```

Related functions

sqlcmt *sqlrbf*

sqlrcd - Return CoDe

Syntax

```
#include <sql.h>


SQLTAPI   sqlrcd (cur, rcd)


SQLTCUR      cur;     /* Cursor handle */
SQLTRCD PTR  rcd;     /* Return code */
```

Description

This function gets the return code for the most-recent SQL/API function. The same code is also returned directly from the function call.

Call the *sqlerr* or *sqlfer* function to get the text associated with the return code. The message text for the return code is in *error.sql*.

Parameters

cur

The cursor handle associated with this function.

rcd

A pointer to the variable where this function returns the return code.

Return value

The return value is zero (0) if the function succeeds and non-zero if it fails.

Example

```
short  rcode;            /* return code */

if (sqlexe(cur))         /* if execute fails */
   sqlrcd (cur, &rcode); /* get the return code */
     ... process error
```

Related functions

sqlerr *sqlfer* *sqlxer*
sqletx

sqlrdb - Restore DataBase

Syntax

```
#include <sql.h>

SQLTAPI   sqlrdb (shandle, dbname, dbnamel, bkpdir, bkpdirl,
                  local, over)

SQLTSVH   shandle;  /* Server handle */
SQLTDAP   dbname;   /* Database name */
SQLTDAL   dbnamel;  /* Database name length */
SQLTFNP   bkpdir;   /* Backup directory */
SQLTFNL   bkpdirl;  /* Backup directory length */
SQLTBOO   local;    /* True: backup directory on local node */
SQLTBOO   over;     /* True: overwrite existing file */
```

Description

This function restores a database from the specified directory. The database is always restored from the file:

database-name.BKP

If this function finds a control file in the restore directory, the function performs the restore operation based on the segmented backups specified in the control file. For details, read the *Database Administrator's Guide*.

You cannot perform a restore while users are connected to the database.

Note: SQLBase supports filenames up to 256 characters including the terminating null character.

Parameters

shandle

The server handle returned by *sqlcsv.*

dbname

A pointer to the string that contains the database name.

dbnamel

The length of the string pointed to by *dbname.* If the string pointed to by *dbname* is null-terminated, specify zero and the system will compute the length.

bkpdir

A pointer to the string that contains the backup directory name.

bkpdirl

The length of the string pointed to by *bkpdir.* If the string pointed to by *bkpdir* is null-terminated, specify zero and the system will compute the length.

local

Source of backup:

0	Backup directory on server.
1	Backup directory on local (client) node.

over

Overwrite indicator:

0	Do not overwrite existing file.
1	Overwrite existing file.

Return value

The return value is zero (0) if the function succeeds and non-zero if it fails.

Example

```
SQLTSVH   shandle;
char*     password;
SQLTDPV   lbmset;
SQLTFNP   bkpdir;
SQLTFNL   bkpdirl;
SQLTRFM   mode=SQLMEOL;
SQLTLNG   lognum;
SQLTBOO   local,over;

static char  dbname1[] = "omed";

password = 0;
bkpdir = "\\BACKUP\\OMED";
bkpdirl = strlen(bkpdir);
printf("value of bkpdir = %s \n",bkpdir);

local=1;
over=1;

/* CONNECT TO SERVER*/

if (rcd = sqlcsv(&shandle,srvname,password))
   apierr("SQLCSV");

/* RESTORE DATABASE */

if (rcd =
sqlrdb(shandle,dbname1,0,bkpdir,bkpdirl,local,over))
   apierr("SQLRDB");
else
   printf("Restored Database \n");
```

```
/* ROLLFORWARD TO END */

sqlrof(shandle,dbname1,0,mode,0,0);

lognum=0;

/*
   The loop below assumes that all log file backups are on
   disk. If a log file backup is not on disk, lognum is set
   to a non-zero value which causes the loop to terminate.
*/
while (lognum == 0)
   {
     /* GET NEXT LOG */
     sqlgnl(shandle,dbname1,0,&lognum);

     /* RESTORE LOG FILES */
     sqlrlf(shandle,dbname1,0,bkpdir,bkpdirl,local,over);
}

/* END ROLLFORWARD */

if (rcd = sqlenr(shandle,dbname1,0))
   apierr("SQLENR");
else
   printf("End Rollforward \n");
```

Related functions

sqlbdb	*sqlcsv*	*sqlrlf*
sqlblf	*sqlenr*	*sqlrof*
sqlbss	*sqlgnl*	*sqlrss*
sqlcrf	*sqlrel*	

sqlrel - RELease current log

Syntax

```
#include <sql.h>

SQLTAPI   sqlrel (cur)
SQLTCUR   cur;    /* Cursor handle */
```

Description

This function releases the current active log file without waiting for it to fill completely.

A new log file is created automatically when the current active log file becomes full (this is called a log rollover). The *sqlrel* function forces a log rollover and is useful when executed just prior to a backup. In releasing the current active log file, SQLBase can back it up (if logbackup is enabled) and delete it. In doing so, the most up-to-date backup is created.

Parameters

cur

The cursor handle associated with this function.

Return value

The return value is zero (0) if the function succeeds and non-zero if it fails.

Example

```
SQLTSVH  shandle;
char*    password;
SQLTDPV  lbmset;
SQLTFNP  bkpdir;
SQLTFNL  bkpdirl;
SQLTRFM  mode=SQLMEOL;
SQLTLNG  lognum;
SQLTBOO  local,over;

static char   dbname1[] = "omed";

password = 0;
```

```
bkpdir = "\\BACKUP\\OMED";
bkpdirl = strlen(bkpdir);

printf("value of bkpdir = %s \n",bkpdir);

local=1;
over=1;

/* CONNECT TO OMED */

if (rcd = sqlcnc(&cur1,dbname1,0))
   apierr("SQLCNC");
else
   printf("Connected to OMED \n");

/* SET LOGBACKUP MODE ON */

lbmset=1;
if (rcd = sqlset(cur1,SQLPLBM,(ubyte1p)&lbmset,0))
   apierr("SQLSET");
else
   printf("Logbackupmode is set to %d \n", lbmset);

/* MAKE BACKUP DIRECTORIES */

system("mkdir \\backup");
system("mkdir \\backup\\omed");

/* CONNECT TO SERVER*/

if (rcd = sqlcsv(&shandle,srvname,password))
   apierr("SQLCSV");

/* BACKUP DATABASE */

if (rcd =
sqlbdb(shandle,dbname1,0,bkpdir,bkpdirl,local,over))
   apierr("SQLBDB");
else
   printf("Backed Up Database \n");

/* RELEASE LOG */

if (rcd = sqlrel(cur1))
   apierr("SQLREL");
else
```

```
    printf("Released Logs \n");

/* BACKUP LOGS */

if (rcd =
sqlblf(shandle,dbname1,0,bkpdir,bkpdirl,local,over))
    apierr("SQLBLF");
else
    printf("Backed Up Logs \n");
```

Related functions

sqlbdb	*sqlenr*	*sqlrlf*
sqlblf	*sqlgnl*	*sqlrof*
sqlbss	*sqlrdb*	*sqlrss*
sqlcrf		

sqlret - RETrieve a stored command/procedure

Syntax

```
#include <sql.h>

SQLTAPI   sqlret (cur, cnp, cnl)

SQLTCUR   cur ;    /* Cursor handle */
SQLTDAP   cnp;     /* Name of stored command/procedure*/
SQLTDAL   cnl ;    /* Length of stored name */
```

Description

This function retrieves a stored SQL command or stored procedure. Once a command/procedure has been retrieved, data can be bound if needed and the command/procedure can be executed.

Once a command or procedure is retrieved, it cannot be destroyed by a commit or rollback.

If another transaction changes the system catalog items that the retrieved command or procedure depends on between the commit and the execute, the execute fails.

You cannot use stored commands while in restriction mode.

Chained Commands

Several stored commands can be retrieved with one *sqlret* and executed with one *sqlexe*. The *sqlret* function allows a list of stored command names separated by commas.

Bind variables can be shared across commands. The same bind variable can be used in more than one command and it only needs to be bound once.

Commands with a CURRENT OF clause; cannot be part of a chained command.

The command type of a chained command is SQLTCHN.

When using UPDATE in a chained command, you can specify the CHECK EXISTS clause to cause an error to be returned if at least one row is *not* updated.

You can use a SELECT command in a chained command with the following restrictions:

- Only one SELECT command can be in a chained command.
- The SELECT command must be the last command in the chain.
- You cannot use bulk execute mode with a chained command that contains a SELECT.

You can check the SQLPCHS parameter with *sqlget* to see if the chained command contains a SELECT.

Parameters

cur

The cursor handle associated with this function.

cnp

A pointer to the string that contains the name of the SQL command or SQL commands to retrieve. If you are not the creator of the stored command, you must qualify the command name with the creator name and a period. For example, if SYSADM created the command:

```
SYSADM.command-name
```

cnl

The length of the string pointed to by *cnp*. If the string pointed to by *cnp* is null-terminated, specify zero and the system will compute the length.

Return value

The return value is zero (0) if the function succeeds and non-zero if it fails.

Examples

To retrieve a stored command:

```
ret = sqlret(cur, "myquery", 0);
```

This example repetitively executes the stored command *the_cmd*:

```
sqlret(cursor, "the_cmd", 0);/* retrieve the command */

for (;;)
   {
   sqlexe(cursor);/* execute the retrieved command */
   ...
   sqlcmt(cursor);/* commit the work */
   ...
}
```

If you have the stored commands: *do_first*, *do_next* and *do_last*, then instead of:

```
sqlret(cursor, "do_first", 0);/* retrieve the first command
*/
sqlexe(cursor);              /* execute it */
sqlret(cursor, "do_next", 0);/* retrieve the next command */
sqlexe(cursor);              /* execute it */
sqlret(cursor, "do_last", 0);/* retrieve the last command */
sqlexe(cursor);              /* execute it */
```

use:

```
/* retrieve all 3 commands */

sqlret(cursor, "do_first, do_next, do_last", 0);
sqlexe(cursor);   /* execute them in sequence */
```

Related functions

sqldst *sqlsto*

sqlrlf - Restore Log Files

Syntax

```
#include <sql.h>

SQLTAPI     sqlrlf(shandle, dbname, dbnamel, bkpdir, bkpdirl,
                   local, over)

SQLTSVH     shandle; /* Server handle */
SQLTDAP     dbname;  /* Database name */
SQLTDAL     dbnamel; /* Database name length */
SQLTFNP     bkpdir;  /* Backup directory */
SQLTFNL     bkpdirl; /* Backup directory length */
SQLTBOO     local;   /* True: backup directory on local node */
SQLTBOO     over;    /* True: overwrite existing file */
```

Description

This function restores as many transaction log files as possible from the specified directory. It continues restoring logs until all the logs from the backup directory that need to be applied to the database have been exhausted.

After each *sqlrlf* function call, SQLBase displays a message indicating the next log file to be restored. If the log file requested is not available, use the *sqlenr* function to terminate media recovery and recover the database using the information obtained up to that point (if possible).

You cannot perform a restore while users are connected.

Note: SQLBase supports filenames up to 256 characters including the terminating null character.

Parameters

shandle

The server handle returned by *sqlcsv*.

dbname

A pointer to the string that contains the database name.

dbnamel

The length of the string pointed to by *dbname*. If the string pointed to by *dbname* is null-terminated, specify zero and the system will compute the length.

bkpdir

A pointer to the string that contains the backup directory name.

bkpdirl

The length of the string pointed to by *bkpdir*. If the string pointed to by *bkpdir* is null-terminated, specify zero and the system will compute the length.

local

Source of backup:

0	Backup directory on server.
1	Backup directory on local (client) node.

over

Overwrite indicator:

0	Do not overwrite existing file.
1	Overwrite existing file.

Return value

The return value is zero (0) if the function succeeds and non-zero if it fails.

Example

```
SQLTSVH       shandle;
char*         password;
SQLTDPV       lbmset;
SQLTFNP       bkpdir;
SQLTFNL       bkpdirl;
SQLTRFM       mode=SQLMEOL;
SQLTLNG       lognum;
SQLTBOO       local,over;

static   char        dbname1[] = "omed";
```

```
password = 0;
bkpdir = "\\BACKUP\\OMED";
bkpdirl = strlen(bkpdir);
printf("value of bkpdir = %s \n",bkpdir);

local=1;
over=1;

/* CONNECT TO SERVER */

if (rcd = sqlcsv(&shandle,srvname,password))
  apierr("SQLCSV");

/* RESTORE DATABASE */

if (rcd = sqlrdb(shandle,dbname1,0,bkpdir,bkpdirl,local,over))
  apierr("SQLRDB");
else
  printf("Restored Database \n");

/* ROLLFORWARD TO END */

sqlrof(shandle,dbname1,0,mode,0,0);

lognum=0;

/*
  The loop below assumes that all log file backups are on disk.
  If a log file backup is not on disk, lognum is set to a non-
  zero value which causes the loop to terminate.
*/
while (lognum == 0)
  {
    /* GET NEXT LOG */
    sqlgnl(shandle,dbname1,0,&lognum);

    /* RESTORE LOG FILES */
    sqlrlf(shandle,dbname1,0,bkpdir,bkpdirl,local,over);
}

/* END ROLLFORWARD */

if (rcd = sqlenr(shandle,dbname1,0))
  apierr("SQLENR");
else
  printf("End Rollforward \n");
```

Related functions

sqlbdb	*sqlcsv*	*sqlrel*
sqlblf	*sqlenr*	*sqlrof*
sqlbss	*sqlgnl*	*sqlrss*
sqlcrf	*sqlrdb*	

sqlrlo - Read LOng

Syntax

```
#include <sql.h>

SQLTAPI   sqlrlo (cur, slc, bfp, bufl, readl)

SQLTCUR       cur;     /* Cursor handle */
SQLTSLC       slc;     /* Column number */
SQLTDAP       bufp;    /* Data buffer */
SQLTDAL       bufl;    /* Length of buffer */
SQLTDAL  PTR  readl;   /* Length of data read */
```

Description

This function reads data stored in a LONG VARCHAR column.

The number of bytes that can be read in one operation can be less than the length of the LONG VARCHAR column. The *sqlrlo* function can be repeated while there is data to read from the LONG VARCHAR column. This allows incremental reading of columns which contain large amounts of data without having to set up equivalent size data buffers.

The *sqlrlo* call is followed by *sqlelo* which ends the read operation for the LONG VARCHAR column.

The maximum length that you can read in one call to *sqlrlo* is 32,767 bytes.

Parameters

cur

The cursor handle associated with this function.

slc

The sequence number (starting with 1) of the column in the SELECT list.

bufp

A pointer to the variable where this function returns the LONG VARCHAR data that was read.

bufl

The length of the variable pointed to by *bufp*.

readl

A pointer to the variable where this function returns the number of bytes read. If this value is zero, it means that the end of data was reached.

Return value

The return value is zero (0) if the function succeeds and non-zero if it fails.

Example

```
static char sqlsel[] = "select text from documents where
                       caseno = 100";

char buffer[BUFSIZ];             /* output buffer */
int len = 1;
short ret;

while ((ret = sqlfet(cur)) == 0) /* till end of fetch */
  {
    while (len)                  /* till no more data */
      {
        if (ret = sqlrlo(cur, 1, buffer, BUFSIZ, &len))
          ... process error
      }
    if (sqlelo (cur))          /* end long for this fetch */
      process error
  }
```

Related functions

sqlelo *sqllsk* *sqlwlo*

sqlrof - ROllForward

Syntax

```
#include <sql.h>

SQLTAPI   sqlrof (shandle, dbname, dbnamel, mode, datetime,
                  datetimel)

SQLTSVH   shandle;  /* Server handle */
SQLTDAP   dbname;   /* Database name */
SQLTDAL   dbnamel;  /* Database name length */
SQLTRFM   mode;     /* Rollforward mode */
SQLTDAP   datetime; /* Date/time value: "mm/dd/yy hh:mm:ss" */
SQLTDAL   datetimel;/* Length of date/time value */
```

Description

This function recovers a database by applying transaction log files to bring a backup up-to-date after a *sqlrdb*.

A restore function cannot be performed while users are connected.

You must have backed up all the database's log files and must apply them in order or the ROLLFORWARD will fail. If you are missing any of the log files, you will not be able to continue rolling forward from the point of the last consecutive log. For example, if you have *1.log*, *2.log*, *4.log* and *5.log*, but *3.log* is missing, you will only be able to recover the work logged up to *2.log*. *4.log* and *5.log* cannot be applied to the database. An unbroken sequence of log files is required by recover a database backup to its most consistent state.

Parameters

shandle

The server handle returned by *sqlcsv*.

dbname

A pointer to the string that contains the database name.

dbnamel

The length of the string pointed to by *dbname*. If the string pointed to by *dbname* is null-terminated, specify zero and the system will compute the length.

mode

The following rollforward modes are defined in *sql.h:*

Constant	Description
SQLMEOL	Rollforward to end of all available logs. This recovers as much work as possible.
SQLMEOB	Rollforward to end of backup. This recovers all committed work up to the point when the database backup was completed.
SQLMTIM	Rollforward to specified time. This recovers a database up to a specific point in time, and in effect rolls back large "chunks" of committed and logged work that you no longer want applied to the database. For example, if data is erroneously entered into the database, you would want to restore the database to the state it was in before the bad data was entered.

datetime

A pointer to the string that specifies the date and time to roll forward to in the format "mm/dd/yy hh:mm:ss".

datetimel

The length of the string pointed to by *datetime.* If the string pointed to by *datetime* is null-terminated, specify zero and the system will compute the length.

Return value

The return value is zero (0) if the function succeeds and non-zero if it fails.

Example

```
SQLTSVH  shandle;
char*    password;
SQLTDPV  lbmset;
SQLTFNP  bkpdir;
SQLTFNL  bkpdirl;
SQLTRFM  mode=SQLMEOL;
SQLTLNG  lognum;
SQLTBOO  local,over;

static   char     dbname1[] = "omed";

password = 0;
```

```
bkpdir = "\\BACKUP\\OMED";
bkpdirl = strlen(bkpdir);
printf("value of bkpdir = %s \n",bkpdir);

local=1;
over=1;

/* CONNECT TO SERVER*/

if (rcd = sqlcsv(&shandle,srvname,password))
  apierr("SQLCSV");

/* RESTORE DATABASE */

if (rcd =
sqlrdb(shandle,dbname1,0,bkpdir,bkpdirl,local,over))
  apierr("SQLRDB");
else
  printf("Restored Database \n");

/* ROLLFORWARD TO END */

sqlrof(shandle,dbname1,0,mode,0,0);

lognum=0;
/*
  The loop below assumes that all log file backups are on
disk.
  If a log file backup is not on disk, lognum is set to a
non-
  zero value which causes the loop to terminate.
*/
while (lognum == 0)
  {
    /* GET NEXT LOG */
    sqlgnl(shandle,dbname1,0,&lognum);

    /* RESTORE LOG FILES */
    sqlrlf(shandle,dbname1,0,bkpdir,bkpdirl,local,over);
  }

/* END ROLLFORWARD */

if (rcd = sqlenr(shandle,dbname1,0))
  apierr("SQLENR");
else
  printf("End Rollforward \n");
```

Related functions

sqlbdb	*sqlcsv*	*sqlrel*
sqlblf	*sqlenr*	*sqlrlf*
sqlbss	*sqlgnl*	*sqlrss*
sqlcrf	*sqlrdb*	

sqlrow - number of ROWs

Syntax

```
# include <sql.h>

SQLTAPI   sqlrow (cur, row)

SQLTCUR          cur;   /* Cursor handle */
SQLTROW   PTR    row;   /* Variable */
```

Description

This function gets the number of rows affected by the most-recent UPDATE, DELETE, INSERT, or *sqlfet*. This function is most useful for counting the number of rows affected by an UPDATE or DELETE.

Parameters

cur

The cursor handle associated with this function.

row

A pointer to a variable where this function returns the number of rows.

Return value

The return value is zero (0) if the function succeeds and non-zero if it fails.

Example

```
long rows;                    /* return code */

ret = sqlrow(cur, &rows))     /* get number of rows */
```

Related functions

sqlgnr *sqlnrr*

sqlrrs - restart Restriction and Result Set modes

Syntax

```
# include <sql.h>

SQLTAPI   sqlrrs; (cur,rsp,rsl)

SQLTCUR   cur;      /* Cursor handle */
SQLTDAP   rsp;      /* Result set name buffer */
SQLTDAL   rsl;      /* Result set name length */
```

Description

This function opens a saved result set and turns on restriction mode and result set mode. The result set must have been saved with the sqlcrs function.

The SELECT command must be recompiled and re-executed before the rows can be fetched with *sqlfet*.

Be cautious about using saved result sets. Internally, a saved result set is a list of row identifiers (ROWIDs) that is stored in the SYSROWIDLISTS system catalog table. A ROWID changes whenever the row is updated. If one of the rows is updated after you have saved and closed a result set, you will get an error if you open the result set later and try to fetch the row.

Parameters

cur

The cursor handle associated with this function.

rsp

A pointer to the string that contains name of the result set.

rsl

The length of the string pointed to by *rsp*. If the string pointed to by *rsp* is null-terminated, specify zero and the system will compute the length.

Return value

The return value is zero (0) if the function succeeds and non-zero if it fails.

Example

```
ret = sqlrrs(cur, "saveres", 0);
```

Related functions

sqlcrs	*sqlscn*	*sqlstr*
sqldrs	*sqlspr*	*sqlurs*
sqlprs	*sqlsrs*	

sqlrsi - Reset Statistical Information

Syntax

```
#include <sql.h>

SQLTAPI  sqlrsi (shandle)

SQLTSVH  shandle;         /* Server handle                */
```

Description

This function resets the statistical information counters in the server. After this function completes the server's statistical counters will be reset.

Parameters

shandle

The server handle returned by *sqlcsv*.

Return value

If this function returns zero, it was successful. If this function returns a non-zero value, it was unsuccessful.

Example

```
sqlrsi (shandle);
```

sqlrss - Restore SnapShot

Syntax

```
#include <sql.h>

SQLTAPI  sqlrss (shandle, dbname, dbnamel, bkpdir, bkpdirl,
                 local, over)

SQLTSVH  shandle;  /* Server handle */
SQLTDAP  dbname;   /* Database name */
SQLTDAL  dbnamel;  /* Database name length */
SQLTFNP  bkpdir;   /* Backup directory */
SQLTFNL  bkpdirl;  /* Backup directory length */
SQLTBOO  local;    /* True: backup directory on local mode */
SQLTBOO  over;     /* True: overwrite existing file */
```

Description

This function restores and recovers a database and its associated log files that were created with the *sqlbss* function. This is the only step necessary to recover the database; you should not follow this call with the *sqlrof* function.

The database is always restored from the file:

database-name.BKP

A restore function cannot be performed while users are connected.

Note: SQLBase supports filenames up to 256 characters including the terminating null character.

Parameters

shandle

The server handle returned by *sqlcsv*.

dbname

A pointer to the string that contains the database name.

dbnamel

The length of the string pointed to by *dbname*. If the string pointed to by *dbname* is null-terminated, specify zero and the system will compute the length.

bkpdir

A pointer to the string that contains the backup directory name.

bkpdirl

The length of the string pointed to by *bkpdir*. If the string pointed to by *bkpdir* is null-terminated, specify zero and the system will compute the length.

local

Source of backup:

0	Backup directory on server.
1	Backup directory on local (client) node.

over

Overwrite indicator:

0	Do not overwrite existing files.
1	Overwrite existing files.

Return value

The return value is zero (0) if the function succeeds and non-zero if it fails.

Example

```
SQLTSVH   shandle;
char*     password;
SQLTDPV   lbmset;
SQLTFNP   bkpdir;
```

```
  SQLTFNL   bkpdirl;
  SQLTRFM   mode=SQLMEOL;
  SQLTLNG   lognum;
  SQLTBOO   local,over;

  static char dbname1[] = "omed";/* default database name */
  static char dbname2[] = "xomed";/* default database name
*/
  static char srvname[] = ""; /* server name */

  password = 0;

  local=1;
  over=1;

/* CONNECT TO SERVER */

if (rcd = sqlcsv(&shandle,srvname,password))
  apierr("SQLCSV");

/* MAKE BACKUP DIRECTORIES */

system("mkdir \\backup\\snapshot");

bkpdir = "\\BACKUP\\SNAPSHOT";
bkpdirl = strlen(bkpdir);

/* BACKUP SNAPSHOT */

if (rcd =
sqlbss(shandle,dbname1,0,bkpdir,bkpdirl,local,over))
  apierr("SQLBSS");
else
  printf("Backup Snapshot Database \n");

/* RESTORE SNAPSHOT */

if (rcd =
sqlrss(shandle,dbname1,0,bkpdir,bkpdirl,local,over))
  apierr("SQLRSS");
else
  printf("Restore Snapshot \n");
```

Related functions

sqlbdb	*sqlcsv*	*sqlrel*
sqlblf	*sqlenr*	*sqlrlf*
sqlbss	*sqlgnl*	*sqlrof*
sqlcrf	*sqlrdb*	

sqlsab - Server ABort database process

Syntax

```
#include <sql.h>

SQLTAPI   sqlsab (shandle, pnum)

SQLTSVH   shandle;    /* Server handle */
SQLTPNM   pnum;       /* Server process number */
```

Description

This function aborts a database server process. You cannot abort a process that the server is currently processing. For example, if a client sends a SELECT statement to a server, the process cannot be aborted until the server begins returning rows.

When a database process is aborted, it ends its network sessions and does a rollback of its transactions.

Parameters

shandle

The server handle returned by *sqlcsv*.

pnum

The process number to abort. Retrieve the database process numbers by calling the *sqlgsi* function.

Return value

The return value is zero (0) if the function succeeds and non-zero if it fails.

Example

```
sqlsab(shandle, processno);
```

Related functions

sqlcdr *sqlgsi* *sqlstm*
sqlcsv *sqlsdn*

sqlscl - Set CLient name

Syntax

```
#include <sql.h>

SQLTAPI   sqlscl (cur/shandle, namp, naml)

SQLTSVH   cur/shandle;/* Database cursor or server handle */
SQLTDAP   namp;       /* Client name */
SQLTDAL   naml;       /* Length of client name */
```

Description

This function assigns a client name to a process.

Parameters

cur/shandle

The cursor handle returned by *sqlcnc* if the parameter is associated with a cursor or database. The server handle returned by *sqlcsv* if the parameter is associated with a server.

namp

A pointer to a string that contains the client name. The maximum length of the client name is 12 characters. Client names are case sensitive.

Specify a null value to de-assign a client name.

naml

The length of the string pointed to by *namp*. If the string pointed to by *namp* is null-terminated, specify zero (0) and the system will compute the length.

Return value

The return value is zero (0) if the function succeeds and non-zero if it fails.

Examples

```
SQLTSVH   shandle;  /* Server handle */
short     rcd;      /* Return code */

if (rcd = sqlscl(shandle, “Clname”, 0))
{
  ...process error
}
```

sqlscn - Set Cursor Name

Syntax

```
# include <sql.h>

SQLTAPI   sqlscn (cur, namp, naml)

SQLTCUR   cur;      /* Cursor handle */
SQLTDAP   namp;     /* Cursor name */
SQLTDAL   naml;     /* Length of cursor name */
```

Description

This function assigns a name to a cursor. A cursor name is used in a SQL command that contains a CURRENT OF clause or ADJUSTING clause.

There is some overhead for fetches when a a cursor name is assigned because the server must keep track of the current cursor position. You can deassign a cursor name by specifying an empty string in the *namp* argument. The server optimizes fetches when a cursor name is not assigned.

Parameters

cur

The cursor handle for the cursor being named.

namp

A pointer to the string that contains the cursor name. The maximum length of the cursor name is 8 characters. Cursor names are case insensitive ("c1" is the same as "C1").

To de-assign a cursor name, pass an empty string.

naml

The length of the string pointed to by *namp*. If the string pointed to by *namp* is null-terminated, specify zero and the system will compute the length.

Return value

The return value is zero (0) if the function succeeds and non-zero if it fails.

Example

```
short ret;         /* return code */

if (ret = sqlscn (cur, "C1", 0))
  {
    ... process error
  }
```

sqlscp - Set Cache Pages

Syntax

```
#include <sql.h>

SQLTAPI  sqlscp(pages)

SQLTNPG  pages;      /* Number of cache pages */
```

Description

This function sets the number of cache pages to use for the next connect.

The size of the cache is set at server startup by the *cache* keyword in the configuration file (*sql.ini*) and it cannot be changed while the server is running. This function only changes the number of cache pages at the next connect and it must be the only cursor connected to a single-user database.

Parameters

pages

The number of cache pages to use in the next connect. If a value of zero is specified, the default number of cache pages is used.

Return value

The return value is zero (0) if the function succeeds and non-zero if it fails.

Example

```
main()
{

SQLTNPG  pages=500;
static char dbnam[]="demox";/* database name */

/* CONNECT TO THE DATABASE */

cur = 0;
if (rcd = sqlcnc(&cur, dbnam, 0))/* perform connect */
  apierr("SQLCNC");
```

```
if (rcd = sqlscp(pages))
  apierr("SQLSCP");

/* DISCONNECT FROM THE DATABASE */

if (rcd = sqldis(cur))/* failure on disconnect? */
  apierr("SQLDIS");
}
```

sqlsdn - ShutDowN database

Syntax

```
#include <sql.h>

SQLTAPI   sqlsdn (dbname, dbnaml)

SQLTDAP   dbname;   /* Database name */
SQLTDAL   dbnaml;   /* Database name length */
```

Description

This function prevents new connections to a database so that it can shut down gracefully.

Only SYSADM can shut down a database.

After this function completes, anyone trying to connect to the database receives a "shutdown in progress" message. All current users remain connected and all current transactions continue.

Once this function completes, the only way to reactivate the database is to call *sqlded* and *sqlind* to deinstall and install the database.

Parameters

dbname

A pointer to the string that contains the connect string, which is the username, database name, and password separated by forward slashes:

```
database/username/password
```

These rules are used:

- The characters before the first forward slash are the database name.

- Any characters after the first forward slash and before the second forward slash are the user name.
- Any characters after the second forward slash are the password.

If the database name, user name, or password is not specified, then the system uses the current default. For example, you can specify a connect string as "//*password*" and use the default database name and username.

The default database name, username:defaultuser name, and password are determined by:

- The *defaultdatabase*, *defaultuser*, and *defaultpassword* keywords in *sql.ini*.
- The default of DEMO/SYSADM/SYSADM.

dbnaml

The length of the string pointed to by *dbname*. If the string pointed to by *dbname* is null-terminated, specify zero and the system will compute the length.

Return value

The return value is zero (0) if the function succeeds and non-zero if it fails.

Example

```
sqlsdn(dbname, dbnaml);
```

Related functions

sqlsab *sqlstm*

sqlsds - ShutDown Server

Syntax

```
#include <sql.h>

SQLTAPI  sqlsdx (shandle, shutdownflag)
SQLTSVH  shandle;          /* Server handle                */
SQLTFLG  shutdownflag;      /* 0 = enable, 1 = shutdown    */
                            /* 2 = reserved, */
                            /* 3 = shutdown w/ exit, */
                            /* 4 = shutdown w/crash */
```

Description

This function prevents new connections to a server. Only SYSADM can shut down a server.

After this function completes, anyone trying to connect to the database receives a "shutdown server in progress" message All current users remain connected and all current transactions continue. SYSADM can reactivate the server with a call with a flag setting of zero.

Parameters

shandle

The server handle returned by *sqlcsv.*

shutdownflag

A flag to shutdown or bring the server back on-line. A value of zero brings the server back on-line. A value of one shuts down the server. A value of three shuts down the server after the users exit. A value of four shuts down the server even if users are on it. If you select a value of four, you must have recovery enabled or your database will be corrupted.

Return value

If this function returns zero, it was successful. If this function returns a non-zero value, it was unsuccessful.

Example

```
sqlsds (shandle, shutdownflag);
```

sqlsdx - ShutDown database eXtended

Syntax

```
#include <sql.h>

SQLTAPI  sqlsdx (dbname, dbnaml, flag)

SQLTDAP  dbname;         /* Database name */
SQLTDAL  dbnaml;         /* Database name length */
SQLTFLG  shutdownflag;/* 0 = enable, 1 = shutdown */
```

Description

This function prevents new connections to a database so that it can shut down gracefully or will bring the database back on-line. Only SYSADM can shut down a database.

After this function completes anyone trying to connect to the database receives a "shutdown in progress" message All current users remain connected and all current transactions continue.

Once this function completes, the SYSADM can reactivate the database with a call to *sqlsdx* with a flag setting of zero.

Parameters

dbname

A pointer to the string that contains the connect string, which is the username, database name, and password separated by forward slashes:

```
database/username/password
```

This parameters has the following guidelines:

- The characters before the first forward slash are the database name.
- Any characters after the first forward slash and before the second forward slash are the username.
- Any characters after the second forward slash are the password.

If the database name, username, or password is not specified, then the system uses the current default. For example, you can specify a connect string as "// password" and use the default database name and username.

The default database name, username, and password are determined by:

- The *defaultdatabase*, *defaultuser*, and *defaultpassword* keywords in *sql.ini*.
- The default of DEMO/SYSADM/SYSADM.

dbnaml

The length of the string pointed to by dbname. If the string pointed to by dbname is null-terminated, specify zero and the system will compute the length.

shutdownflag

A flag to shutdown or bring the database back on-line. A value of one will shutdown the database. A value of zero will bring the database back on-line.

Return value

If this function returns zero, it was successful. If this function returns a non-zero value, it was unsuccessful.

Example

```
sqlsdx (dbname, dbnaml, shutdownflag);
```

sqlset - SET parameter

Syntax

```
#include <sql.h>

SQLTAPI   sqlset (cur, param, pbuf, length)

SQLTCUR   cur;     /* Cursor handle */
SQLTPTY   param;   /* Parameter type */
SQLTDAP   pbuf;    /* Pointer to value */
SQLTDAL   length;  /* Length of value */
```

Description

This function sets a database parameter. The parameter types are shown in the table below.

Note: SQLBase supports filenames up to 256 characters including the terminating null character.

Parameter	Description
SQLPAID	**Adapter Identifier.** This parameter allows the setting of a network adapter identification string. If you call *sqlset* and specify the SQLPAID parameter, it changes the setting of the *adapter_id* keyword in *win.ini*.
SQLPALG	**Process Activity log file name** The file to which SQLBase writes the messages displayed on a multi-user server's Process Activity screen. To turn on logging, specify a pointer to the file name in the *pbuf* parameter. You must have DBA authority to set this parameter.
SQLPANL	**Apply net log.** This parameter disables internal condition checking while a netlog is being applied. This keyword is useful to Centura technical support and development personnel only. If you call *sqlset* and specify the SQLPANL parameter, it changes the setting of the *applynetlog* statement in *sql.ini*. 0 = Off 1 = On
SQLPAPT	**Activate process timing.** When this parameter is On (1), activation times are accumulated for prepares, executes and fetches. Activation times are accumulated at three different levels; system, process and cursor. By default, this parameter is turned off. 0 = Off 1 = On Note that if you are using the *sqlset* function to set the SQLPAPT (activate process timing) parameter, settings for the SQLPCTL (command time limit) timing) and SQLPSTA (statistics for server) parameters can be affected in the following ways: • When you enable a command time limit (by specifying a non-zero value in either the *cmdtimeout* keyword of the server's *sql.ini* file or with the SQLPCTL parameter), SQLPSTA (statistics for server) and SQLPAPT (process timing) are automatically turned on. • If you turn off a command time limit, SQLPSTA (statistics for server) and SQLPAPT (process timing) are automatically turned off, unless you explicitly turned on either parameter after you enabled a command time limit. • If you explicitly turn off either SQLPSTA (statistics for server) or SQLPAPT (process timing), your command time limit (if you enabled on) is turned off and *sql.ini* is updated to reflect *cmdtimeout*=0. It is recommended that if you set a value for any of these three parameters, you should set the same value for the other two. For example, if you set SQLPAPT parameter On (1), you should also set SQLPCTL and SQLPSTA parameters On (1).

Parameter	Description
SQLPAUT	**Autocommit.** Commits the database automatically after each SQL command. By default, this parameter is Off (0) and SQLBase commits the database only when you issue a COMMIT command. Autocommit is cursor-specific. When you set autocommit On (1) for a cursor and then perform an operation with that cursor, SQLBase commits *all* of the transaction's cursors. Performing operations with cursors that do not have autocommit set on does not affect the rest of the transaction's cursors. You cannot have autocommit and bulk execute on simultaneously.
SQLPAWS	**OS averaging window size.** This parameter specifies the number of samples of the CPU % Utilization value to keep for determining the average value. You can specify a window size of 1 to 255. The default setting is one (1). If you call *sqlset* and specify the SQLPAWS parameter, it changes the setting of the *osavgwindow* keyword in *sql.ini*. 0 = Off 1 = 255 units
SQLPBLK	**Bulk execute mode**. Reduces the network traffic for multi-row inserts, deletes, and updates, particularly across a network. In bulk execute mode, data values are buffered so that many rows can be sent to the server in one message. Increasing the size of the output message buffer (with the *sqloms* function) increases the number of operations that can be buffered in one message to the server, thereby improving performance. This setting is cursor specific. If this is On (1), as many operations are buffered in the output message buffer as possible. By default, bulk execute mode is Off (0). Bulk execute mode cannot be on at the same time as the autocommit (SQLPAUT) option.
SQLPCAC	**Size of database cache (in KBytes)**. This parameter sets the size of the cache, which buffers database pages in memory. The larger the cache, the less the disk input and output. In other words, as you increase the value of the cache setting, disk access is reduced. The default cache size for Windows is 500K; for all other platforms, the default is 2M. The minimum is 15K and the maximum is 32767K. If you call *sqlget* and specify the SQLPCAC parameter, it changes the setting of the cache keyword in *sql.ini*, but the new setting does not take effect until SQLBase is restarted.

Parameter	Description
SQLPCCB	**Connect Closure Behavior**. This parameter specifies the connect closure behavior that occurs when you terminate a connection using the *sqldch* function. Valid options are COMMIT, ROLLBACK, or DEFAULT. The default is 0 which means that connect closure behavior is dependent on the database server to which the user is connected. In the case of SQLBase, the DEFAULT setting (0) issues a COMMIT before a connection handle is terminated. To determine the DEFAULT behavior for other servers, read the applicable server documentation. Setting this parameter on (1) instructs the server to issue a COMMIT before a connection handle is terminated, while a setting of (2) issues a ROLLBACK. This option also specifies whether a COMMIT or ROLLBACK is issued before disconnecting to a cursor with an implicit connection using the *sqlcnc* function.
SQLPCCK	**Client check.** This parameter tells SQLBase to send the client a RECEIVE upon receipt of a request. By default, clientcheck is off (0). When SQLBase has finished executing a command, it issues a SEND request to the client with the results of the command. If successful, the server then issues a RECEIVE request and waits to receive another command. Setting this parameter on (1) instructs SQLBase to issue a RECEIVE request before beginning execution of the command, not after it finishes executing the command. Doing so allows SQLBase to detect a situation where the client session is dropped or a cancel request is made during command processing. If you call *sqlset* and specify the SQLPCCK parameter, it changes the setting of the *clientcheck* keyword in *sql.ini*. 0 = Off 1 = On
SQLPCGR	**Contiguous cache pages in cache group.** This parameter specifies the number of contiguous cache pages to allocate. For example if you set cache at 3000, and cachegroup at 30, SQLBase allocates 100 cache groups, consisting of 30 pages each. To set the number of cache pages per group to 50: cachegroup = 50 The default is 30. If you call *sqlset* and specify the SQLPCGR parameter, it changes the setting of the *cachegroup* keyword in *sql.ini*.
SQLPCIS	**Client identifier.** This parameter returns a client identification string. The client identification string will consist of: MAIL_ID\NETWORK_ID\ADAPTER_ID\APP_ID\CLIENT_NAME Each of these identification strings can be returned separately by calling *sqlget* with the appropriate parameter.

Parameter	Description

SQLPCLG	**Commit logging.** When this parameter is On (1), SQLBase causes every database transaction in which data was modified to log a row of data. The data that is logged contains the transaction's Transaction ID and a unique sequence number.
	
	When the COMMIT operation is executed for a transaction that is modified, the data is logged in the system utility table SYSCOMMITORDER. This table lists transactions that operated on the database in the order in which they were committed. Turning the SQLPCLG parameter Off (0) stops commit logging. By default, this parameter is Off (0).
	
	You must have DBA privileges to set the SQLPCLG parameter and to use DDL commands with this parameter.
	
	The following example is a SQL API call to set the commit-order logging to Off (0). This stops the insertion of rows into the SYSCOMMITORDER table during transaction COMMIT operations.
	
	`SQLTDPV value=1;`
	`if (rcd=sqlset(cur, SQLPCLG,(SQLTDAP)&value, 0)`
	`  printf ("Cannot start commit logging \n");`
	
	`Note that commit logging is also supported for replication with Centura Ranger.`

SQLPCLI	**LOAD/UNLOAD Client Value.** The load/unload's ON CLIENT clause value. 0 = Off (file is on the server) 1 = On (file is on the server) This parameter indicates where the load/unload file will reside. Before using this parameter, compile the load/unload statement first.
SQLPCLN	**Client name.** Sets/changes the name of a client computer on the server's display for the duration of the session.
SQLPCMP	**Message compression**. When message compression is On (1), messages sent between a client and the database server or gateway are compressed. This means that messages are shorter, and more rows can be packed into a single message during bulk insert and fetch operations. The compression algorithm collapses repeating characters (run-length encoding). SQLBase performs the compression incrementally as each component of a message is posted. By default, message compression is Off (0) because it incurs a CPU cost on both the client and server machines. This parameter is cursor-specific.
SQLPCSV	**Commit server status.** Indicates whether commit service is enabled for the server. 0 = Off 1 = On

Parameter	Description
SQLPCTI	**Checkpoint time interval.** How often SQLBase performs a recovery checkpoint operation. SQLBase's automatic crash recovery mechanism requires that recovery checkpoints be done. The default checkpoint time interval is one minute. This should yield a crash recovery time of less than a minute. If your site can tole rate a longer crash recovery time, you can increase this interval to up to 30 minutes. Depending on the applications running against the database server, a checkpoint operation can affect performance. If this happens, you can increase the checkpoint interval until you attain the desired performance. You must be the DBA to set this parameter.
SQLPCTL	**Command time limit.** The amount of time (in seconds) to wait for a SELECT, INSERT, UPDATE, or DELETE statement to complete execution. After the specified time has elapsed, SQLBase rolls back the command or transaction. Valid values range from 1 to 43,200 seconds (12 hours maximum), and include 0 (zero) which indicates an infinite time limit. The value of the parameter overrides and changes the *cmdtimeout* keyword in the server's *sql.ini* file. Note that if you are using the *sqlset* function to set the SQLPCTL (command time limit) parameter, parameter settings for the SQLPAPT (activate process timing) and SQLPSTA (statistics for server) parameters can be affected in the following ways: • When you enable a command time limit (by specifying a non-zero value in either the *cmdtimeout* keyword of the server's *sql.ini* file or with the SQLPCTL parameter), SQLPSTA (statistics for server) and SQLPAPT (process timing) are automatically turned on.
	• If you turn off a command time limit, SQLPSTA (statistics for server) and SQLPAPT (process timing) are automatically turned off, unless you explicitly turned on either parameter after you enabled a command time limit. • If you explicitly turn off either SQLPSTA (statistics for server) or SQLPAPT (process timing), your command time limit (if you enabled on) is turned off and *sql.ini* is updated to reflect *cmdtimeout*=0. It is recommended that if you set a value for any of these three parameters, you should set the same value for the other two. For example, if you set SQLPAPT parameter On (1), you should also set SQLPCTL and SQLPSTA parameters On (1).

Parameter	Description
SQLPCTS	**Character set file name.** This parameter identifies a file that specifies different values for the ASCII character set. This is useful for non-English speaking countries where characters in the ASCII character set have different hex values than those same characters in the U.S. ASCII character set. If you call *sqlset* and specify the SQLPCTS parameter, it changes the setting of the *characterset* keyword in *sql.ini*.
SQLPCTY	**Country file section (for example, France).** This parameter tells SQLBase to use the settings in the specified section of the *country.sql* file. SQLBase supports English as the standard language, but it also supports many national languages including those spoken in Europe and Asia. You specify information that enables SQLBase to support another language in the *country.sql* file. If you call *sqlset* and specify the SQLPCTY parameter, it changes the setting of the *country* keyword in *sql.ini*.
SQLPDBD	**DBDIR keyword information.** Sets the drive, path, and database directory name information for the *sql.ini's* DBDIR keyword.
SQLPDDB	**Default database name.** Sets the default database name, overriding the SQLBase default database name of DEMO. Setting this parameter changes the *defaultdatabase* keyword in the section of *sql.ini* called [dbdfault] or [winclient].
SQLPDDR	**Database directory**. The drive, path, and directory name where the database you are connected to resides.
SQLPDIS	**Describe information control.** When (and if) SQLBase sends describe information for a SELECT command to a client. This parameter is cursor-specific. **SQLDELY** (0) means early and is the default value. The server sends describe information after a call to *sqlcom*. Call *sqldes*, *sqldsc*, or *sqlgdi* after *sqlcom* and before *sqlexe*. The server also sends describe information after a call to *sqlcex*. Call *sqldes*, *sqldsc*, or *sqlgdi* after *sqlcex* and before *sqlfet*. **SQLDDLD** (1) means delayed. The server sends describe information after a call to *sqlexe*. Call *sqldes*, *sqldsc*, or *sqlgdi* after *sqlexe*, but before the first *sqlfet*. Calling *sqldes*, *sqldsc*, or *sqlgdi* at any other time is illegal. The server also sends describe information after *sqlcex*. Call *sqldes*, *sqldsc*, or *sqlgdi* after *sqlcex* and before *sqlfet*. Use this setting to reduce message traffic for database servers that do not support compile (*sqlcom*) operations. **SQLDNVR** (2) means never. The server never sends describe information. Any call to *sqldes*, *sqldsc*, or *sqlgdi* is illegal. When you set SQLPDIS to SQLDNVR, *sqlnsi* always returns zero (0). You must hard-code the number of columns in the SELECT command so that the application knows how many times to call *sqlssb.* Use this setting to reduce message traffic when the application always knows the number and type of columns in a SELECT command and never makes calls to *sqldes*, *sqldsc*, or *sqlgdi*.

Parameter	Description
SQLPDMO	Demo version of database. 0 = No 1 = Yes
SQLPDPW	**Default password.** Sets the default password, overriding the SQLBase default password of SYSADM. Setting this parameter changes the *defaultpassword* keyword in the section of *sql.ini* called [dbdfault] or [winclient].
SQLPDTL	**Database command time limit.** This parameter sets the amount of time (in seconds) to wait for a SELECT, INSERT, UPDATE or DELETE command to complete execution. This only includes the time to prepare and execute, not the time to fetch. After the specified time has elapsed, SQLBase rolls back the command. The time limit is valid only for the database requested. A global server command time limit is available by using SQLPCTL. 0 = no time limit 1 = 43,000 secs
SQLPDTR	**Set distributed transaction mode.** If this parameter is on (1), all subsequent CONNECTs and SQL statements will be part of a distributed transaction. Currently, you can have one distributed transaction per session. The default for this parameter is off (0). 0 = Off 1 = On
SQLPDUS	**Default user name.** Sets the default user name, overriding the SQLBase default user name of SYSADM. Setting this parameter changes the *defaultuser* keyword in the section of *sql.ini* called [dbdfault] or [winclient].
SQLPEMT	**Error message tokens.** Sets the error token strings used to customize user errors.

Parameter	Description
SQLPERF	**Error filename.** Specifies a file that contains entries to translate standard SQLBase return codes into user-defined return codes: errorfile=filename The file contains entries for error code translation in the form: sbrcd,udrcd where *sbrcd* is a SQLBase return code found in *error.sql*, and *udrcd* is a user-defined return code. The *sbrcd* value must be a positive integer; the *udrcd* can be a positive or negative integer. There can be no white space between the values or after the comma. The client application converts the *sbrcd* value to the *udrcd* value using the *sqltec* API function. For example, SQLBase returns a value of '1' to indicate an end-of-fetch condition, while DB2 returns a value of '100'. If you want an application to convert all SQLBase return codes of '1' to '100', the entry in the errorfile would look like this: 1,100 When your application calls the *sqltec* function, if the SQLBase return code doesn't exist, SQLBase returns a non-zero return code that means that the translation did not occur. To force translation to occur, you can create a global translation entry using the asterisk (*) character and a generic return code (like '999'). For example, assume an errorfile of SQLBase return codes and corresponding DB2 return codes. For those SQLBase return codes that have no corresponding DB2 return code, you can force the application to return the generic return code '999' with the following entry: *,999 If you call *sqlset* and specify the SQLPERF parameter, it changes the setting of the *errorfile* keyword in *sql.ini*.
SQLPEXP	**Execution plan.** Retrieves the execution plan of the last SQL statement that SQLBase compiled.
SQLPEXS	**Extension size** (in MBytes for partitioned databases, and in KBytes for non-partitioned databases). SQLBase databases grow dynamically as data is added, and expand in units called *extensions*. When a database becomes full, SQLBase must add another extension (or *extent*) to the database. When you set the size for a partitioned database, SQLBase rounds the number up to the next megabyte.

Parameter	Description
SQLPFRS	**Frontend result sets**. SQLBase supports backend result sets, but for those database servers that do not, Centura offers frontend result sets (maintained on the client computer). For SQLBase, SQLPFRS is Off (0). For database servers that do not support backend end result sets, like DB2, SQLPFRS is On (1). You can use *sqlset* to turn off SQLBase's backend result sets, and force result sets to be maintained on the client computer. This is useful when you are using SQLBase to test client applications that will eventually access a database server that does not support backend result sets. This parameter is cursor-specific.
SQLPFT	**Fetchthrough mode**. If fetchthrough is On (1), rows are fetched from the database server even if they are available from the client's input message buffer. Since data could have been updated since you last fetched it (into the input message buffer), using the fetchthrough feature ensures that you see the most up-to-date data. If fetchthrough is Off (0), rows are fetched from the client's input message buffer when possible. In fetchthrough mode, rows are fetched from the backend one at a time; there is no multi-row buffering. Because of this, and the network traffic involved, fetchthrough increases response time. Note for procedures, if you want the On Procedure Fetch section to execute exactly once for every fetch call from the client, returning one row at a time, set fetchthrough mode to On (1) at the client (the default is Off). If the result set you are fetching was created by a SELECT command that included an aggregate function, defined a complex view, or included a DISTINCT, GROUP BY, HAVING, UNION, or ORDER BY clause, then SQLBase creates a virtual table. The rows of this virtual table *cannot* be mapped to the rows in the database. For this reason, if a row in the result set is UPDATEd, when you fetch it, it will *not* reflect the UPDATE even if fetchthrough is On. This parameter is cursor-specific.
SQLPGBC	**Global cursor**. The 5.2/6.0 COBOL SQLPrecompiler uses this parameter. It is listed here for the sake of completeness.
SQLPHFS	**Read-only history file size** (in KBytes). If read-only mode is enabled, setting this parameter limits the size of the read-only history file. The default size is 1 MByte (1000 KBytes).

Parameter	Description
SQLPISO	**Isolation level.** Sets the locking isolation level of all the cursors that the program connects to a database. See the *sqlsil* function for an explanation of the isolation levels: SQLILRR = Repeatable Read SQLILCS = Cursor Stability SQLILRO = Read-Only SQLILRL = Release Locks If you change isolation levels, SQLBase implicitly commits all cursors that the program has connected to the database. In turn, the commit destroys all compiled commands.
SQLPLBM	**Transaction log backup mode**. If media recovery is important to your site, set this parameter On (1) to instruct SQLBase to backup all logs before deleting them. This parameter is database-specific and you should set it On only *once*. The setting will stay active until changed. You do not need to set this each time a database is brought back online. Resetting this option affects whether log files are deleted or saved for archiving. To avoid gaps in your log files, set this parameter once to On. By default, this parameter is not enabled (0) and SQLBase deletes log files as soon as they are not needed to perform transaction rollback or crash recovery. This is done so that log files do not accumulate and fill up the disk. If SQLPLBM is Off (0), you are not able to recover the database if it is damaged by user error or a media failure. This parameter must be On (1) when you back up databases (*sqlbdb*) and log files (*sqlblf*), but does not need to be On when you back up snapshots (*sqlbss*).
SQLPLCK	**Lock limit allocations.** This parameter specifies the maximum number of lock entries to allocate. SQLBase allocates lock entries dynamically (in groups of 100) on an as-needed basis. The default setting is 0, which means that there is no limit on the number of locks allocated; as many lock entries can be allocated as memory permits. If you call *sqlset* and specify the SQLPLCK parameter, it changes the setting of the *locks* keyword in *sql.ini*.
SQLPLDR	**Transaction log directory**. The disk drive and directory that contains the log files. SQLBase creates log files in the home database directory by default, but you can redirect them to a different drive and directory with the *sql.ini*'s *lodgir* keyword.
SQLPLDV	**Load version**. This parameter is not applicable to SQLBase v6.0. If the load file was created by a previous SQLBase release, this parameter allows you to specify what version created the load file. This parameter is cursor-specific.
SQLPLFF	**Support long data with front-end result sets.** Lets (1) you or prevents (0) you from reading and writing long data when using front end result sets with SQLNetwork routers and gateways. This parameter is cursor-specific.

Parameter	Description
SQLPLFS	**Transaction log file size** (in KBytes). The default log file size is 1 MByte (1000 KBytes) and the smallest size is 100,000 bytes. SQLBase rounds up the size of the actual log file by one MByte from the value that you specify. For example, if you set the log file size to 1024 KBtyes, the file will grow to 2048 KBytes. When the current log file grows to the specified size, SQLBase creates a new log file. Specifying a large log file size ensures that log files are not created too frequently, however, if the log file is too large, it wastes disk space.
SQLPLOC	Local/remote database server. Specifies whether the database being accessed is local or remote. 0 = Remote 1 = Local engine
SQLPLRD	**Local result set directory**. If the database server does not support backend result sets, this parameter sets the name of the directory on the client computer that contains the frontend result set file. By default, the current working directory holds the result set.
SQLPMID	**E-Mail Identifier.** This parameter allows the setting of an E-Mail identification string. If you call *sqlset* and specify the SQLPMID parameter, it changes the setting of the *mail_id* keyword in *win.ini*.
SQLPNCK	**Check network transmission errors.** This parameter enables and disables a checksum feature that detects transmission errors between the client and the server. To use this feature, both the client and the server must enable netcheck. The default is off (0). If you call *sqlset* and specify the SQLPNCK parameter, it changes the setting of the *netcheck* keyword *sql.ini*. 0 = Off 1 = On
SQLPNCT	**Netcheck algorithm.** This parameter specifies the algorithm SQLBase uses when netcheck is enabled. Configure this keyword only when you enable netcheck By default, checksum(0) is enabled. To switch to CRC/16: netchecktype = 1 If you call *sqlset* and specify the SQLPNCT parameter, it changes the setting of the *netchecktype* statement in *sql.ini*. 0 = Checksum 1 = CRC/16
SQLPNDB	Mark as brand new database. Used in conjunction with COUNTRY.DBS. 0 = False 1 = True

Parameter	Description
SQLPNID	**Network identifier.** This parameter allows the setting of an Network identification string. If you call *sqlset* and specify the SQLPNID parameter, it changes the setting of the *network_id* keyword in *win.ini*.
SQLPNIE	**Null indicator error.** Controls what *sqlfet* returns in *sqlssb*'s *pfc* parameter when the value is null: 0 = *sqlfet* returns zero (default). 1 = *sqlfet* returns FETRNUL (7). Note that to use the FETRNUL indicator in *sqlssb*'s PFC parameter, you must set the SQLPNIE parameter to 1. Setting SQLPNIE affects all the cursors connected by the application that set it; it does not affect other applications.
SQLPNLB	**Next transaction log file to back up**. Specify the number (integer) of the next log file to back up. If SQLPLBM is On, you need to set this parameter after doing an *offline* backup. Setting this parameter tells SQLBase that you did an offline backup and that there are now log files eligible for deletion. For example, if you back up *mydbs.dbs*, *1.log*, *2.log*, and *3.log* offline, you should set SQLPNLB to 4. SQLBase then knows that *1.log*, *2.log*, and *3.log* can be deleted, while *4.log* and all other logs that follow need to be saved for archiving.
SQLPNLG	**Net log file.** This parameter invokes a diagnostic server utility that records database messages to a specified log file. This utility logs all messages that pass between a server and clients on a network. Do not use the netlog utility unless instructed to do by Centura's Technical Support staff. By default, the netlog utility is off. If you call *sqlset* and specify the SQLPNLG parameter, it changes the setting of the *netlog* keyword in *sql.ini*.
SQLPNPB	**Do not prebuild result sets**. If SQLPNPB is Off (0), SQLBase prebuilds result sets. The database server releases shared locks before returning control to the client. The client application must wait until the entire result set is built before it can fetch the first row. If SQLPNPB is On (1), SQLBase does *not* prebuild result sets if the client is in result set mode and Release Locks (RL) isolation level. The advantage of having SQLPNPB on is that the client does not have to wait very long before fetching the first row. SQLBase builds the result set as the client fetches data. By default, SQLPNPB is On (1) for single-user engines and Off (0) for multi-user servers. This parameter is cursor-specific.

Parameter	Description
SQLPNPF	**Net prefix character.** This parameter allows SQLBase to distinguish a database on one server from an identically-named database on another server and to circumvent the network's requirement of name uniqueness. You can specify a value with which SQLBase prefaces each database name on the server. If you have a netprefix entry in the server's *sql.ini* file, all clients connecting to databases on that server must specify the same netprefix value in their configuration files. If you call *sqlset* and specify the SQLPNPF parameter, it changes the setting of the *netprefix* keyword in *sql.ini*.
SQLPOBL	**Optimized bulk execute mode.** This is similar to, but even faster than, bulk execute mode (SQLPBLK) which reduces the network traffic for multi-row inserts, deletes, and updates. The difference is that if an error occurs, SQLBase rolls back the entire transaction. In bulk execute mode, data values are buffered so that many rows can be sent to the server in one message. Increasing the size of the output message buffer (with the *sqloms* function) increases the number of operations that can be buffered in one message to the server, thereby improving performance. This setting is cursor specific. If this is On (1), as many operations are buffered in the output message buffer as possible. By default, bulk execute mode is Off (0). Bulk execute mode cannot be on at the same time as the autocommit (SQLPAUT) option.
SQLPOFF	**Optimize first fetch.** This parameter lets you set the optimization mode for a particular cursor. All queries that are compiled or stored in this cursor inherit the optimization mode in effect. 0 = optimizes the time it takes to return the entire result set. 1 = optimize the time it takes to fetch the first row of the result set. If you call *sqlget* and specify the SQLPOFF parameter, it overrides the setting for *optimizefirstfetch* in *sql.ini* for the particular cursor. If you do not specify this parameter, the optimization mode for the cursor is determined by the setting of the *optimizefirstfetch* value of the server. If *sql.ini* does not have an *optimizefirstfetch* keyword, the default setting is 0 (optimize the time it takes to return the entire result set). Note that a command that earlier stored, retrieved, and executed will continue to use the execution plan with which it was compiled.

Parameter	Description

SQLPOOJ	**Oracle outer join.** This parameter enables and disables Oracle-style join processing. Oracle's outer join implementation differs from the ANSI and industry standard implementation. To paraphrase the ANSI standard, the correct semantics of an outer join are to display all the rows of one table that meet the specified constraints on that table, regardless of the constraints on the other table. For example, assume two tables (A and B) with the following rows:
	`Table A (a int)` `Table B (b int)`
	`1` `1`
	`2` `2`
	`3` `3`
	`4`
	`5`
	If you issue the following SQL command:
	SELECT a, b
	FROM A, B
	WHERE A.a = B.b (+)
	AND B.b IS NULL;
	the ANSI result is:
	`Table A (a int)` `Table B (b int)`
	`1`
	`2`
	`3`
	`4`
	`5`
	Assuming the same two tables and the same SQL command, the correct result for Oracle is:
	`Table A (a int)` `Table B (b int)`
	`4`
	`5`
	If you set *oracleouterjoin*=1; you receive the Oracle result shown directly above. If you call *sqlset* and specify the SQLPOOJ parameter, it changes the setting of the *oracleouterjoin* keyword in *sql.ini*.
	0 = Off
	1 = On

Parameter	Description
SQLPOPL	**Optimizer techniques**. Determines the optimizing techniques that SQLBase uses for all clients that connect to a server. This parameter lets you fall back on old optimizing techniques after upgrading to newer versions of SQLBase. If you discover better performance of a query when this parameter is set to 1, you should report it to Centura's Technical Support team. Be sure not to include compilation time in the comparison of settings 1 and 2. 1 = SQLBase uses old optimizing techniques. 2 = SQLBase uses current optimizing techniques (default). This parameter setting overrides the value of the *optimizerlevel* keyword in *sql.ini*.
SQLPOSR	**OS statistics sample rate.** This parameter specifies the frequency at which operating system statistics (CPU % Utilization) are gathered. You can specify a setting of 0 to 255 seconds. The default setting is zero (0), which disables the gathering of CPU statistics. If you call *sqlset* and specify the SQLPOSR parameter, it changes the setting of the *ossamplerate* keyword in *sql.ini*. 0 = Off 1 = 255 secs
SQLPPAR	**Partitioned database.** Indicates if the database is partitioned. 0 = No 1 = Yes
SQLPPCX	**Cursor context preservation**. If cursor context preservation is On (1), SQLBase prevents a COMMIT from destroying an active result set, thereby enabling an application to maintain its position after a COMMIT, INSERT, or UPDATE. Locks are kept on pages required to maintain the fetch position. Be aware that this can block other applications trying to access the same data. Also, locks can prevent other applications from doing DDL operations. By default, cursor context preservation is Off (0). A COMMIT destroys a cursor's result set or compiled command. SQLBase does *not* preserve cursor context after an isolation level change or after a system-initiated ROLLBACK, such as a deadlock, timeout, etc. SQLBase *does* preserve cursor context after a user-initiated ROLLBACK if both of the following are true: 1) The application is in Release Locks (RL) isolation level. 2) A data definition language (DDL) statement was not performed. If the result set you are fetching was created by a SELECT command that included an aggregate function, defined a complex view, or included a DISTINCT, GROUP BY, HAVING, UNION, or ORDER BY clause, then SQLBase creates a virtual table. The rows of this virtual table *cannot* be mapped to the rows in the database. For this reason, if a row in the result set is UPDATEd, when you fetch it, it will *not* reflect the UPDATE even if fetchthrough is On. This parameter is cursor-specific.

Parameter	Description
SQLPPDB	**Access to partitioned databases.** Enables and disables access to partitioned databases. When you set this parameter to TRUE, you enable user access to partitioned databases; when FALSE (0), you disable user access, allowing you to restore MAIN.DBS.
SQLPPLF	**Preallocate transaction log files**. By default, this parameter is Off (0) and a log files grows in increments of 10% of its current size. This uses space conservatively, but can lead to a fragmented log file which can affect performance. If this parameter is On (1), log files are created full size (preallocated).
SQLPPLV	**Level of Process Activity display**. Sets the level (0 - 4) of detail of the messages on a multi-user server's Process Activity display. You must have DBA authority to set this parameter.
SQLPROD	**Read-only database.** Makes a database accessible on a read-only basis. SQLBase disallows you from executing data definition language (DDL) or data manipulation language (DML) commands. Before you can turn on this feature, you must set *tempdir* in the *autoexec.bat* or the *sql.ini* file to point to the directory where SQLBase should store its temporary files. The temporary files are stored in a subdirectory of the directory pointed to by *tempdir*. The name of this subdirectory is the same as the database name. You must be the only connected user to set this parameter. If this parameter is On (1), SQLBase disables both the Read-Only isolation level and transaction logging.
SQLPROM	**Read-only transaction mode**. Allows users connecting to any of the databases on the server to use the RO (read-only) isolation level. The RO isolation level allows users with a consistent view of data during their session. If this parameter is On (1), SQLBase allows users to use the RO isolation level. All future server sessions for all databases on the server are started with RO transactions enabled; and SQLBase maintains a read-only history file that contains multiple copies of modified database pages; when users try to access pages changed by other users, SQLBase retries a copy of the original page from the history file. Read-only transactions can affect performance, so, by default, this parameter is Off (0), prohibiting users from setting the RO isolation level. If you call *sqlset* and specify the SQLPROM parameter, it changes the setting of the *readonly* keyword in *sql.ini*, but the new setting does not take effect until you restart SQLBase. 0 = Off 1 = On NOTE: To turn on RO transaction mode for a single database and the current session, use SQLPROT.

Parameter	Description
SQLPROT	**Read-only transaction mode**.If this parameter is On (SQLVON), SQLBase allows applications to set the RO (read-only) isolation level on for a single database and the current server session. SQLBase maintains a read-only history file that contains one or more copies of pages that have been modified. Read-only transactions can affect performance so, by default, this parameter is Off (SQLVOFF), prohibiting use of the RO isolation level. If this parameter is set to the default (SQLVDFL), SQLBase uses the *readonly* keyword setting in the *sql.ini* file to determine whether to allow read-only transactions. If you do not provide a value for this keyword, SQLBase uses the internal default (SQLVOFF). You can turn on the RO isolation level only for multi-user versions of SQLBase. If you set this feature on with a *sqlset* call, it applies to the current database. If you set this feature on by modifying the *readonly* keyword setting in *sql.ini*, the setting applies to all databases on the server. You can also turn on RO isolation level for all databases on the server by using the SQLPROM parameter.
SQLPRTO	**Rollback on lock timeout**. This parameter is On (1) by default and SQLBase rolls back an entire transaction when there is a lock timeout. If this parameter is Off (0), SQLBase rolls back only the current command. This parameter is cursor-specific.
SQLPSIL	**Silent mode.** This parameter turns the display for multi-user server on (0) and off (1). To set the display of the server screens off: silent = 1 By default, multi-user server displays are on(0). If you call *sqlset* and specify the SQLPSIL parameter, it changes the setting of the *silent* statement in *sql.ini*. 0 = On 1 = Off

Parameter	Description
SQLPSTA	**Statistics for server.** This parameter collects the following timer and counter information: Timers: Compile. Execute. Fetch. Counters: Physical disk writes. Physical disk reads. Virtual disk writes Virtual disk reads. Total number of disconnects. Total number of connects. Hash joins - number of joins that have occurred. Sorts - number of sorts that have been performed Deadlocks - number of deadlocks that have occurred. Process switches - number of process switches. Full table scan - number of times a full table scan occurred. Index use - number of times an index has been used. Transactions - number of completed transactions. Command type executed - one counter for each command type. The default for this parameter is off (0). 0 = off 1 = on Note that if you are using the sqlset function to set the SQLPCTL (command time limit) parameter, it affects the setting of the SQLPSTA (statistics for server), as well as the SQLPAPT (activate process timing) parameter. The following behavior occurs: • When you enable a command time limit (by specifying a non-zero value in either the CMDTIMEOUT keyword of the server's *sql.ini* file or with the SQLPCTL parameter), statistics for server and process timing are automatically turned on. • If you turn off a command time limit, statistics gathering and process timing are automatically turned off, unless you explicitly turned on either statistics gathering or process timing after you enabled a command time limit. • If you explicitly turn off either statistics for server or process timing, your command time limit (if you enabled on) is turned off and *sql.ini* is updated to reflect CMDTIMEOUT=0. It is recommended that if you set a value for any of these three parameters, you should set the same value for the other two. For example, if you set SQLPCTL parameter to On (1), you should also set SQLPSTA and SQLPAPT parameters to On (1).

Parameter	Description
SQLPSTC	**Sort cache size in pages.** This parameter specifies the number of cache pages to use for sorting. Sorting is done when you specify a DISTINCT, ORDER BY, GROUP BY, or CREATE INDEX clause, or when SQLBase creates a temporary table for join purposes. The default is 64, as is the maximum. If you call *sqlset* and specify the SQLPSTC parameter, it changes the setting of the *sortcache* keyword in *sql.ini*.
SQLPSVN	**Name of server.** This parameter shows the name of the server you are connected to. Setting of this parameter will only change the setting in the *sql.ini*. To activate the new setting, the server must be restarted. You must have DBA authority to set this parameter.
SQLPTCO	**Time colon only.** This parameter configures SQLBase to recognize when a delimiter other than a colon(:) is being used to separate the hours, minutes, and seconds portions of a time value. The default is off (0). If you call *sqlset* and specify the SQLPTCO parameter, it changes the setting of the *timecolononly* keyword in *sql.ini*. 0 = No 1 = Yes
SQLPTHM	**Thread mode.** This parameter specifies whether to use native threads or SQLBase threads. A value of 1 indicates SQLBase threads and a value of 2 indicates native threads. Note for Windows 95, SQLBase now uses Windows 95 native threads only. By default, *threadmode* is 1, except on Windows 95 where the default is 2. On Netware platforms, if you are running in Ring 0, Centura recommends using SQLBase threads which invoke stack switching. This should yield better performance. Novell disallows stack switching in Ring 3, so be sure to set threadmode to 2 when in Ring 3. If you call *sqlset* and specify the SQLPTHM parameter, it changes the setting of the *threadmode* keyword in *sql.ini*.
SQLPTMO	**Client request time out.** This parameter specifies the time period that the server waits for a client to make a request. If the client does not make a request within the specified period, SQLBase rolls back the client session, processes, and transactions. The time-out clock restarts each time the client makes a request. The time-out value is 0 (infinite by default, and the maximum value is 200 minutes. If you call *sqlset* and specify the SQLPTMO parameter, it changes the setting of the *timeout* statement in *sql.ini*.
SQLPTMS	**Timestamp.** If set to TRUE (1), SQLBase timestamps the messages on a multi-user server's Process Activity display; if FALSE (0), SQLBase does not. You must have DBA authority to set this parameter.

Parameter	Description
SQLPTMZ	**Time zone.** This parameter sets the value of SYSTIMEZONE, a SQLBase keyword that returns the time zone as an interval of Greenwich Mean Time. SYSTIMEZONE uses the expression (SYSTIME - TIMEZONE) to return the current time in Greenwich Mean Time. By default, timezone is 0. If you call *sqlset* and specify the SQLPTMZ parameter, it changes the setting of the *timezone* keyword in sql.ini.
SQLPTPD	**Temp directory.** This parameter specifies the directory where SQLBase places temporary files. In the course of processing, SQLBase can create several kinds of temporary files: sort files, read-only history files, and general-use files. To specify d:\tmp as the temporary directory: tempdir = d:\tmp You must set tempdir for read-only databases. If you call *sqlset* and specify the SQLPTPD parameter, it changes the setting of the *tempdir* keyword in *sql.ini*.
SQLPTRC	**Trace stored procedures.** Enables or disables statement tracing for procedures. 0 = Off 1 = On
SQPTRF	**Tracefile name.** Directs statement output to a file on the server. If you do not set this parameter to a file name, the statement output goes to the server's Process Activity screen.
SQLPTSL	**Transaction span limit.** The number of log files that SQLBase allows an active transaction to span. When SQLBase creates a new log file, it checks this limit for all active transactions and rolls back any transaction that violates the limit. By default, the transaction span limit is set to zero (0) which disables the limit checking. Long running transactions can pin down disk log files that otherwise could be deleted. You can limit the amount of logs pinned down by active transactions by specifying the transaction span limit. SQLBase rolls back long running transactions that exceed the limit, thereby freeing pinned log files and deleting them (or backing them up and deleting them if log backup is enabled).
SQLPTSS	**Thread stack size.** This parameter specifies the stack size. By default, threadstacksize is 10 kilobytes and the minimum value is 8192 bytes. You should not decrease the default value. Running complex queries when threadstacksize is set to 8192 can result in a stack overflow error. If you receive stack overflow errors, increase the value of threadstacksize by 512 bytes at a time. If you call *sqlset* and specify the SQLPTSS parameter, it changes the setting of the *threadstacksize* keyword in *sql.ini*.

Parameter	Description
SQLPUID	**Application identifier.** This parameter allows the setting of an user identification string. If you call *sqlset* and specify the SQLPUID parameter, it changes the setting of the *app_id* keyword *win.ini*.
SQLPUSR	**Number of users.** This parameter specifies the maximum number of client applications that can connect to the server simultaneously. This means, for example, that a server configured with users=5 could support five clients running one application each, or one client running five applications, or two clients with one running two applications and the other running three applications, and so on. The default value of users is 128, and the maximum is 800. If you call *sqlset* and specify the SQLPUSR parameter, it changes the setting of the *users* keyword in *sql.ini*.
SQLPWKA	**Work space allocation unit.** This parameter specifies the basic allocation unit of a work space. For example, if a SQL command requires 5000 bytes and the default value of 1000 is in effect, SQLBase makes 5 memory allocation requests to the operating system (5 * 100 = 5000). The default is 1000 bytes. If you call *sqlset* and specify the SQLPWKA parameter, it changes the setting of the *workalloc* keyword in *sql.ini*.
SQLPWKL	**Maximum work space limit.** This parameter specifies a maximum memory limitation for SQL commands. For example, if you specify: worklimit = 4000 SQLBase cannot execute SQL commands requiring more than 4000 bytes of memory. The default is NULL, meaning that no memory limitation exists. If you call *sqlset* and specify the SQLPWKL parameter, it changes the setting of the *worklimit* statement in *sql.ini*.
SQLPWTO	**Lock wait timeout.** Specify the number of seconds for SQLBase to wait for a database lock to be acquired. After the specified time has elapsed, SQLBase rolls back the command or transaction. The default is 300 seconds. Valid timeout values are: 1 - 1800 inclusive (1 second to 30 minutes) 0 = never wait; return error immediately -1 = wait forever This parameter is only relevant for multi-user servers and it is transaction-specific. You can also set the lock wait timeout value with the *sqltio* function.

Parameters

cur

A cursor handle if the parameter is associated with a cursor. A value of 'No' in the following table indicates that a cursor handle is not required. In this case, specify a zero (0).

parm

The name of the parameter to set. The parameter types are defined in *sql.h* and are shown in the following table.

pbuf

A pointer to the variable that contains the parameter setting. The data type and size of the variable depends on the parameter as defined in the following table.

length

The length of the value pointed to by *pbuf*. The following table shows whether a length needs to be specified for a parameter.

For strings, even if a length is needed, you can specify zero to indicate that the value pointed to by *pbuf* is null-terminated and the system will compute the length.

Specify a length of zero for null-terminated string parameters such as SQLPISO.

Parameter Types

The following table lists:

- *parm* - the parameter type.
- *cur* - whether the parameter requires a cursor handle.
- *pbuf* - the size of the variable pointed to by *pbuf*.
- *len* - whether you need to specify a length for the parameter.

The parameter types and *pbuf* types and sizes are defined in *sql.h*.

parm	cur	pbuf	len
SQLPALG	Yes	SQLMFNL	Yes
SQLPAPT	Yes	SQLTDPV	No
SQLPAUT	Yes	SQLTDPV	No

parm	cur	pbuf	len
SQLPBLK	Yes	SQLTDPV	No
SQLPCAC	Yes	SQLTDPV	No
SQLPCLG	Yes	SQLTDPV	No
SQLPCLN	Yes	SQLMFNL	Yes
SQLPCMP	Yes	SQLTDPV	No
SQLPCSV	No	SQLTDPV	No
SQLPCTI	Yes	SQLTDPV	No
SQLPCTL	Yes	SQLTDPV	No
SQLPCTS	No	SQLMNPL	Yes
SQLPDBD	Yes	SQLMFNL	Yes
SQLPDDB	No	Character field of size SQLMDNM + 1	Yes
SQLPDIS	Yes	SQLTDPV	No
SQLPDMO	Yes	SQLTDPV	No
SQLPDPW	No	Character field of size SQLMSID + 1	Yes
SQLPDTR	No	SQLTDPV	No
SQLPDUS	No	Character field of size SQLMSID + 1	Yes
SQLPEMT	Yes	SQLMXER	Yes
SQLPEXP	Yes	SQLMFNL	Yes
SQLPEXS	No	SQLMFNL	No
SQLPFRS	Yes	SQLTDPV	No
SQLPFT	Yes	SQLTDPV	No
SQLPGBC	Yes	Pass a null pointer (such as SQLNPTR)	No
SQLPHFS	Yes	SQLTDPV	No
SQLPISO	Yes	SQLMFNL	Yes

parm	cur	pbuf	len
SQLPLBM	Yes	SQLTDPV	No
SQLPLDV	Yes	SQLMFNL	Yes
SQLPLFF	Yes	SQLTDPV	No
SQLPLFS	Yes	SQLTDPV	No
SQLPLOC	Yes	SQLTDPV	No
SQLPOFF	Yes	SQLTDPV	No
SQLPLRD	Yes	SQLMFNL	Yes
SQLPNDB	Yes	SQLTDPV	No
SQLPNIE	No	SQLTDPV	No
SQLPNLB	Yes	SQLTDPV	No
SQLPNPB	Yes	SQLTDPV	No
SQLPOPL	Yes	SQLTDPV	No
SQLPPAR	Yes	SQLTDPV	No
SQLPPCX	Yes	SQLTDPV	No
SQLPPDB	No	SQLTDPV	No
SQLPPLF	Yes	SQLTDPV	No
SQLPPLV	Yes	SQLTDPV	No
SQLPROD	Yes	SQLTDPV	No
SQLPROM	Yes	SQLTDPV	No
SQLPROT	Yes	SQLTDPV	No
SQLPRTO	Yes	SQLTDPV	No
SQLPSTA	Yes	SQLTDPV	No
SQLPTMS	Yes	SQLTDPV	No
SQLPTRC	Yes	SQLTDPV	No
SQLPTRF	Yes	Character field of size SQLMFNL + 1	Yes

parm	cur	pbuf	len
SQLPTSL	Yes	SQLTDPV	No
SQLPWTO	Yes	SQLTDPV	No

Return value

The return value is zero (0) if the function succeeds and non-zero if it fails.

Examples

```
#include <sql.h>
char dbn [SQLMDNM + 1];/* database name buffer */
SQLTRCDrcd;      /* return code */

if (rcd = sqlset(0, SQLPDDB, dbn, 0))/* set dbname */
  printf("Failure Setting Database Name (rcd = %d)\n", rcd);
```

The function below sets the log file size to 500 kilobytes. When the active log grows to this size, it is closed and a new log is started.

```
SQLTDPVsize=500
sqlset(cur, SQLPLFS, (SQLTDAP)&size, 0);
```

The function below sets the log backup mode to OFF. This means that the user does not want to backup log files and wants to delete log files from disk as soon as they are not needed for crash recovery.

```
lbmset=0;
if (rcd = sqlset(cur1,SQLPLBM,(SQLTDAP)&lbmset,0))
  apierr("SQLSET");
else
  printf("Logbackupmode is set to %d \n", lbmset);
```

Related functions

sqlget

sqlsil - Set Isolation Level

Syntax

```
#include <sql.h>

SQLTAPI   sqlsil (cur, isolation)

SQLTCUR   cur;        /* Cursor handle */
SQLTILV   isolation;  /* Isolation level */
```

Description

This function sets the isolation level at which the application will operate in a multi-user environment.

The isolation level controls the effect that changes made by one user have on another user accessing the same tables. SQLBase supports these isolation levels:

- Read Repeatability (RR)
- Cursor Stability (CS)
- Read Only (RO)
- Release Locks (RL)

Choose an isolation level based on the application's requirements for consistency and concurrency.

The isolation level you set applies to *all* the cursors that an application connects to the same database.

If you change isolation levels, it causes an implicit commit for *all* cursors that the program has connected to the database. In turn, an implicit commit destroys all compiled commands for the database. However, calling *sqlsil* and specifying an isolation level that is the same as the current isolation level does *not* cause an implicit commit.

Isolation Levels and the Input Message Buffer

Each isolation level uses the input message buffer differently. This buffer is allocated on the client computer and the server computer. The database server builds a message and sends it to the input message buffer on the client computer. This buffer is considered "input" with respect to the client computer.

There is one input message buffer per connected cursor on the client computer. On the server, there is one input message buffer that is the size of the largest input message buffer on the client computer.

The input message buffer receives data requested by the client that has been fetched with *sqlfet* and sent by the server.

Any row in the input message buffer may have a shared lock on it depending on the isolation level setting, preventing other users from changing that row.

The table below summarizes how page locking and the input message buffer are affected by each isolation level.

Isolation level	Input message buffer	Shared lock duration and scope
RR	Fills the input message buffer.	Maintained for duration of transaction and more than one page may be locked.
CS	One row sent to input message buffer.	Maintained for duration of transaction, but only the current page is locked.
RO	Fills the input message buffer.	None.
RL	Fills the input message buffer.	All shared locks are released by the time control returns to the client.

Read Repeatability (RR)

This isolation level means that all pages which you access stay locked for other users until you COMMIT your transaction. If the same data is read again during the transaction, those rows would not have changed. This guarantees that the data accessed is consistent for the life of the transaction. Identical SELECT commands will return identical rows since data cannot be changed by other users during the transaction. In this situation, other users must wait for your COMMIT command.

Read Repeatability is the default isolation level.

The Read Repeatability isolation level fills the input message buffer with rows. All shared locks remain regardless of the size of the input message buffer until the application issues COMMIT or ROLLBACK.

Cursor Stability (CS)

This isolation level means that only the page you are processing at the moment is locked to other users. A shared lock is placed on a page for as long as the cursor is on that page (while the cursor is stable). Exclusive locks and shared locks are held until a COMMIT. Other pages you accessed during the transaction are available to other users and they do not have to wait for your COMMIT.

Data that has been read during a transaction may be changed by other users when the cursor moves to a new page.

Only one row is sent to the input message buffer under the Cursor Stability isolation level despite the size of the buffer. In other words, each *sqlfet* causes the client and server to exchange messages across the network.

Use Cursor Stability when you want to update one row at a time using the CURRENT OF cursor clause. When the row is fetched to the client input message buffer, its page will have a shared lock, which means that no other transaction will be able to update it.

Read-Only (RO)

This isolation level places no locks on the database and can only be used for reading data. DDL and DML operations are not allowed while in read-only isolation. This isolation level provides a view of the data as it existed when the transaction began. If you request a page that is locked by another concurrent transaction, SQLBase provides an older copy of the page from the read-only history file. The read-only history file maintains multiple copies of database pages that have been changed.

This is an appropriate isolation level if the data wanted must be consistent but not necessarily current. This isolation level also guarantees maximum concurrency.

Read-only transactions may affect performance, so they are disabled by default. Read-only transactions can be turned on by calling the *sqlset* function with the SQLPROM or SQLPROT parameters, or by specifying the *readonly* keyword in *sql.ini*. If you set the read-only isolation level with *sqlset* and the SQLPROM parameter, or with the *readonly* keyword in *sql.ini*, all future server sessions and all databases on the server are started with read-only transactions enabled. If you call *sqlset* with the SQLPROT parameter, read-only isolation level is set only for a single database and the current server session. Read the section on the *sqlset* function for details.

This isolation level fills the input message buffer with rows.

The Read-Only isolation level is disabled when the *SQLPROD* parameter is on.

Release Locks (RL)

Under Cursor Stability, when a reader moves off a database page, the shared lock acquired when the page was read is dropped. *However*, if a row from the page is still in the message buffer, the page is still locked. In contrast, the *Release Locks (RL)* isolation level increases concurrency by releasing all shared locks at the time that control returns to the client.

When the next message or command is sent to the database, SQLBase acquires share locks on only those pages that belong the current cursor. The locks are obtained regardless of the current command type. Just before returning to the user, SQLBase releases all shared locks. It also internally notes the page numbers of those pages that had locks on them.

This isolation level fills the input message buffer with rows, which minimizes network traffic.

Use this isolation level for browsing applications which display a set of rows to a user.

Parameters

cur

The cursor handle associated with this function.

isolation

A pointer to the variable that contains the isolation level:

```
SQLILRR "RR"   /* Repeatable Read isolation */
SQLILCS "CS"   /* Cursor Stability isolation */
SQLILRO "RO"   /* Read-Only isolation */
SQLILRL "RL"   /* Release Locks isolation */
```

Return value

The return value is zero (0) if the function succeeds and non-zero if it fails.

Example

This example sets the isolation level to Cursor Stability.

```
ret = sqlsil(cur, SQLILCS);
```

Related functions

sqlims

sqlspr - StoP Restriction mode

Syntax

```
#include <sql.h>

SQLTAPI  sqlspr (cur)

SQLTCUR  cur;     /* Cursor handle */
```

Description

This function turns off restriction mode but leaves on result set mode. Result set mode lets the application set a position at any row in the result set while not restricting each subsequent query.

In result set mode, once a result set has been created, you can get any row in the result set with the *sqlprs* function without sequentially fetching forward. Once the cursor is positioned, later fetches start from that row.

Parameters

cur

The cursor handle associated with this function.

Return value

The return value is zero (0) if the function succeeds and non-zero if it fails.

Example

```
if (rcd = sqlspr(cur))
  apierr("SQLSPS");
```

Related functions

sqlcrs *sqlrrs* *sqlstr*
sqldrs *sqlsrs* *sqlurs*
sqlprs

sqlsrs - Start Restriction Set and Result Set modes

Syntax

```
#include <sql.h>

SQLTAPI   sqlsrs(cur)

SQLTCUR   cur;     /* Cursor handle */
```

Description

This function starts restriction mode and result set mode.

In *result set mode*, once a result set has been created, you can get any row in the result set with the *sqlprs* function without sequentially fetching forward. Once the cursor is positioned, later fetches start from that row.

In *restriction mode*, the result set of one query is the basis for the next query. Each query further restricts the result set. This continues for each subsequent query.

After you call *sqlsrs*, you can call the *sqlspr* function to turn off restriction mode but leave result set mode on. You can call the *sqlstr* function to turn on restriction mode again after being in only result set mode.

While in restriction mode, you can "undo" the current result set and return to the result set as it was before the last SELECT with the *sqlurs* function.

If you are in restriction mode and you query a different table, the previous result set is lost.

You turn off both result set mode and restriction mode with the *sqlcrs* function. The *sqlcrs* function lets you optionally give a name to the result set and save it. To use a saved result set later, call the *sqlrrs* function and specify the saved result set name. The *sqlrrs* function turns on result set mode and restriction mode.

Be cautious about using saved result sets. Internally, a saved result set is a list of row identifiers (ROWIDs) that is stored in the SYSROWIDLISTS system catalog table. A ROWID changes whenever the row is updated. If one of the rows is updated after you have saved and closed a result set, you will get an error if you open the result set later and try to fetch the row.

You cannot use restriction mode with the following features:

- Aggregate functions
- DISTINCT

- GROUP BY
- HAVING
- UNION
- ORDER BY
- Stored commands

Parameters

cur

The cursor handle associated with this function.

Return value

The return value is zero (0) if the function succeeds and non-zero if it fails.

Example

```
ret = sqlsrs(cur);
```

Related functions

sqlcrs	*sqlrrs*	*sqlstr*
sqldrs	*sqlspr*	*sqlurs*
sqlprs	*sqlsrs*	

sqlssb - Set SELECT Buffer

Syntax

```
#include <sql.h>

SQLTAPI   sqlssb(cur, slc, pdt, pbp, pdl, sca, cvl, pfc)

SQLTCUR        cur;   /* Cursor handle */
SQLTSLC        slc;   /* Column number */
SQLTPDT        pdt;   /* Program data type */
SQLTDAP        pbp;   /* Program buffer */
SQLTPDL        pdl;   /* Program buffer length */
SQLTSCA        sca;   /* Scale of packed decimal data */
SQLTCDL   PTR  cvl;   /* Current value length */
SQLTFSC   PTR  pfc;   /* Fetch status code */
```

Description

This function sets up buffers that receive data from a *sqlfet*. This function associates an item in the SELECT list with a data buffer where the data is fetched.

This function tells the system where to put fetched data, the size of the receiving area, and the application program data type.

Also, this function sets up variables that are set after each *sqlfet*:

- Length of fetched data (*cvl* argument).
- Fetch status code (*pfc* argument).

This function must be issued once for each item in the SELECT list for which data is to be retrieved.

Parameters

cur

The cursor handle associated with this function.

slc

The column number indicates the sequence number (starting with 1) of the item in the SELECT list for which the program is setting up a select buffer.

pdt

The data type of the SELECT item as declared by the program. SQLBase automatically converts fetched data into this requested data type.

The program data types are listed below. These are defined in *sql.h*.

Program Data Type	Description
SQLPBUF	Character buffer
SQLPDAT	Internal datetime
SQLPDOU	Double
SQLPDTE	Date only
SQLPEBC	EBCDIC buffer
SQLPFLT	Float
SQLPLON	Long text string
SQLPLBI	Long binary buffer

Program Data Type	Description
SQLPLVR	Char/long varchar >254
SQLPNBU	Numeric buffer
SQLPNST	Numeric string
SQLPNUM	Internal numeric
SQLPSCH	Character
SQLPSIN	Integer
SQLPSLO	Long
SQLPSPD	Signed packed decimal
SQLPSSH	Short
SQLPSTR	String (null-terminated)
SQLPTIM	Time only
SQLPUCH	Unsigned character
SQLPUIN	Unsigned integer
SQLPULO	Unsigned long
SQLPUPD	Unsigned packed decimal
SQLPUSH	Unsigned short

pbp

A pointer to the variable where a later *sqlfet* returns the data for a SELECT list item.

Assign a value to this variable before each *sqlfet*. When you fetch a column with a null value, the value of *pbp* is not changed.

pdl

The length of the value pointed to by *pbp*.

sca

The scale (number of decimal places) for a packed-decimal data type. This argument is ignored for other data types. If you are not using a packed-decimal data type, specify zero.

cvl

The length of the data received by *pbp*. If the size of *pbp* is smaller than the actual data received, the data is truncated and an error will be indicated in the fetch status code for this column.

If the actual data received into *pbp* is shorter than *pdl*, then *cvl* is set to the actual length received after a *sqlfet*. For example, if the string "TEST" is received into a 20 character variable, *cvl* is set to 4.

Specify a null pointer (SQLNPTR) if this information is not wanted by the application.

If the data type is packed-decimal, see the section called *Packed-Decimal Data Types* in chapter 3.

pfc

A pointer to the variable where the *sqlfet* function returns the fetch status code for the specified column. If the fetch was successful, the fetch return code is zero. The following is a list of the fetch errors which can be returned. These codes are defined in *sql.h*.

Note: To set the pfc parameter to the constant FETRNUL, you must set the SQLPNIE parameter of the *sqlfet* function to 1 (on). Setting SQLPNIE affects all the cursors connected by the application that set it; it does not affect other applications.

Specify a null pointer (SQLNPTR) if this information is not wanted by the application.

Constant	Value	Description
FETRTRU	1	Data was truncated
FETRSIN	2	Signed number fetched into unsigned variable
FETRDNN	3	Data is not numeric
FETRNOF	4	Numeric overflow
FETRDTN	5	Data type is not supported
FETRDND	6	Data is not a date
FETRNUL	7	Data is null

Return value

The return value is zero (0) if the function succeeds and non-zero if it fails.

Example

```
#define NAMESIZE 25
#define COLWID 30

static char select[] = "select name, phone from emp where
           acode = :1";
char dataline[80];
unsigned char cvl;
char fcd;
short ret;          /* return code */
char *lp = dataline;/* line pointer */
SQLTCUR cur;        /* cursor*/
SQLTSLC col;        /* column number */


/* Set buffer for receiving data as locations on a line */

memset (dataline, ' ', sizeof(dataline));/*initialize */
for (col=1, col <= 2, col++)
  {
    if (ret = sqlssb(cur,col,SQLPBUF, lp, COLWID, 0, &cvl,
          &fcd))
      {
      ... process error
      }
    /* set line location for next item of data*/
    lp += (COLWID+2);
}
```

Related functions

sqldes *sqlfet* *sqlgfi*

sqlsta - STAtistics

Syntax

```
#include <sql.h>

SQLTAPI sqlsta (cur, svr, svw, spr, spw)

SQLTCUR       cur; /* Cursor handle       */
SQLTSTC  PTR  svr; /* Virtual reads       */
SQLTSTC  PTR  svw; /* Virtual writes      */
SQLTSTC  PTR  spr; /* Physical reads      */
SQLTSTC  PTR  spw; /* Physical writes     */
```

Description

This function returns database statistics about physical and virtual disk reads and writes since the specified cursor was connected.

The numbers returned for physical reads and writes refer to disk input/output operations. Physical means that data was physically transferred to or from the disk. Logical means that data was accessed by SQLBase access methods. This may or may not result in physical input/output.

The number of virtual reads and writes can be greater than, but never less than, the physical reads and writes. More physical writes can occur because a page may be forced out of the cache by a commit or a read.

The amount of disk input/output can be affected by the size of the server cache.

Parameters

cur

The cursor handle associated with this function.

svr

A pointer to the variable where this function returns the number of virtual reads.

svw

A pointer to the variable where this function returns the number of virtual writes.

spr

A pointer to the variable where this function returns the number of physical reads.

spw

A pointer to the variable where this function returns the number of physical writes.

Return value

The return value is zero (0) if the function succeeds and non-zero if it fails.

Example

```
SQLTCUR cur;
unsigned long svr; /* virtual reads */
unsigned long svw; /* virtual writes */
unsigned long spr; /* physical reads */
unsigned long spw; /* physical writes */

if (sqlsta(cur,&svr,&svw,&spr,&spw))
  {
    process error
  }
printf("Virtual reads:%ld  Virtual writes:%ld\n",svr,svw);
printf("Physical reads:%ld  Physical writes: %ld\n",
  spr,spw);
```

Related functions

sqlgsi

sqlstm - Server TerMinate

Syntax

```
#include <sql.h>

SQLTAPI  sqlstm (shandle)

SQLTSVH  shandle;  /* Server handle */
```

Description

This function causes the server program to exit. The server program terminates just as though a user had pressed **Esc** at the server computer.

You must call *sqlcsv* (Connect to SerVer) prior to calling this function.

If no users are connected, then it is a graceful shutdown.

If users are connected, then their sessions are terminated and the server exits. Connected users will get a "session terminated" message. All open transactions are left uncommitted. When the server is brought back up, crash recovery is performed and any uncommitted transactions will be rolled back.

Parameters

shandle

The server handle returned by *sqlcsv.*

Return value

The return value is zero (0) if the function succeeds and non-zero if it fails.

Example

```
if (rcd = sqlcsv(&handle,srvname,password))
  printf("Error : SQLCSV : rcd= %d\n",rcd);
if (rcd = sqlstm(handle))
  printf("Error : SQLSTM : rcd= %d\n",rcd);
```

Related functions

sqlcdr *sqlsab* *sqlsdn*
sqlcsv

sqlsto - STOre a compiled command/procedure

Syntax

```
#include <sql.h>

SQLTAPI  sqlsto (cur,cnp,cnl,ctp,ctl)

SQLTCUR  cur;   /* Cursor handle */
SQLTDAT  cnp;   /* Command/procedure name buffer */
SQLTDAL  cnl;   /* Command/procedure name length */
SQLTDAP  ctp;   /* Command/procedure text buffer */
SQLTDAL  ctl;   /* Command/procedure text length */
```

Description

This function compiles and stores a SQL command or procedure in the database with the specified name in the SYSCOMMANDS system catalog table. A stored SQL command or procedure must be retrieved (*sqlret*) before it can be executed.

Parameters

cur

The cursor handle associated with this function.

cnp

A pointer to the string that contains the name of the SQL command or procedure. The maximum length of the name is 18 characters.

cnl

The length of the string pointed to by *cnp*. If the string pointed to by *cnp* is null-terminated, specify zero and the system will compute the length.

ctp

A pointer to the string that contains the SQL command or procedure to compile and store.

ctl

The length of the string pointed to by *ctp*. If the string pointed to by *ctp* is null-terminated, specify zero and the system will compute the length.

Return value

The return value is zero (0) if the function succeeds and non-zero if it fails.

Example

```
short   rcd;      /* return code */
static  char   statement [] = "INSERT INTO CUST (:1, :2, :3,
:4)";

if (rcd = sqlsto(cur, "ADDCUST", 0, statement, 0))
  {
    printf("Error storing SQL statement (rcd = %d)\n",rcd);
    exit(0);
  }
```

Related functions

sqldst *sqlret*

sqlstr - STart Restriction mode

Syntax

```
#include <sql.h>

SQLTAPI  sqlstr (cur);

SQLTCUR  cur;   /* Cursor handle */
```

Description

This function turns on restriction mode after being in result mode only.

In restriction mode, the result set of one query is the basis for the next query. Each query further restricts the result set. This continues for each subsequent query.

After you call *sqlstr*, you can call the *sqlspr* function to turn off restriction mode but leave result set mode on.

While in restriction mode, you can "undo" the current result set and return to the result set as it was before the last SELECT with the *sqlurs* function.

If you are in restriction mode and you query a different table, the previous result set is lost.

You turn off both result set mode and restriction mode with the *sqlcrs* function. The *sqlcrs* function lets you optionally give a name to the result set and save it. To use a saved result set later, call the *sqlrrs* function and specify the saved result set name. The *sqlrrs* function turns on result set mode and restriction mode.

Be cautious about using saved result sets. Internally, a saved result set is a list of row identifiers (ROWIDs) that is stored in the SYSROWIDLISTS system catalog table. A ROWID changes whenever the row is updated. If one of the rows is updated after you have saved and closed a result set, you will get an error if you open the result set later and try to fetch the row.

Parameters

cur

The cursor handle associated with this function.

Return value

The return value is zero (0) if the function succeeds and non-zero if it fails.

Example

```
if (rcd = sqlstr(cur))
  apierr("SQLSTR");
```

Related functions

sqlcrs *sqlrrs* *sqlsrs*
sqldrs *sqlspr* *sqlurs*
sqlprs

sqltec - Translate Error Code

Syntax

```
#include <sql.h>

SQLTAPI  sqltec (rcd, np)

SQLTRCD          rcd;   /* SQLBase return code to convert */
SQLTRCD  PTR     np;    /* Converted return code */
```

Description

This function translates the specified SQLBase return code to another return code based on an entry in the error translation file specified by the *errorfile* keyword in *sql.ini*.

For information on the *errorfile* configuration keyword, see Chapter 4 and the *Configuration* chapter in the *Database Administrator's Guide*.

Parameters

rcd

The SQLBase return code to translate.

np

A pointer to the variable where this function returns the translated return code.

Return value

If this function returns zero, it was successful. If this function returns a non-zero value, it was unsuccessful or the SQLBase return code was not found in the error translation file.

Example

```
#include <sql.h>

SQLTRCD  srcd;    /* SQLBase return code */
SQLTRCD  trcd;    /* translated return code */
SQLTRCD  rcd;     /* function call return code */

if (rcd = sqltec(srcd, &trcd))/* translate SQLBase rcd */
  printf("Failure translating rcd (rcd = %d)\n", rcd);
```

Related functions

sqlerr *sqlfer* *sqlrcd*

sqltem - Tokenize Error Message

Syntax

```
#include <sql.h>

SQLTAPI   sqltem (cur, rcd, msgtyp, bfp, bfl, txtlen)

SQLTCUR         cur    /* Cursor handle */
SQLTXER   PTR   rcd;   /* Database return code */
SQLTPTY         msgtyp;/* Error message text type to return*/
SQLTDAP         bfp;   /* Pointer to error text buffer */
SQLTDAL         bfl;   /* Length of error text buffer */
SQLTDAL   PTR   txtlen;/* Length of error text */
```

Description

This function returns one or more of the following from the *error.sql* file for the specified cursor handle:

- Error message
- Error reason

- Error remedy

Each API function call returns a code. You can retrieve the most recent return code with the *sqltem* function, and use it to look up the error message, error reason, and error remedy.

This function formats an error message with tokens in order to provide users with more informational error messages. A tokenized error message contains one or more variables that SQLBase substitutes with object names (tokens) when it returns the error message to the user.

For example, formerly, SQLBase error 175:

```
SQL OLC Cannot open local client workstation file
```

is now:

```
SQL OLC Cannot open local client workstation file <filename>
```

where *filename* is a variable that gets replaced with the name of the file that SQLBase was unable to open.

Tokenized error messages produce informative integrity errors. For example, the following message text for error 9601 reports the table or index name as well as merely informing you that the table is corrupt or the index is bad:

```
CHECK Failure (IDX BPT): <index page corrupted>
```

When this error occurs, SQLBase replaces the *index page corrupted* variable (and the brackets) with the actual name of the index that contains the corruption.

Non-SQLBase database servers

By default, the *sqltem* function returns the native error code and message from non-SQLBase database servers, but does not return the error reason or remedy.

For example, if you are connected to the Informix server and you receive an error for a table that already exists, the error returned is the Informix error code 310:

```
An attempt was made to create a tablespace which already
exists
```

not SQLBase's equivalent 338:

```
Table, view, or synonym <name> already exists
```

If you are accessing a non-SQLBase database server and have set error mapping on, any non-SQLBase error that doesn't have a corresponding SQLBase error is mapped to a generic error message. You can use the *sqltem* function to retrieve the native error code and message that caused the problem.

Note: The other error message handling functions (*sqlerr*, *sqlfer*, and *sqletx*) use a specified return code to retrieve the corresponding error message from the *error.sql* file. An error message returned by any of these functions contains the variable, not the object name; only the *sqltem* function replaces the variable with an actual object name.

Parameters

cur

The cursor handle on which an error occurred. Use this cursor handle to retrieve the error message, reason, and/or remedy of a SQLBase error.

Do not attempt to call the *sqltem* function when you fail to establish a connection to a database. In such a case, the cursor is invalid because it was unable to connect to the database. Use the *sqletx* function, pass it the error code, and specify a *msgtyp* parameter value of 6 in order to retrieve the error message reason and remedy.

rcd

A pointer to the return code value.

The error code is database-specific, so when you are accessing a non-SQLBase database server, the return code value does not have a corresponding error reason and/or remedy in *error.sql*.

msgtyp

You can specify the following message types individually or together by adding their constant values. For example, a value of seven indicates that you want the error message text, reason, and remedy all returned in the buffer to which *bfp* points.

Constant name	*Value*	*Explanation*
SQLXMSG	1	Retrieve the error message text.
SQLXREA	2	Retrieve the error message reason.
SQLXREM	4	Retrieve the error message remedy.

A value of SQLXMSG (1) is assumed for non-SQLBase database servers.

bfp

A pointer to the buffer where this function copies the error message text, reason, or remedy.

bfl

The length of the buffer pointed to by *bfp*.

If you are retrieving the error message text, reason, and remedy, you can specify the *sql.h* constant SQLMETX for this argument. SQLMETX is always set to a value that is large enough to hold the error message text, reason, and remedy.

If you are only retrieving the error message text, you can specify the *sql.h* constant SQLMERR for this argument. SQLMERR is always set to a value that is large enough to hold the error message text.

txtlen

A pointer to the variable where this function returns the number of bytes that exist for either the error message text, reason, or remedy.

If the buffer pointed to by *bfp* holds 100 bytes but the text you are retrieving is 500 bytes, *sqltem* returns 100 bytes of text to your application and sets this parameter to 500. At this point, your application can reallocate a larger buffer in order to retrieve all the text.

Specify a null pointer to indicate that you are not interested in the total length of the text.

Return value

The return value is zero (0) if the function succeeds and non-zero if it fails.

Example

```
SQLTCUR   cur;           /* Cursor value */
SQLTXER   rcd;           /* Return code */
char      buf[SQLMETX];  /* Buffer to receive the text */
SQLTDAL   txtlen;        /* Length of text returned*/

if (sqlcom (cur, “CREATE TABLE EMP (LASTNAME CHAR(20))”, 0))
{
  sqltem (cur, &rcd, SQLXMSG + SQLXREA + SQLXREM, buf,
         sizeof(buf), &txtlen)
  printf (“Error Explanation:\n%s\n”, buf);
}
```

Related functions

sqlerr *sqletx* *sqlfer*
sqlxer

sqltio - TIme Out

Syntax

```
#include <sql.h>

SQLTAPI sqltio (cur, timeout)

SQLTCUR   cur;      /* Cursor handle */
SQLTTIV   timeout;  /* Wait period in seconds */
```

Description

This function specifies a wait time for getting a lock after which a timeout happens and the transaction rolls back. The timeout is set on a per-cursor basis and stays in effect until the next *sqltio* function.

This function is not useful for a single-user server.

Parameters

cur

The cursor handle associated with this function.

timeout

The timeout period in seconds to wait for a database lock to be acquired. After the specified time has elapsed, the transaction is rolled back.

Valid timeout values are:

1-1800	Seconds to wait for a lock (1 second to 30 minutes)
-1	Wait forever for a lock held in an incompatible mode by another transaction (infinite timeout)
0	Never wait for a lock and immediately return a timeout error

The default setting is 300 seconds.

Return value

The return value is zero (0) if the function succeeds and non-zero if it fails.

Example

```
main()
{
  SQLTCUR   cur;
  SQLTTIV   timeout=500;

  static    char dbnam[]="demox";      /* database name */

/* CONNECT TO THE DATABASE */

cur = 0;
if (rcd = sqlcnc(&cur, dbnam, 0))/* perform connect
                                    operation */
  apierr("SQLCNC");

if (rcd = sqltio(cur,timeout))
  apierr("SQLTIO");

/* DISCONNECT FROM THE DATABASE */

if (rcd = sqldis(cur))          /* failure on disconnect? */
  apierr("SQLDIS");
}
```

Related functions

sqlsil

sqlunl - UNLOAD command

Syntax

```
#include <sql.h>

SQLTAPI  sqlunl(cur, cmdp, cmdl)

SQLTCUR  cur;     /* cursor number */
SQLTDAP  cmdp;    /* UNLOAD command */
SQLTDAL  cmdl;    /* length of above command */
```

Description

This function processes the UNLOAD command and sends it to the backend for compilation and execution. If the unload file destination is on the server, the

execution is handled completely at SERVER. If it is ON CLIENT, this function handles the retrieval of the unload data from the SERVER and writes it to the destination file.

To unload to multiple file segments, you can create a control file that defines your segments and specify the control file name in this function. For details on creating the control file, read the *Database Administrator's Guide*.

Parameter

cur

The cursor handle associated with this function

cmdp

A pointer to the string that contains the UNLOAD command.

cmdl

The length of the string pointed to by *cmdp*. If the string pointed to by *cmdp* is null-terminated, specify zero and the system will compute the length.

Return value

If this function returns zero, it was successful. If this function returns a non-zero value, it was unsuccessful.

Example

The following sample program calls the UNLOAD command and inputs a file name that exists online:

```
static char unlcmd[] =
"UNLOAD COMPRESS DATA SQL db.unl ALL ON SERVER ;";
ret = sqlunl(cur, unlcmd, 0);
```

You can also create a customized program to manipulate the unload buffer in the client, such as unloading to archive data. For an example, see the *Loading and unloading databases* section in the chapter, *Using the SQL/API*.

sqlurs - Undo Result Set

Syntax

```
#include <sql.h>

SQLTAPI  sqlurs (cur)

SQLTCUR  cur;   /* Cursor handle */
```

Description

In restriction mode, this function undoes the current result set and returns the result set to the state it was in before the last query.

Parameters

cur

The cursor handle associated with this function.

Return value

The return value is zero (0) if the function succeeds and non-zero if it fails.

Example

Execute these commands in restriction mode:

```
SELECT * FROM LOC WHERE STATE_POP > 2000000;
SELECT * FROM LOC WHERE STATE_AREA > 150000;
```

The *sqlurs* function below drops the result table created by the second command and makes all later queries for the table operate on the result table produced by the first command.

```
ret = sqlurs(cur);
```

Related functions

sqlcrs *sqlrrs* *sqlsrs*
sqldrs *sqlspr* *sqlstr*
sqlprs

sqlwlo - Write LOng

Syntax

```
#include <sql.h>

SQLTAPI  sqlwlo (cur, bufp, bufl)

SQLTCUR  cur;     /* Cursor handle */
SQLTDAP  bufp;    /* Data to write */
SQLTDAL  bufl;    /* Length of long data */
```

Description

This function writes *bufl* bytes at a time to a LONG VARCHAR column.

This function is called after *sqlcom* has been performed and the LONG VARCHAR column has been bound, but before *sqlexe*.

This function allows the incremental writing of large columns without having to set up equivalent size data buffers to hold the data.

The function is called repeatedly until the total amount of data is written to the database column. After each call to *sqlwlo*, the API increments a pointer that indicates where the next *sqlwlo* should begin. The API resets the pointer after a *sqlelo*.

The sequence of binding, writing, and ending the operation must be completed before the next bind for a LONG VARCHAR.

You *cannot* seek to a position within a LONG VARCHAR with the *sqllsk* function and start writing with *sqlwlo*. You must always start writing the LONG VARCHAR column at the first byte.

The maximum length that you can write in one call to *sqlwlo* is 32,767 bytes.

Parameters

cur

The cursor handle associated with this function.

bufp

A pointer to the string that contains the LONG VARCHAR data to write.

bufl

The length of the string pointed to by *bufp*. If the string pointed to by *bufp* is null-terminated, specify zero and the system will compute the length.

Return value

The return value is zero (0) if the function succeeds and non-zero if it fails.

Example

```
static char updlong[] = "update people set biography = :1
      where name = :2";

/* Prior to sqlwlo, the above SQL statement has been */
/* compiled */
/* (sqlcom), :1 is bound using sqlbln; :2 is bound */
/* using sqlbnn */

FILE *fp;
int count;

char buffer[500];
while (count = fread(buffer, 1, sizeof(buffer), fp))
  {
    if (!(ret = sqlwlo(cur, buffer, count)))
      {
      ... process error
      }
  }
if (sqlelo(cur))
  ...
```

Related functions

sqlbld *sqlelo* *sqlrlo*
sqlbln

sqlxad - eXtended ADd

Syntax

```
#include <sql.h>

SQLTAPI  sqlxad(op, np1, nl1, np2, nl2);

SQLTNMP  op;      /* Output number */
SQLTNMP  np1;     /* First number */
SQLTNML  nl1;     /* Length of first number */
SQLTNMP  np2;     /* Second number */
SQLTNML  nl2;     /* Length of second number */
```

Description

This function adds two SQLBase internal numbers.

Incorrect data in any argument can cause unpredictable results.

Parameters

op

A pointer to a variable where this function returns the sum. Define the length of this variable as SQLSNUM.

np1

A pointer to a variable that contains the first number to add. Define the length of this variable as SQLSNUM.

nl1

The length of the number pointed to by *np1*.

np2

A pointer to a variable that contains the second number to add. Define the length of this variable as SQLSNUM.

nl2

The length of the number pointed to by *np2*.

Return value

This function returns the length of the resulting number if execution is successful. If execution is not successful, this function returns a negative value.

Example

```
/* ADD NUMBER 1 AND NUMBER 2, PUTTING THE RESULT */
/* INTO NUMBER 3 */

char   num1[SQLSNUM];   /* number 1 */
int    nl1;             /* number 1 length */
char   num2[SQLSNUM];   /* number 2 */
int    nl2;             /* number 2 length */
char   num3[SQLSNUM];   /* number 3 */
int    nl3;             /* number 3 length */

nl3 = sqlxad(num3, num1, nl1, num2, nl2);
```

Related functions

sqlxdv *sqlxml* *sqlxsb*

sqlxcn - eXtended CoNvert

Syntax

```
#include <sql.h>

SQLTAPI  sqlxcn(op, ip, il)

SQLTNMP  op;      /* Output number */
SQLTDAL  ip;      /* Input character string */
SQLTNML  il;      /* Length of input string */
```

Description

This function converts a character string to a SQLBase internal number.

Incorrect data in any argument can cause unpredictable results.

Parameters

op

A pointer to the variable where this function returns the SQLBase internal number. Define the length of this variable as SQLSNUM.

ip

A pointer to the string that contains the character string to convert.

il

The length of the string pointed to by *ip*. If the string pointed to by *ip* is null-terminated, specify zero and the system will compute the length.

Return value

This function returns the length of the resulting number if execution is successful. If execution is not successful, this function returns a negative value.

Examples

Example 1

```
char     num[SQLSNUM];  /* internal SQLBase number */
int      nl;            /* length of internal number */

nl = sqlxcn(num, "5900.99", 7);
```

Example 2

```
#include “sql.h”
#include “stdio.h”
#include “string.h”

main ()
{
char   output[12];
int    rcd;
char   num[SQLSNUM];
int    nl;

nl = sqlxcn(num, “123456”,6);
printf(“nl = %d\n”, nl);
rcd = sqlxnp(output,sizeof(output),num,nl,”zzz,zzz.99”,10);
printf(“RCD = %d output = %s\n”,rcd,output);
exit(1);
```

Related functions

sqlxnp

sqlxda - eXtended Date Add

Syntax

```
#include <sql.h>

SQLTAPI  sqlxda(op, dp, dl, days)

SQLTNMP  op;      /* Output date */
SQLTNMP  dp;      /* Internal SQLBase date */
SQLTNML  dl;      /* Length of internal SQLBase date */
SQLTDAY  days;    /* Number of days to add */
```

Description

This function adds *n* days to a SQLBase internal date.

Incorrect data in any argument can cause unpredictable results.

Parameters

op

A pointer to the variable where this function returns the output date. Define the length of this variable as SQLSDAT.

dp

A pointer to the variable that contains the SQLBase internal date. Define the length of this variable as SQLSDAT.

dl

The length of the internal date pointed to by *dp*. Pass the length from the *sqlxpd* or *sqlssb.*

days

The number of days to add to *dp*.

Return value

This function returns the length of the resulting date if execution is successful. If execution is not successful, this function returns a negative value.

Example

```
/* ADD 30 DAYS TO A DATE */

char    date1[SQLSDAT];/* starting date */
int     dl1;           /* starting date length */
char    date2[SQLSDAT];/* resulting date */
int     dl2;           /* resulting date length */

dl2 = sqlxda(date2, date1, dl1, 30);
```

Related functions

sqlxdp *sqlxpd*

sqlxdp - eXtended Date to Picture

Syntax

```
#include <sql.h>

SQLTAPI  sqlxdp (op, ol, ip, il, pp, pl)

SQLTDAP  op;    /* Null-terminated string */
SQLTDAL  ol;    /* Length of null-terminated string */
SQLTNML  ip;    /* Internal SQLBase date */
SQLTNLM  il;    /* Length of internal SQLBase date */
SQLTDAP  pp;    /* Picture specification */
SQLTDAL  pl;    /* Length of picture specification */
```

Description

This function converts a SQLBase internal date to a string using the specified picture format.

Use the *cvl* argument in the *sqlssb* function to pass the length to *sqlxdp* (*il* argument).

Incorrect data in any argument can cause unpredictable results.

Parameters

op

A pointer to the variable where this function returns the output string. The output is formatted as a null-terminated string. If the length is less than the specified picture length, the output is truncated.

ol

The length of the variable pointed to by *op*.

ip

A pointer to the variable that contains the SQLBase internal date. Define the size of this variable as SQLSDAT.

il

The length of the internal date pointed to by *ip*. Do *not* use a fixed length because SQLBase internal numbers are variable length.

pp

A pointer to the variable that contains the picture specification. This function performs the following substitutions in the picture specification.

Characters	Replaced by
MM	A two digit number representing the month.
MON	A three character abbreviation for the month.
DD	A two digit number representing the day of the month.
YY	The last two digits of the year.
YYYY	The four digits of the year.
HH	A two digit number representing hours in military time.
MI	A two digit number representing minutes.
SS	A two digit number representing seconds.
AM or PM	Two characters: either AM or PM.
999999	A 6 or more digit number representing microseconds.

The characters, such as MM, are not case-sensitive. They can appear in upper- or lower-case in the picture. For example, if the input picture string is "Mon.dd.yyyy" and the input date is June 28, 1987, the output is "Jun.28.1987".

A backslash forces the next character into the output from the picture. For example: a picture of "Mo\mmy was born in YYYY" produces an output string of "Mommy was born in 1956" instead of "Mo04y was born in 1956".

pl

The length of the string pointed to by *pp*. Specify a zero if the string pointed to by *pp* is null-terminated.

Return value

The return value is zero (0) if the function succeeds and non-zero if it fails.

Example

```
char   date[SQLSDAT];
char   buf[14];
int    cvl;

rcd = sqlxdp(buf, sizeof(buf), date, cvl, "mon. dd, yyyy",
       0);
```

Related functions

sqlxpd

sqlxdv - eXtended DiVide

Syntax

```
#include <sql.h>

SQLTAPI   sqlxdv(op, np1, nl1, np2, nl2)

SQLTNMP   op;    /* Output number */
SQLTNMP   np1;   /* First number */
SQLTNML   nl1;   /* Length of first number */
SQLTNMP   np2;   /* Second number */
SQLTNML   nl2;   /* Length of second number */
```

Description

This function divides a SQLBase internal number by another SQLBase internal number.

Incorrect data in any argument can cause unpredictable results.

Parameters

op

A pointer to the variable where this function returns the output number. Define the length of this variable as SQLSNUM.

np1

A pointer to the variable that contains the first number. This number is divided by the number in *np2*. Define the length of this variable as SQLSNUM.

nl1

The length of the number pointed to by *np1*.

np2

A pointer to the variable that contains the second number. This number is divided into the number in *np1*. Define the length of this variable as SQLSNUM.

nl2

The length of the number pointed to by *np2*.

Return value

This function returns the length of the resulting number if execution is successful. If execution is not successful, this function returns a negative value.

Example

```
/* DIVIDE NUMBER 1 BY NUMBER 2; PUTTING THE RESULT */
/* INTO NUMBER 3 */

char   num1[SQLSNUM];   /* number 1        */
int    nl1;             /* number 1 length */
char   num2[SQLSNUM];   /* number 2        */
int    nl2;             /* number 2 length */
char   num3[SQLSNUM];   /* number 3        */
int    nl3;             /* number 3 length */

nl3 = sqlxdv(num3, num1, nl1, num2, nl2);
```

Related functions

sqlxad *sqlxml* *sqlxsb*

sqlxer - eXtended ERror

Syntax

```
#include <sql.h>

SQLTAPI  sqlxer (cur, rcd, errbuf, buflen)

SQLTCUR        cur;      /* Cursor handle */
SQLTXER  PTR   rcd;      /* Return code */
SQLTDAP        errbuf;   /* Ptr to receiving buffer */
SQLTDAL  PTR   buflen;   /* Length of receiving buffer */
```

Description

This function returns the most recent error code and associated error message text for the specified cursor handle. This function is used with non-SQLBase database servers to retrieve the native error code and message from the server.

You call this function when users or developers prefer to use native database error codes and messages instead of those in *error.sql*. Each SQLNetwork router or gateway has an equivalence table that maps native database error numbers to SQLBase error numbers (from *error.sql*). The router or gateway automatically translates the native database error codes to the *error.sql* error codes. You use *sqlxer* to retrieve the native error codes and messages.

For example, the Informix error code for a duplicate table is 310. The router or gateway translates this to SQLBase error code 336:

- The *sqlexe* return code is 336 and *sqlerr* returns "table or view already exists".
- The *sqlxer* function returns 310 and “An attempt was made to create a tablespace which already exists.”

You also use this function to get more information about generic errors. Any native database error number that does not have an equivalent SQLBase error number is mapped to a common generic error number. The generic error number is 2550 plus the value of SQLPBRN (database brand parameter). You use *sqlxer* to retrieve the native error code and message that caused the generic error.

For example, if the error code is Informix’s 310, the router or gateway translates this to the SQLBase generic error code 2553 (2550 plus 3):

- The *sqlexe* return code is 2553 and *sqlerr* returns "Oracle processing error; more info available".
- The *sqlxer* function returns 310 and “An attempt was made to create a tablespace which already exists.”

Parameters

cur

The cursor handle associated with this function.

rcd

A pointer to the buffer where this function returns the most-recent error code for the cursor.

errbuf

A pointer to the buffer where this function copies the error message text.

You can use the *sql.h* constant SQLMXER to set the size of this buffer.

buflen

A pointer to the variable where this function returns the number of bytes in the retrieved error message text.

Return value

The return value is zero (0) if the function succeeds and non-zero if it fails.

Example

```
sqlxer(cur, rcd, errbuf, buflen)
```

Related functions

sqlerr *sqlfer* *sqlrcd*
sqletx

sqlxml - eXtended MuLtiply

Syntax

```
#include <sql.h>

SQLTAPI   sqlxml(op, np1, nl1, np2, nl2)

SQLTNMP   op;  /* Output number */
SQLTNMP   np1; /* First number */
SQLTNML   nl1; /* Length of first number */
SQLTNMP   np2; /* Second number */
SQLTNML   nl2; /* Length of second number */
```

Description

This function multiplies two SQLBase internal numbers.

Incorrect data in any argument can cause unpredictable results.

Parameters

op

A pointer to the variable where this function returns the output number.

np1

A pointer to the variable that contains the first number. This number is multiplied by the number in *np2*. Define the length of this variable as SQLSNUM.

nl1

The length of the number pointed to by *np1*.

np2

A pointer to the variable that contains the second number. This number is multiplied by the number in *np1*. Define the length of this variable as SQLSNUM.

nl2

The length of the number pointed to by *np2*.

Return value

This function returns the length of the resulting number if execution is successful. If execution is not successful, this function returns a negative value.

Example

```
/* MULTIPLY NUMBER 1 & NUMBER 2; PUTTING THE RESULT */
/* INTO NUMBER 3 */

char     num1[SQLSNUM];   /* number 1 */
int      nl1;             /* number 1 length */
char     num2[SQLSNUM];   /* number 2 */
int      nl2;             /* number 2 length */
char     num3[SQLSNUM];   /* number 3 */
int      nl3;             /* number 3 length */

nl3 = sqlxml(num3, num1, nl1, num2, nl2);
```

Related functions

sqlxad *sqlxdv* *sqlxsb*

sqlxnp - eXtended Number to Picture

Syntax

```
#include <sql.h>

SQLTAPI   sqlxnp (outp, outl, isnp, isnl, picp, picl)

SQLTDAP   outp;     /* Converted internal number */
SQLTDAL   outl;     /* Output buffer length */
SQLTNMP   isnp;     /* Internal SQLBase number */
SQLTNML   isnl;     /* Internal SQLBase number length */
SQLTDAP   picp;     /* Picture specification */
SQLTDAL   picl;     /* Picture specification length */
```

Description

This function converts a SQLBase internal number to a string using a picture format.

Incorrect data in any argument can cause unpredictable results.

Parameters

outp

A pointer to the variable where this function returns the converted SQLBase internal number. The output is a null-terminated string. If the output length is less than the specified picture length, the output is truncated.

outl

The length of the variable pointed to by *outp*.

isnp

A pointer to the variable that contains the SQLBase internal number to convert. Define the length of this variable as SQLSNUM.

isnl

The length of the value pointed to by *isnp*.

picp

A pointer to the variable that contains the picture specification. The picture specification must combine to represent a valid number. For example, commas must be spaced three to the left of the decimal point and only one decimal point allowed per number.

If the input number exceeds the number of digits in the picture string, the number is not displayed. Instead, the string is filled with asterisks meaning numeric overflow. If the number contains decimal digits and there are not enough significant decimal places in the picture, the number is rounded.

The following table shows the components that can be used in the picture string.

Character	Description
9	For every "9" in the picture string, a position is reserved. A value of 0 through 9 appears in every position indicated by "9".
.	Positions a decimal point in the output string. It can appear only once in a picture string.
,	Positions a comma in the output string. The commas in a picture string must conform to standard numeric notation.
Z	Replaces leading zeros with blanks (spaces) in the output string. This symbol must appear to the left of any digit specification of a numeric picture string.

Character	Description
$	Places a dollar sign in the output string. It can be at the beginning of a picture string or it can be used as a floating character (the symbol then only appears next to the most significant digit). The $ symbol cannot appear to the right of a 9, Z, or decimal point.
-	Places a minus sign in the output string if the algebraic value is negative.
E	Puts the output string in scientific notation.

picl

The length of the string pointed to by *picp*.

Return value

The return value is zero (0) if the function succeeds and non-zero if it fails.

Examples

Example 1

```
chari   numb[SQLSNUM];
char    inuml;
char    output[SQLSDAT];

rcd=sqlxnp(output,sizeof(output),inumb,inuml,"ZZZ,ZZZ.99-
      ",0);
```

Input Number	Picture String	Output String
123456	999999	123456
123456	9999999	0123456
123456	-99 9,999,999	000,123,456
-123456	-999,999,999	-000,123,456
123456	999,999.99	123,456.00
1234.56	9 99,999.99	001,234.56
12.3456	999,999.99	000,012.35

Input Number	Picture String	Output String
123456	-zzz,zzz,zzz	123,456
-123456	-zzz,z zz,zzz	-123,456
123456	zzz,zzz.99	123,456.00
1234.56	zzz,zzz.99	1,234.56
12.3456	zzz,zzz.99	12.35
123456	-$$$,$$$,$$$	$123,456
-123456	-$$$,$$$,$$$	- $123,456
123456	$$$,$$$.99	123,456.00
1234.56	$$$,$$$.99	$1,234.56
12.3456	$$$,$$$.99	$12.35

Example 2

```
#include “sql.h”
#include “stdio.h”
#include “string.h”

main ()
{
char    output[12];
int     rcd;
char    num[SQLSNUM];
int     nl;

nl = sqlxcn(num, “123456”,6);
printf(“nl = %d\n”, nl);
rcd = sqlxnp(output,sizeof(output),num,nl,”zzz,zzz.99”,10);
printf(“RCD = %d output = %s\n”,rcd,output);
exit(1);
```

Related functions

sqlxcn

sqlxpd - eXtended Picture to Date

Syntax

```
#include <sql.h>

SQLTAPI  sqlxpd (op, olp, ip, pp, pl)

SQLTNMP         op;      /* Output internal SQLBase date */
SQLTNML  PTR    olp;     /* Output length */
SQLTDAP         ip;      /* Null-terminated input string */
SQLTDAP         pp;      /* Picture specification */
SQLTDAL         pl;      /* Length of picture */
```

Description

This function converts a null-terminated string to a SQLBase internal date.

Use the *sqlxpd* function (*olp* argument) before the *sqlxdp* function to pass the length to *sqlxdp* (*il* argument).

Incorrect data in any argument can cause unpredictable results.

Parameters

op

A pointer to the variable where this function returns the SQLBase internal date. Define the length of this variable as SQLSDAT.

olp

The length of the value pointed to by *op*.

ip

A pointer to the variable that contains the null-terminated string to convert to a SQLBase internal date.

pp

A pointer to the string that contains the picture specification. This function performs the following substitutions in the picture string.

Characters	Replaced by
MM	A two digit number representing the month.
MON	A three character abbreviation for the month.
DD	A two digit number representing the day of the month.
YY	The last two digits of the year.
YYYY	The four digits of the year.
HH	A two digit number representing hours in military time.
MI	A two digit number representing minutes.
SS	A two digit number representing seconds.
AM or PM	Two characters: either AM or PM.
999999	A 6 or more digit number representing micro-seconds. Only the least significant 6 digits are considered.

The characters, such as MM, are not case-sensitive. They can appear in upper- or lower-case in the picture. For example, if the picture string is "Mon.dd.yyyy" and the input string is Jun.28.1987, the output is a SQLBase internal date of Jun-28-1987 12:00:00 PM.

A backslash forces the next character in the input to be skipped. For example: a picture of "Mo\mmy was born in YYYY" with an input string of "Mommy was born in 1956" produces a SQLBase internal date of Dec-31-1956 12:00:00 PM.

pl

The length of the string pointed to by *pp*.

Return value

The return value is zero (0) if the function succeeds and non-zero if it fails.

Example

```
char date[SQLSDAT];
int len;

rcd = sqlxpd(date, &len, "Jun. 28, 1987", "mon. dd, yyyy", 0);
```

Related functions

sqlxdp

sqlxsb - eXtended SuBtract

Syntax

```
#include <sql.h>

SQLTAPI   sqlxsb (op, np1, nl1, np2, nl2);

SQLTNMP   op;       /* Output number */
SQLTNMP   np1;      /* First number */
SQLTNML   nl1;      /* First number */
SQLTNMP   np2;      /* Second number */
SQLTNML   nl2;      /* Second number */
```

Description

This function subtracts one SQLBase internal number from another and puts the result in a third SQLBase internal number.

Incorrect data in any argument can cause unpredictable results.

Parameters

op

A pointer to the variable where this function returns the output number.

np1

A pointer to the variable that contains the first number. The value pointed to by *np2* is subtracted from this number. Define the length of this variable as SQLSNUM.

nl1

The length of the number pointed to by *np1*.

np2

A pointer to the variable that contains the second number. This number is subtracted from the number pointed by *np1*. Define the length of this variable as SQLSNUM.

nl2

The length of the number pointed to by *np2*.

Return value

This function returns the length of the resulting number if execution is successful. If execution is not successful, this function returns a negative value.

Example

```
/* SUBTRACT NUMBER 2 FROM NUMBER 1 AND PUTTING */
/* THE RESULT INTO NUMBER 3 */

char        num1[SQLSNUM];/* number 1 */
int         nl1;    /* number 1 length */
char        num2[SQLSNUM];/* number 2 */
int         nl2;    /* number 2 length */
char        num3[SQLSNUM];/* number 3 */
int         nl3;    /* number 3 length */

nl3 = sqlxsb(num3, num1, nl1, num2, nl2);
```

Related functions

sqlxad *sqlxdv* *sqlxml*

Glossary

***access path*—**The path used to get the data specified in a SQL command. An access path can involve an index or a sequential search (table scan), or a combination of the two. Alternate paths are judged based on the efficiency of locating the data.

***aggregate function*—**A SQL operation that produces a summary value from a set of values.

***alias*—**An alternative name used to identify a database object.

***API (application programming interface)*—**A set of functions that a program uses to access a database.

***application*—**A program written by or for a user that applies to the user's work. A program or set of programs that perform a task. For example, a payroll system.

***argument*—**A value entered in a command that defines the data to operate on or that controls execution. Also called parameter or operand.

***arithmetic expression*—**An expression that contains operations and arguments that can be reduced to a single numeric value.

***arithmetic operator*—**A symbol used to represent an arithmetic operation, such as the plus sign (+) or the minus sign (-).

***attribute*—**A characteristic or property. For example, the data type or length of a row. Sometimes, attribute is used as a synonym for column or field.

***audit file*—**A log file that records output from an audit operation.

***audit message*—**A message string that you can include in an audit file

***audit operation*—**A SQLBase operation that logs database activities and performance, writing output to an audit file. For example, you can monitor who logs on to a database and what tables they access, or record command execution time.

***authorization*—**The right granted to a user to access a database.

***authorization-ID*—**A unique name that identifies a user. Associated to each authorization-id is a password. Abbreviated auth-id. Also called username.

***back-end*—**See database server.

***backup*—**To copy information onto a diskette, fixed disk, or tape for record keeping or recovery purposes.

***base table*—**The permanent table on which a view is based. A base table is created with the CREATE TABLE command and does not depend on any other table. A base table has its description and its data physically stored in the database. Also called underlying table.

bindery—A NetWare 3.x database that contains information about network resources such as a SQLBase database server.

***bind variable*—**A variable used to associate data to a SQL command. Bind variables can be used in the VALUES clause of an INSERT command, in a WHERE clause, or in the SET clause of an UPDATE command. Bind variables are the mechanism to transmit data between an application work area and SQLBase. Also called into variable or substitution variable.

***browse*—**A mode where a user queries some of a database without necessarily making additions or changes. In a browsing application, a user needs to examine data before deciding what to do with it. A browsing application allows the user to scroll forward and backward through data.

***buffer*—**A memory area used to hold data during input/output operations.

***C/API*—**A language interface that lets a programmer develop a database application in the C programming language. The C/API has functions that a programmer calls to access a database using SQL commands.

***cache*—**A temporary storage area in computer memory for database pages being accessed and changed by database users. A cache is used because it is faster to read and write to computer memory than to a disk file.

***Cartesian product*—**In a join, all the possible combinations of the rows from each of the tables. The number of rows in the Cartesian product is equal to the number of rows in the first table times the number of rows in the second table, and so on. A Cartesian product is the first step in joining tables. Once the Cartesian product has been formed, the rows that do not satisfy the join conditions are eliminated.

cascade*—*A delete rule which specifies that changing a value in the parent table automatically affects any related rows in the dependent table.

***case sensitive*—**A condition in which names must be entered in a specific lower-case, upper-case, or mixed-case format to be valid.

***cast*—**The conversion between different data types that represent the same data.

***CHAR*—**A column data type that stores character strings with a user-specified length. SQLBase stores CHAR columns as variable-length strings. Also called VARCHAR.

***character*—**A letter, digit, or special character (such as a punctuation mark) that is used to represent data.

***character string*—**A sequence of characters treated as a unit.

***checkpoint*—**A point at which database changes older than the last checkpoint are flushed to disk. Checkpoints are needed to ensure crash recovery.

***clause*—**A distinct part of a SQL command, such as the WHERE clause; usually followed by an argument.

***client*—**A computer that accesses shared resources on other computers running as servers on the network. Also called front-end or requester.

***column*—**A data value that describes one characteristic of an entity. The smallest unit of data that can be referred to in a row. A column contains one unit of data in a row of a table. A column has a name and a data type. Sometimes called field or attribute.

***command*—**A user request to perform a task or operation. In SQLTalk, each command starts with a name, and has clauses and arguments that tailor the action that is performed. A command can include limits or specific terms for its execution, such as a query for names and addresses in a single zip code. Sometimes called statement.

***commit*—**A process that causes data changed by an application to become part of the physical database. Locks are freed after a commit (except when cursor-context preservation is on). Before changes are stored, both the old and new data exist so that changes can be stored or the data can be restored to its prior state.

***commit server*—**A database server participating in a distributed transaction, that has commit service enabled. It logs information about the distributed transaction and assists in recover after a network failure.

***composite primary key*—**A primary key made up of more than one column in a table.

***concatenated key*—**An index that is created on more than one column of a table. Can be used to guarantee that those columns are unique for every row in the table and to speed access to rows via those columns.

***concatenation*—**Combining two or more character strings into a single string.

***concurrency*—**The shared use of a database by multiple users or application programs at the same time. Multiple users can execute database transactions simultaneously without interfering with each other. The database software ensures that all users see correct data and that all changes are made in the proper order.

***configure*—**To define the features and settings for a database server or its client applications.

***connect*—**To provide a valid authorization-id and password to log on to a database.

connection handle—Used to create multiple, independent connections. An application must request a connection handle before it opens a cursor. Each connection handle represents a single transaction and can have multiple cursors. An application may request multiple connection handles if it is involved in a sequence of transactions.

consistency—A state that guarantees that all data encountered by a transaction does not change for the duration of a command. Consistency ensures that uncommitted updates are not seen by other users.

constant—Specifies an unchanging value. Also called literal.

control file—An ASCII file containing information to manage segmented load/unload files.

cooperative processing—Processing that is distributed between a client and a server in a such a way that each computer works on the parts of the application that it is best at handling.

coordinator—The application that initiates a distributed transaction.

correlated subquery—A subquery that is executed once for each row selected by the outer query. A subquery cannot be evaluated independently because it depends on the outer query for its results. Also called a repeating query. Also see subquery and outer query.

correlation name—A temporary name assigned to a table in an UPDATE, DELETE, or SELECT command. The correlation name and column name are combined to refer to a column from a specific table later in the same command. A correlation name is used when a reference to a column name could be ambiguous. Also called range variable.

crash recovery—The procedures that SQLBase uses automatically to bring a database to a consistent state after a failure.

current row—The latest row of the active result set which has been fetched by a cursor. Each subsequent fetch retrieves the next row of the active result set.

cursor—The term cursor refers to one of the following definitions:

- The position of a row within a result table. A cursor is used to retrieve rows from the result table. A named cursor can be used in the CURRENT OF clause or the ADJUSTING clause to make updates or deletions.
- A work space in memory that is used for gaining access to the database and processing a SQL command. This work space contains the return code, number of rows, error position, number of select list items, number of bind variables, rollback flag, and the command type of the current command.

- When the cursor belongs to an explicit connection handle that is created using the SQL/API function call *sqlcch* or the SQLTalk BEGIN CONNECTION command, it identifies a task or activity within a transaction. The task or activity can be compiled/executed independently within a single connection thread.

  Cursors can be associated with specific connection handles, allowing multiple transactions to the same database within a single application. When this is implemented, only one user is allowed per transaction.

- When a cursor belongs to an implicit connection handle created using the SQL/API function call *sqlcnc* or *sqlcnr*, or the SQLTalk CONNECT command, the cursor applies to an application in which you are connecting the cursor to a specific database that belongs to a single transaction.

cursor-context preservation—A feature of SQLBase where result sets are maintained after a COMMIT. A COMMIT does not destroy an active result set (cursor context). This enables an application to maintain its position after a COMMIT, INSERT, or UPDATE. For fetch operations, locks are kept on pages required to maintain the fetch position.

cursor handle—Identifies a task or activity within a transaction. When a connection handle is included in a function call to open a new cursor, the function call returns a cursor handle. The cursor handle can be used in subsequent SQL/API calls to identify the connection thread. A cursor handle is always part of a specific transaction and cannot be used in multiple transactions. However, a cursor handle can be associated with a specific connection handle. The ability to have multiple transactions to the same database within a single application is possible by associating cursor handles with connection handles.

Cursor Stability (CS)—The isolation level where a page acquires a shared lock on it only while it is being read (while the cursor is on it). A shared lock is dropped as the cursor leaves the page, but an exclusive lock (the type of lock used for an update) is retained until the transaction completes. This isolation level provides higher concurrency than Read Repeatability, but consistency is lower.

data dictionary—See system catalog.

data type—Any of the standard forms of data that SQLBase can store and manipulate. An attribute that specifies the representation for a column in a table. Examples of data types in SQLBase are CHAR (or VARCHAR), LONG VARCHAR (or LONG), NUMBER, DECIMAL (or DEC), INTEGER (or INT), SMALLINT, DOUBLE PRECISION, FLOAT, REAL, DATETIME (or TIMESTAMP), DATE, TIME.

***database*—**A collection of interrelated or independent pieces of information stored together without unnecessary redundancy. A database can be accessed and operated upon by client applications such as SQLTalk.

***database administrator (DBA)*—**A person responsible for the design, planning, installation, configuration, control, management, maintenance, and operation of a DBMS and its supporting network. A DBA ensures successful use of the DBMS by users.

A DBA is authorized to grant and revoke other users' access to a database, modify database options that affect all users, and perform other administrative functions.

***database area*—**A database area corresponds to a file. These areas can be spread across multiple disk volumes to take advantage of parallel disk input/output operations.

***database management system (DBMS)*—**A software system that manages the creation, organization, and modification of a database and access to data stored within it. A DBMS provides centralized control, data independence, and complex physical structures for efficient access, integrity, recovery, concurrency, and security.

***database object*—**A table, view, index, synonym or other object created and manipulated through SQL.

***database server*—**A DBMS that a user interacts with through a client application on the same or a different computer. Also called back-end or engine.

***DATE*—**A column data type in SQL that represents a date value as a three-part value (day, month, and year).

***date/time value*—**A value of the data type DATE, TIME, or TIMESTAMP.

***DCL (Data Control Language)*—**SQL commands that assign database access privileges and security such as GRANT and REVOKE.

***DDL (Data Definition Language)*—**SQL commands that create and define database objects such as CREATE TABLE, ALTER TABLE, and DROP TABLE.

***deadlock*—**A situation when two transactions, each having a lock on a database page, attempt to acquire a lock on the other's database page. One type of deadlock is where each transaction holds a shared lock on a page and each wishes to acquire an exclusive lock. Also called deadly embrace.

***DECIMAL*—**A column data type that contains numeric data with a decimal point. Also called DEC.

***default*—**An attribute, value, or setting that is assumed when none is explicitly specified.

***delimited identifier*—**An identifier enclosed between two double quote characters (“). because it contains reserved words, spaces, or special characters.

***delimiter*—**A character that groups or separates items in a command.

dependent object—An object whose existence depends on another object.

For example, if a stored procedure calls an external function, the stored procedure is the dependent object of the external function, since its existence depends on the external function.

***dependent table*—**the table containing the foreign key.

determinant object—An object that determines the existence of another object.

For example, if a stored procedure calls an external function, the external function is the determinant object, since it determines the existence of the stored procedure.

***dirty page*—**A database page in cache that has been changed but has not been written back to disk.

***distributed database*—**A database whose objects reside on more than one system in a network of systems and whose objects can be accessed from any system in the network.

***distributed transaction*—**Coordinates SQL statements among multiple databases that are connected by a network.

***DLL (Dynamic Link Library)*—**A program library written in C or assembler that contains related modules of compiled code. The functions in a DLL are not read until run-time (dynamic linking).

***DML (Data Manipulation Language)*—**SQL commands that change data such as INSERT, DELETE, UPDATE, COMMIT, and ROLLBACK.

***DOUBLE PRECISION*—**A column data type that stores a floating point number.

***DQL (Data Query Language)*—**The SQL SELECT command, which lets a user request information from a database.

***duplicates*—**An option used when creating an index for a table that specifies whether duplicate values are allowed for a key.

***embedded SQL*—**SQL commands that are embedded within a program, and are prepared during precompilation and compilation before the program is executed. After a SQL command is prepared, the command itself does not change (although values of host variables specified within the command can change). Also called static SQL.

***engine*—**See database server.

***entity*—**A person, place, or thing represented by a table. In a table, each row represents an entity.

***equijoin*—**A join where columns are compared on the basis of equality, and all the columns in the tables being joined are included in the results.

***Ethernet*—**A LAN with a bus topology (a single cable not connected at the ends). When a computer wants to transmit, it first checks to see if another computer is transmitting. After a computer transmits, it can detect if a collision has happened. Ethernet is a broadcast network and all computers on the network hear all transmissions. A computer selects only those transmissions addressed to it.

***exclusive lock (X-lock)*—**An exclusive lock allows only one user to have a lock on a page at a time. An exclusive lock prevents another user from acquiring a lock until the exclusive lock is released. Exclusive locks are placed when a page is to be modified (such as for an UPDATE, INSERT, or DELETE).

An exclusive lock differs from a shared lock because it does not permit another user to place any type of lock on the same data.

***expression*—**An item or a combination of items and operators that yield a single value. Examples are column names which yield the value of the column in successive rows, arithmetic expressions built with operators such as + or - that yield the result of performing the operation, and functions which yield the value of the function for its argument.

***extent page*—**A database page used when a row is INSERTed that is longer than a page or when a row is UPDATEd and there is not enough space in the original page to hold the data.

external function—A user-defined function that resides in an "external" DLL (Dynamic Link Library) invoked within a SQLBase stored procedure.

***event*—**See timer event.

***field*—**See column.

***file server*—**A computer that allows network users to store and share information.

***FLOAT*—**A column data type that stores floating point numbers.

***floating point*—**A number represented as a number followed by an exponent designator (such as 1.234E2, -5.678E2, or 1.234E-2). Also called E-notation or scientific notation.

***foreign key*—**Foreign keys logically connect different tables. A foreign key is a column or combination of columns in one table whose values match a primary key in another table. A foreign key can also be used to match a primary key within the same table.

***front-end*—**See client.

***function*—**A predefined operation that returns a single value per row in the output result table.

***grant*—**That act of a system administrator to permit a user to make specified use of a database. A user may be granted access to an entire database or specific portions, and have unlimited or strictly-limited power to display, change, add, or delete data.

***GUI (Graphical User Interface)*—**A graphics-based user interface with windows, icons, pull-down menus, a pointer, and a mouse. Microsoft Windows and Presentation Manager are examples of graphical user interfaces.

***history file*—**Contains previous versions of changed database pages. Used when read-only (RO) isolation level is enabled.

***host language*—**A program written in a language that contains SQL commands.

***identifier*—**The name of a database object.

***index*—**A data structure associated with a table used to locate a row without scanning an entire table. An index has an entry for each value found in a table's indexed column or columns, and pointers to rows having that value. An index is logically ordered by the values of a key. Indexes can also enforce uniqueness on the rows in a table.

***INTEGER*—**A column data type that stores a number without a decimal point. Also call INT.

***isolation level*—**The extent to which operations performed by one user can be affected by (are isolated from) operations performed by another user. The isolation levels are Read Repeatability (RR), Cursor Stability (CS), Release Locks (RL), and Read Only (RO).

***join*—**A query that retrieves data from two or more tables. Rows are selected when columns from one table match columns from another table. See also Cartesian product, self-join, equijoin, natural join, theta join, and outer join.

***key*—**A column or a set of columns in an index used to identify a row. A key value can be used to locate a row.

***keyword*—**One of the predefined words in a command language.

***local area network (LAN)*—**A collection of connected computers that share data and resources, and access other networks or remote hosts. Usually, a LAN is geographically confined and microcomputer-based.

***lock*—**To temporarily restrict other usersÕ access to data to maintain consistency. Locking prevents data from being modified by more than one user at a time and prevents data from being read while being updated. A lock serializes access to data and prevents simultaneous updates that might result in inconsistent data. See shared lock (S-lock) and exclusive lock (X-lock).

***logical operator*—**A symbol for a logical operation that connects expressions in a WHERE or HAVING clause. Examples are AND, OR, and NOT. An expression formed with logical operators evaluates to either TRUE or FALSE. Logical operators define or limit the information sought. Also called Boolean operator.

***LONG VARCHAR*—**In SQL, a column data type where the value can be longer than 254 bytes. The user does not specify a length. SQLBase stores LONG VARCHAR columns as variable-length strings. Also called LONG.

***mathematical function*—**An operation such as finding the average, minimum, or maximum value of a set of values.

***media recovery*—**Restoring data from backup after events such as a disk head crash, operating system crash, or a user accidentally dropping a database object.

***message buffer*—**The input message buffer is allocated on both the client computer and the database server. The database server builds an input message in this buffer on the database server and sends it across the network to a buffer on the client. It is called an input message buffer because it is input from the client's point of view.

The out put message buffer is allocated on both the client computer and on the database server. The client builds an output message in this buffer and sends it to a buffer on the database server. It is called an output message buffer because it is output from the client's point of view.

***modulo*—**An arithmetic operator that returns an integer remainder after a division operation on two integers.

***multi-user*—**The ability of a computer system to provide its services to more than one user at a time.

***natural join*—**An equijoin where the value of the columns being joined are compared on the basis of equality. All the columns in the tables are included in the results but only one of each pair of joined columns is included.

NDS (NetWare Directory Services)—A network-wide directory included with NetWare 4.x, that provides global access to all network resources, regardless of their physical location. The directory is accessible from multiple points by network users, services and applications.

***nested query*—**See subquery.

***NetWare*—**The networking components sold by Novell. NetWare is a collection of data link drivers, a transport protocol stack, client computer software, and the NetWare server operating system. NetWare runs on Token Ring, Ethernet, and ARCNET.

***NetWare 386*—**A server operating system from Novell for computers that controls system resources on a network.

***NLM (NetWare Loadable Module)*—**An NLM is a NetWare program that you can load into or unload from server memory while the server is running. When loaded, an NLM is part of the NetWare operating system. When unloaded, an NLM releases the memory and resources that were allocated for it.

***null*—**A value that indicates the absence of data. Null is not considered equivalent to zero or to blank. A value of null is not considered to be greater than, less than, or equivalent to any other value, including another value of null.

***NUMBER*—**A column data type that contains a number, with or without a decimal point and a sign.

***numeric constant*—**A fixed value that is a number.

***ODBC*—**The Microsoft Open DataBase Connectivity (ODBC) standard, which is an application programming interface (API) specification written by Microsoft. It calls for all client applications to write to the ODBC standard API and for all database vendors to provide support for it. It then relies on third-party database drivers or access tools that conform to the ODBC specification to translate the ODBC standard API calls generated by the client application into the database vendor's proprietary API calls.

***operator*—**A symbol or word that represents an operation to be performed on the values on either side of it. Examples of operators are arithmetic (+, -, *, /), relational (=, !=, >, <, >=, <=), and logical (AND, OR, NOT).

***optimization*—**The determination of the most efficient access strategy for satisfying a database access.

***outer join*—**A join in which both matching and non-matching rows are returned. Each preserved row is joined to an imaginary row in the other table in which all the fields are null.

***outer query*—**When a query is nested within another query, the main query is called the outer query and the inner query is called the subquery. An outer query is executed once for each row selected by the subquery. A subquery cannot be evaluated independently but that depends on the outer query for its results. Also see subquery.

***page*—**The physical unit of disk storage that SQLBase uses to allocate space to tables and indexes.

***parent table*—**The table containing the primary key.

***parse*—**To examine a command to make sure that it is properly formed and that all necessary information is supplied.

***partitioning*—**A method of setting up separate user areas to maximize disk space. Databases can be stretched across several different network partitions.

***password*—**A sequence of characters that must be entered to connect to a database. Associated to each password is an authorization-id.

***picture*—**A string of characters used to format data for display.

***precedence*—**The default order in which operations are performed in an expression.

***precision*—**The maximum number of digits in a column.

***precompilation*—**Processing of a program containing SQL commands or procedures that takes place before compilation. SQL commands are replaced with statements that are recognized by the host language compiler. Output from precompilation includes source code that can be submitted to the compiler.

***predicate*—**An element in a search condition that expresses a comparison operation that states a set of criteria for the data to be returned by a query.

***primary key*—**The columns or set of columns that are used to uniquely identify each row in a table. All values for a key are unique and non-null.

***privilege*—**A capability given to a user to perform an action.

***procedure*—**A named set of SAL or SQL statements that can contain flow control language. You compile a procedure for immediate and/or later execution.

***query*—**A request for information from a database, optionally based on specific conditions. For example, a request to list all customers whose balance is greater than $1000. Queries are issued with the SELECT command.

***Read Only (RO)*—**The isolation level where pages are not locked, and no user has to wait. This gives the user a snapshot view of the database at the instant that the transaction began. Data cannot be updated while in the read-only isolation level.

***Read Repeatability (RR)*—**The isolation level where if data is read again during a transaction, it is guaranteed that those rows would not have changed. Rows referenced by the program cannot be changed by other programs until the program reaches a commit point. Subsequent queries return a consistent set of results (as though changes to the data were suspended until all the queries finished). Other users will not be able to update any pages that have been read by the transaction. All shared locks and all exclusive locks are retained on a page

until the transaction completes. Read repeatability provides maximum protection from other active application programs. This ensures a high level of consistency, but lowers concurrency. SQLBase default isolation level.

***REAL*—**A column data type that stores a single-precision number.

***record*—**See row.

***recovery*—**Rebuilding a database after a system failure.

***referential cycle*—**Tables which are dependents of one another.

***referential integrity*—**Guarantees that all references from one database table to another are valid and accurate. Referential integrity prevents problems that occur because of changes in one table which are not reflected in another.

***relation*—**See table.

***relational database*—**A database that is organized and accessed according to relationships between data items. A relational database is perceived by users as a collection of tables.

***relational operator*—**A symbol (such as =, >, or <) used to compare two values. Also called comparison operator.

***Release Locks (RL)*—**With the Cursor Stability isolation level, when a reader moves off a database page, the shared lock is dropped. However, if a row from the page is still in the message buffer, the page is still locked.

In contrast, the Release Lock (RL) isolation level increases concurrency. By the time control returns to the application, all shared locks have been released.

***repeating query*—**See correlated subquery.

***requester*—**See client.

***restore*—**Copying a backup of a database or its log files to a database directory.

***restriction mode*—**In restriction mode, the result set of one query is the basis for the next query. Each query further restricts the result set. This continues for each subsequent query.

***result set mode*—**Normally, result table rows are displayed and scrolled off the screen. In result set mode, the rows of the result table are available for subsequent scrolling and retrieval.

***result table*—**The set of rows retrieved from one or more tables or views during a query. A cursor allows the rows to be retrieved one by one.

***revoke*—**The act of withdrawing a user's permission to access a database.

***rollback*—**To restore a database to the condition it was in at its last COMMIT. A ROLLBACK cancels a transaction and undoes any changes that it made to the database. All locks are freed unless cursor-context preservation is on.

***rollforward*—**Reapplying changes to a database. The transaction log contains the entries used for rollforward.

***router*—**A client application talks to a SQLBase server through a router program. The router enables a logical connection between a client and the server. Once this connection is established on the LAN, the client application uses the router program to send SQL requests to the server and to receive the results.

***row*—**A set of related columns that describe a specific entity. For example, a row could contain a name, address, telephone number. Sometimes called record or tuple.

***ROWID*—**A hidden column associated with each row in a SQLBase table that is an internal identifier for the row. The ROWID can be retrieved like any other column.

***ROWID validation*—**A programming technique that ensures that a given row that was SELECTed has not been changed or deleted by another user during a session. When a row is updated, the ROWID is changed.

***SAP (Service Advertisement Protocol)*—**A NetWare protocol that resources (such as database servers) use to publicize their services and addresses on a network.

***savepoint*—**An intermediate point within a transaction to which a user can later ROLLBACK to cancel any subsequent commands, or COMMIT to complete the commands.

***scale*—**The number of digits to the right of the decimal point in a number.

***search condition*—**A criterion for selecting rows from a table. A search condition appears in a WHERE clause and contains one or more predicates.

***search*—**To scan one or more columns in a row to find rows that have a certain property.

***self-join*—**A join of a table with itself. The user assigns the two different correlation names to the table that are used to qualify the column names in the rest of the query.

***self-referencing table*—**A table that has foreign and primary keys with matching values within the same table.

***server*—**A computer on a network that provides services and facilities to client applications.

***shared cursor*—**A handle that is used by two or more Windows applications.

shared lock (S-lock)—A shared lock permits other users to read data, but not to change it. A shared lock lets users read data concurrently, but does not let a user acquire an exclusive lock on the data until all the users' shared locks have been released. A shared lock is placed on a page when the page is read (during a SELECT). At a given time, more than one user can have a shared lock placed on a page. The timing of the release of a shared lock depends on the isolation level.

A shared lock differs from an exclusive lock because it permits more than one user to place a lock on the same data.

single-user—A computer system that can only provide its services to one user at a time.

SMALLINT—A column data type that stores numbers without decimal points.

socket—An identifier that Novell's IPX (Internetwork Packet Exchange) uses to route packets to a specific program.

SPX (Sequenced Packet Exchange)—A Novell communication protocol that monitors network transmissions to ensure successful delivery. SPX runs on top of Novell's IPX (Internetwork Packet Exchange).

SQL (Structured Query Language)—A standard set of commands used to manage information stored in a database. These commands let users retrieve, add, update, or delete data. There are four types of SQL commands
Data Definition Language (DDL), Data Manipulation Language (DML), Data Query Language (DQL), and Data Control Language (DCL). SQL commands can be used interactively or they can be embedded within an application program. Pronounced ess-que-ell or sequel.

SQLBase—A relational DBMS that lets users access, create, and update data.

SQLTalk—SQLTalk is an interactive user interface for SQLBase that is used to manage a relational database. SQLTalk has a complete implementation of SQL and many extensions. SQLTalk is a client application.

static SQL—See embedded SQL.

statistics—Attributes about tables such as the number of rows or the number of pages. Statistics are used during optimization to determine the access path to a table.

storage group—A list of database areas. Storage groups provide a means to allow databases or tables to be stored on different volumes.

stored procedure—A precompiled procedure that is stored on the backend for future execution.

string delimiter—A symbol used to enclose a string constant. The symbol is the single quote (').

***string*—**A sequence of characters treated as a unit of data.

***subquery*—**A SELECT command nested within the WHERE or HAVING clause of another SQL command. A subquery can be used anywhere an expression is allowed if the subquery returns a single value. Sometimes called a nested query. Also called subselect. See also correlated subquery.

synonym—A name assigned to a table, view, external function that may be then used to refer to it. If you have access to another user's table, you may create a synonym for it and refer to it by the synonym alone without entering the user's name as a qualifier.

***syntax*—**The rules governing the structure of a command.

***system catalog*—**A set of tables SQLBase uses to store metadata. System catalog tables contain information about database objects, privileges, events, and users. Also called data dictionary.

***system keywords*—**Keywords that can be used to retrieve system information in commands.

***table*—**The basic data storage structure in a relational database. A table is a two-dimensional arrangement of columns and rows. Each row contains the same set of data items (columns). Sometimes called a relation.

***table scan*—**A method of data retrieval where a DBMS directly searches all rows in a table sequentially instead of using an index.

***theta join*—**A join that uses relational operators to specify the join condition.

***TIME*—**A column data type in the form of a value that designates a time of day in hours, minutes, and possibly seconds (a two- or three-part value).

***timeout*—**A time interval allotted for an operation to occur.

***TIMESTAMP*—**A column data type with a seven-part value that designates a date and time. The seven parts are year, month, day, hour, minutes, seconds, and microseconds (optional). The format is

yyyy-mm-dd-hh.mm.ss.nnnnnn

***timer event*—**Executes a procedure at a predetermined time. You can optionally repeat the timer event at specified intervals.

***token*—**A character string in a specific format that has some defined significance in a SQL command.

***Token-Ring*—**A LAN with ring topology (cable connected at the ends). A special data packet called a token is passed from one computer to another. When a computer gets the token, it can attach data to it and transmit. Each computer

passes on the data until it arrives at its destination. The receiver marks the message as being received and sends the message on to the next computer. The message continues around the ring until the sender receives it and frees the token.

***tokenized error message*—**An error message formatted with tokens in order to provide users with more informational error messages. A tokenized error message contains one or more variables that SQLBase substitutes with object names (tokens) when it returns the error message to the user.

***transaction*—**A logically-related sequence of SQL commands that accomplishes a particular result for an application. SQLBase ensures the consistency of data by verifying that either all the data changes made during a transaction are performed, or that none of them are performed. A transaction begins when the application starts or when a COMMIT or ROLLBACK is executed. The transaction ends when the next COMMIT or ROLLBACK is executed. Also called logical unit of work.

***transaction log*—**A collection of information describing the sequence of events that occur while running SQLBase. The information is used for recovery if there is a system failure. A log includes records of changes made to a database. A transaction log in SQLBase contains the data needed to perform rollbacks, crash recovery, and media recovery.

***trigger*—**Activates a stored procedure that SQLBase automatically executes when a user attempts to change the data in a table, such as on a DELETE or UPDATE command.

***two-phase commit*—**The protocol that coordinates a distributed transaction commit process on all participating databases.

***tuple*—**See row.

***unique key*—**One or more columns that must be unique for each row of the table. An index that ensures that no identical key values are stored in a table.

***username*—**See authorization-id.

***value*—**Data assigned to a column, a constant, a variable, or an argument.

***VARCHAR*—**See CHAR.

***variable*—**A data item that can assume any of a given set of values.

***view*—**A logical representation of data from one or more base tables. A view can include some or all of the columns in the table or tables on which it is defined. A view represents a portion of data generated by a query. A view is derived from a base table or base tables but has no storage of its own. Data for a view can be updated in the same manner as for a base table. Sometimes called a virtual table.

***wildcard*—**Characters used in the LIKE predicate that can stand for any one character (the underscore _) or any number of characters (the percent sign%) in pattern-matching.

***Windows*—**A graphical user interface from Microsoft that runs under DOS.

With Windows, commands are organized in lists called menus. Icons (small pictures) on the screen represent applications. A user selects a menu item or an icon by pointing to it with a mouse and clicking.

Applications run in windows that can be resized and relocated. A user can run two or more applications at the same time and can switch between them. A user can run multiple copies of the same application at the same time.

***write-ahead log (WAL)*—**A transaction logging technique where transactions are recorded in a disk-based log before they are recorded in the physical database. This ensures that active transactions can be rolled back if there is a system crash.

Index

C

E

F

G

H

I

L

M

N

O

P

Q

R

S

T