Introduction to Borland Database Engine

This is the complete user guide and reference for Borland® Database Engine.

Borland Database Engine (BDE) is the proven 32-bit Windows-based core database engine and connectivity software behind Delphi®, Delphi Client/Server®, IntraBuilder®, Paradox® for Windows, and Visual dBASE® for Windows. BDE offers a rich and robust set of features to assist developers of client-server applications.

Architecture: The BDE database-driver architecture includes numerous shared services utilized by database drivers and other functions. The included set of database drivers enables consistent access to standard data sources: Paradox, dBASE, FoxPro, Access, and text databases. You can add Microsoft ODBC drivers as needed to the built-in ODBC socket. Optionally, Borland's SQL Links product provides access to a range of SQL servers, including Informix, DB2, InterBase, Oracle, and Sybase. Together with its database drivers and consistent API, BDE gives Microsoft Windows 95 and Windows NT application developers a direct, clean, separate, and shared high-level access to multiple data sources.

Object-orientation: BDE is object-oriented in design. At run time, application developers interact with BDE by creating various BDE objects. These run-time objects are then used to manipulate database entities, such as tables and queries. BDE's extensive application program interface (API) provides direct C and C++ optimized access to the database engine, as well as BDE's built-in drivers for dBASE, Paradox, FoxPro, Access, and text databases.

Guide to programming with BDE: The core database engine files consist of a set of DLLs that are fully re-entrant and thread-safe. Included with BDE are a helpful set of supplemental tools and examples with sample code to get you started. See Introduction to BDE programming for detailed examples of each stage of the programming process, including a BDE template program that you can copy and use as a functional framework for building your own BDE applications. Also, in the Function reference of this guide you will find examples illustrating the use of each function in both C and Delphi (Pascal) languages.

What's new: See What's new for BDE 5? for an overview of BDE 5 features and important changes from previous versions of BDE.

Configuration: You configure the BDE system using the BDE Administrator (BDEADMIN.EXE). BDE provides flexible and powerful configuration management capabilities.

Local SQL: Included with BDE is Borland's Local SQL, a subset of ANSI-92 SQL enhanced to support Paradox and dBASE (standard) naming conventions for tables and fields (called "columns" in SQL). Local SQL lets you use SQL to query "local" standard database tables that do not reside on a database server as well as "remote" SQL servers. Local SQL is also essential to make multi-table queries across both local standard tables and those on remote SQL servers.

(Note: You might occasionally encounter internal references to the older name for the BDE API: the "Integrated Database Application Program Interface" or "IDAPI".)

{button ,AL(`intro')} Other topics in this introduction to BDE

{button ,AL(`bdedocs')} Other BDE online documentation

Features

BDE offers these features:

- A uniform and consistent API to access multiple database formats including dBASE, Paradox, Text, FoxPro, Access, InterBase, Oracle, Sybase, and Microsoft SQL Server as well as any Microsoft Open Database Connectivity (ODBC) data source. Developers can easily change where and in what format the data resides without having to rewrite their application.
- BDE is ideally suited for client/server applications because it enables application developers to access both local and server data, which allows easy upsizing of the applications.
- BDE gives applications direct and live access to data sources without the need for importing and exporting.
- BDE is the highest performance database engine for Paradox and dBASE file formats.
- BDE serves the needs of developers coming from two different paradigms: set and navigational. BDE allows access to data using ISAM (Indexed Sequential Access Method, which is also used by the Paradox Engine), SQL (Structured Query Language), or QBE (Query by Example).
- BDE is a data-integration engine, providing services that can be shared across different database servers. This includes the ability to easily copy data from one format to another, as well as linking and querying data across formats. For example, you can do a query across a dBASE and an Oracle table, copy records from InterBase to Paradox, or establish one-to-many relationships between an InterBase and an Oracle table.
- BDE's query engines provides a consistent query language for SQL and QBE and set-oriented access. You can define and access data in both SQL-based servers and file-based databases.
- BDE supports full 32-bit functionality, including multi-threading, preemptive multi-tasking, universal naming convention (UNC), and long filenames. You can run multiple queries in the background while using BDE features in the foreground. Multiple applications can run simultaneously and can access the same database files. You can access servers by pathname rather than by drive letters. You can give BDE files long, descriptive names--up to 260 characters--that may contain spaces.

{button ,AL(`intro')} Other topics in this introduction to BDE

BDE components

This table is a high-level overview of Borland Database Engine software components.

Component	Description
Core BDE files	Core .DLL files are the essential files that make up the Borland Database Engine.
BDE API functions	The Borland Database Engine API, a set of function calls for managing the environment, configuration, session, error handling, record and table locking, cursor, index, query, transaction, database, table, and schema operations.
SnipIt Code Viewer	Use to display and run precompiled and linked code segments that demonstrate the use of BDE functions.
BDE Administrator	Convenient tool for configuring BDE: to register drivers and aliases, set date format options, and customize BDE drivers. (BDEADMIN.EXE)
Database Desktop	Use to view, create, and restructure tables and run queries with a graphic interface.
Query engines	BDE's shared high-performance SQL database query engine supports Structured Query Language (SQL) with extensive ANSI SQL 92. BDE's QBE query engine supports Query by Example (QBE) languages.
Database drivers	Five standard database drivers (Paradox, dBASE, FoxPro, Access, and Text) are included with BDE.
Optional drivers	Other database drivers may be added as needed, including drivers for InterBase servers and native SQL drivers for DB2, Informix, Oracle, Sybase, and Microsoft SQL Server.
ODBC connectivity	ODBC connectivity that allows access to any data source for which an ODBC driver is available. (BDE applications get the benefits of BDE even when using an ODBC driver.)
DBPing	Allows you to connect to SQL databases.
Tools & examples	A collection of tools to ease the task of application development. A series of sample programs demonstrate the use of BDE functions. (In addition to these, you will find many examples, both C and Delphi style, throughout this online guide.)

{button ,AL(`intro')} Other topics in this Introduction to BDE

Core Borland Database Engine files

The core BDE files include:

Core File	Description
DBCLIENT.DLL	BDE DataSet clients DLL.
IDAPI32.DLL	Primary BDE DLL.
BLW32.DLL	International Language Driver support functions.
IDBAT32.DLL	Contains the batch operations.
IDQBE32.DLL	Local QBE Query Engine.
IDSQL32.DLL	SQL Query Engine.
IDASCI32.DLL	ASCII Text driver.
IDPDX32.DLL	Paradox Driver.
IDDBAS32.DLL	dBASE driver.
IDODBC32.DLL	ODBC Socket Driver (allows the use of any ODBC 3.0 driver).
IDR20009.DLL	Resource file for error messages.
IDDAO32.DLL	Access Driver for Access 95 and Jet Engine 3.0.
IDDA3532.DLL	Access Driver for Access 97 and Jet Engine 3.5.
IDDR32.DLL	Data Repository.
BDEADMIN.EXE	BDE Administrator utility for managing configuration information stored in the Windows Registry and aliases in the IDAPI.CFG.
BDEADMIN.HLP	Help file for BDE Administrator.
BDEADMIN.CNT	Table of contents file for BDEADMIN.HLP. This must remain in same directory with BDEADMIN.HLP.
BDE32.HLP	The online reference for 32-bit BDE.
BDE32.CNT	Table of contents file for BDE32.HLP. This should remain in same directory with BDE32.HLP.
IDAPI.CFG	File containing application-specific BDE configuration information, primarily database aliases.
*.BTL	Ctype information (casing, soundex, etc.)
CHARSET.CVB	Character set conversion.

{button ,AL(`intro')} Other topics in this introduction to BDE

Tools and examples

The Borland Database Engine includes a number of supplemental tools and examples that simplify the job of developing applications with BDE.

Tool	Description
Database Desktop	Simple user interface for viewing and creating tables.
BDE Administrator	Tool for managing driver and system configuration.
Default configuration file	Depending on whether you save the settings in the BDE Administrator in 32-bit or composite 16-bit/32-bit format, some or all configuration information may be stored in the default configuration file.
BDE32.TOK	BDE syntax highlighting file for the BC 5 IDE.
BDE32.HLP	BDE WinHelp File for 32-bit systems only.
DBPing	Connection testing utility.
Query	Dynamic SQL and QBE tool.

Example File	Description
SNIPIT	60 simple examples written in C. Range from basic to advanced concepts.
INVENTORY	Simple inventory example, works on Paradox tables. Written in C.
ADDRESS	Simple AddressBook example. Works with any table type. Written in C.
TABLES	Sample tables.
Template program	A complete basic BDE program written in C and suitable for use as a template for structuring your own programs.
Chk function	The complete code for the Chk function, which returns more complete error information about BDE functions than would be returned by the standard error string.

See Introduction to BDE programming for detailed examples of each stage of the programming process.

Also see the Function reference section of this guide for examples illustrating the use of each function in both C and Delphi (Pascal) languages.

{button ,AL(`intro')} Other topics in this introduction to BDE

Initialization

You should be aware of how BDE is loaded at startup. This is important if you have other versions or multiple copies of BDE on your system.

The search algorithm for loading the BDE dll, Idapi32.dll, is:

1 Current directory (might be different from applications startup directory!).
  If not found, then:

2 BDE path registry entry:

```
HKEY_LOCAL_MACHINE/ SOFTWARE/ Borland/
Database Engine/ DLLPATH/xxxxx
```

  If not found, then:

3 LoadLibrary algorithm. Application's startup directory.

  1 Current directory. If not found, then

  2 System directory. If not found, then

  3 Windows directory. If not found, then

  4 PATH environment

Loading driver dll's follows the same pattern, except the first directory to be searched is the directory where Idapi32.dll loaded (replaces step1).

Note: Loading from a current directory of an application might be useful in certain situations, but it effectively prevents other BDE applications from running simultaneously, because they would likely find another Idapi32.dll and fail at initialization time (DbiInit) with DBIERR_MULTIPLEIDAPI.

Shared memory loading addresses for DLLs

BDE reserves certain preferred memory addresses for use by its DLLs. In most cases, if a DLL cannot be loaded at its preferred address, it will load at some other address determined by the system.

However the DLLs listed below must be loaded at the same preferred address in all applications using BDE. The native BDE drivers reserve the following addresses:

BDE DLLs	Addresses
IDAPI32.DLL	0x4BDE0000
IDPDX32.DLL	0x4CDE0000
IDDBAS32.DLL	0x4DDE0000
IDASCI32.DLL	0x4EDE0000

If these memory locations are already in use by other applications, the BDE DLLs assigned to those locations might not load, in which case an error message would be generated:

- If Idapi32.dll cannot load because the address space is already used, then DBIERR_CANTLOADIDAPI is returned.
- If any of the shared drivers cannot load because of a conflict, then DBIERR_CANTLOADLIBRARY is returned with the name of the driver.

SQL drivers are not shared and do not require fixed loading addresses.

Multiple Initializations and Exits

You may make multiple calls of DbiInit from within the same process, but each DbiInit should be paired with a corresponding DbiExit.

Error Recovery

In the event of a fatal application error, it is recommended to use DbiExit to shut down BDE cleanly.

In the event of a fatal BDE error, it is recommended to close down all applications using BDE.

{button ,AL(`intro')} Other topics in this introduction to BDE

Database Desktop

(\DBD\DBD32.EXE)

The Database Desktop (DBD) is basically a stripped-down version of Paradox for Windows. It lets you visually create, inspect, and modify tables. This greatly simplifies the task of creating tables in BDE.

Configuration utility

The BDE Administrator: \IDAPI\BDEADMIN.EXE

The BDE Administrator is a visual tool for managing BDE system configuration information in the Windows Registry and alias information in the BDE configuration file (IDAPI.CFG).

You can also modify system information and existing aliases by using the functions: DbiOpenCfgInfoList, DbiAddDriver, DbiDeleteDriver, DbiAddAlias, and DbiDeleteAlias.

For detailed information on the BDE Administrator and the meanings of all BDE configuration settings, see BDE Administrator Help.

For an overview from a developers' perspective and complete guidance on configuring ODBC connectivity, see Configuration management

Default configuration file

The BDE configuration file used at application startup: \IDAPI\IDAPI.CFG It is listed in the Windows Registry as CONFIGFILE01.

For example:

```
HKEY_LOCAL_MACHINE/ SOFTWARE/ Borland/
 Database Engine/CONFIGFILE01
```

You may name your configuration file anything provided that:

- it ends in ".CFG"; and
- is no more than 255 characters long, including spaces; and
- does not contain the characters:
  \ / : * ? " < > |

The configuration file always contains database aliases and the active NET DIR entry for Paradox tables in the Paradox section of the configuration file. This NET DIR setting is always active and will take precedence over any other NET DIR parameters that may exist in older 16-bit configuration files, or in the System Init section of the current configuration file, or in the Registry.

If saved in the Windows 3.11-compatible format (16-/32-bit composite), the configuration file may duplicate some of the System and Driver entries in the Registry.

The Registry includes all driver information, entries, the size of the Buffer Manager (Database Data cache), and various other system information.

For details on saving configuration information in the BDE Administrator and where and how that information is stored, see Saving configuration information

BDE32.TOK

(\BDE32\DOC\BDE32.TOK)

This file is used by the BC 5 IDE to provide syntax highlighting for BDE functions and types. The BDE32TOK.TXT file in the same directory provides information on using this file.

BDE32.HLP

(\BDE32\DOC\BDE32.HLP)

This is the WinHelp file that you are looking at right now. It contains the complete user's guide and BDE function reference. It requires the presence of WINHELP.EXE and Windows 95/NT and its associated WinHelp Contents file, BDE32.CNT.

DBPING

(\BDE32\EXAMPLES\C\DBPING\DBPING32.EXE)

This example is used to determine if the BDE can connect to a given database. Basically, this application attempts to connect to the specified alias using the DbiOpenDatabase function.

QUERY

(\BDE32\EXAMPLES\QUERY\QUERY32.EXE)

This is a basic InterActive query tool which allows the user to connect to any data source and perform *ad hoc* queries. That is, the user can type in SQL Queries or QBE Queries and see the results of the operation.

SNIPIT

(\BDE32\EXAMPLES\SNIPIT\SNIPIT32.EXE)

This example contains many simple examples on BDE. Run the program to get an idea of the examples provided.

INVENTORY

(\BDE32\EXAMPLES\C\INVENTRY\INVTRY32.EXE)

This is a simple, stand alone, C windows application using the BDE. Because this example works only with Paradox tables, it is a good example to use for people familiar with the Paradox Engine. Note that all engine code is isolated in the ENGINE.C file, so it should be easy to incorporate aspects of this program in user applications.

ADDRESS

\BDE32\EXAMPLES\C\ADDRESS\ADRESS32.EXE

An enhanced version of the sample inventory table, this example will work with all table types. This is a good example of performing basic BDE operations on a given table type (driver).

TABLES

\BDE32\EXAMPLES\TABLES

This directory contains a number of sample tables used by the SNIPIT examples.

Configuration management

BDE Administrator Help

The BDE Administrator (BDEADMIN.EXE) is a redistributable application that you use to set up and manage your application's configuration. The Configuration page contains the parameters for BDE system configuration, database aliases, database server drivers, and ODBC connectivity. The utility includes context-sensitive help to guide you in making configuration changes (BDEADMIN.HLP).

The BDE Administrator is installed with the Borland Database Engine. Assuming you have no other BDE-based applications on your workstation at installation time, the installation program sets up IDAPI.CFG as the default BDE configuration file. This means that the first time you open the BDE Administrator it will display the parameters stored in IDAPI.CFG as well as the Windows Registry.

The default configuration file, if any, is defined in the Windows Registry under:

`HKEY_LOCAL_MACHINE/SOFTWARE/Borland/Database Engine/CONFIGFILE01`

The BDE Administrator gives you the option of saving configuration information in two formats: 32-bit format and a composite 16-/32-bit format for compatibility with 16-bit BDE applications. See Saving configuration information for details about how and where configuration information may be stored.

For complete information on configuring the BDE system, managing database aliases, and configuring database server drivers, see: BDE Administrator Help

Overriding configuration file defaults

You can override the default configuration file by using the BDE function DbiInit, but only one configuration file may be used at a time. If one application is active and you attempt to override the default configuration file while a second application starts with a different configuration file, the error message DBIERR_CFGMULTIFILE is generated.

Whether you override the default configuration file by using DbiInit or not, the NET DIR entry in the Paradox section of the configuration file is always active and will take precedence over any other NET DIR parameters that may exist in older 16-bit configuration files, or in the System Init section of the current configuration file, or in the Registry. These other NET DIR entries will have no effect. To access a Paradox table on a network drive, the active NET DIR parameter in the Paradox section of the BDE configuration file must point to a network drive.

Any other information in the Drivers and System sections will be drawn from the Registry.

{button ,AL(`intro')} Other topics in this Introduction to BDE

Overview of new features and changes in this release

This section is an overview of the new features and enhancements for Borland Database Engine version 5, along with other changes since the last release, BDE version 4.51. You can get detailed descriptions of many of these features by using the jumps to other topics in this online reference.

Oracle 8 support

- Abstract Data Types (ADTs): Allow you to define your own data types or business objects for corporate business rules.
- VARRAY (variable-length arrays): Traditional scalar types and new object relational data can be saved in this persistent array type. VARRAYs can store lists of business objects, which can refer to additional ADTs.
- Nested tables: Allow repeating groups of information to be stored directly into an existing table, without requiring keysto be generated for each row of the nested table. Functions are provided for determining the number of nested rows and manipulating them.
- REFs (object pointers): References to nested objects are stored in this new data type. Resolved REF pointers bring referenced objects into the client object space for examination and modification (pinned).
- BFILE (external file references): Reference files stored external to the database, eliminating duplication of large files (such as business documents). Increases performance, provides more flexible file access, and reduces storage requirements.
- Oracle BLOBs: Multimedia applications (and other applications requiring non-structured data) can now have multiple binary objects, each up to 4Gb in size, stored in a single record of a table. Large OBject (LOB) types store locators that specify the location of LOBs stored out-of-line (storing only the locator in the data row, not the actual LOB data) or in an external file.

Microsoft Transaction Server (MTS)

- Resource pooling: Provides faster connections and reduced network load, increasing the performance and responsiveness of your database applications.
- Two-phase commit: Database applications can now perform transactions across different database servers (like Oracle, Sybase, and DB2), for both Microsoft operating systems and hetergeneous environments. Makes available a more reliable and robust transaction system for any database natively supported by the BDE.
- X/Open XA support: Open standard XA allows transaction servers (such as Oracle) to communicate with the MTS resource manager. The client application can now see a single success or failure for a package of transactions submitted to a server in a heterogeneous environment.

Architectural overview

BDE has a driver-based architecture. Each distinct database format or data source usually requires a separate BDE driver. A given driver can support a closely related family of data sources. For example, the dBASE driver supports dBASE III, dBASE IV and later, and Microsoft FoxPro version 2.0, 2.5, and 2.6 file formats.

BDE is object-oriented in design, making it easy to extend and customize. To extend BDE to access an additional database system, simply install the appropriate BDE driver or ODBC driver for that database system.

In a client/server environment, the applications and development tools reside on the client PCs, while the data source resides on the SQL server. BDE is ideally suited for a client/server environment, because it provides transparent access to both server databases and local databases on PCs.

{button ,AL(`concepts')} More basic concepts

Shared Services

BDE is based upon a software component model. To ease driver development and maximize reuse, the BDE infrastructure provides the following shared services.

Note: These shared components are mostly internal to BDE and its drivers; they are described here to help you understand the BDE architecture.

Buffer Manager

BDE's priority-based buffer manager enables all BDE drivers to share the same buffer pool. Buffers owned by different drivers can coexist in this buffer pool. BDE drivers are not required to use the common buffer manager, but using it maximizes overall system resources.

Sort Engine

BDE's high-performance sort engine is used internally by the query engine and by the Paradox and dBASE drivers.

OS services

BDE's OS services isolate the BDE environment from all OS and platform dependencies, including file I/O, network access, and OS level memory allocation. This makes BDE highly portable.

Memory manager

BDE's memory manager provides a sub-allocation service, minimizing OS overhead for small memory allocations.

BLOB cache

BDE's BLOB caching service makes BLOB access as efficient as possible, so that programmers don't need to use their own caching schemes. The BLOB cache is accessible to all BDE drivers. Multiple BLOBs can be simultaneously opened. The BLOB cache automatically overflows into a shared physical file to handle large BLOBs. The BLOB cache makes random access to BLOBS possible, eliminating the need for application developers to transfer BLOBs to files. This facility is available from BDE even when the data source/server does not provide random access to BLOBs.

Query Engine

The shared query engine supports both SQL and QBE query languages with data format independence.

SQL Generator

The query engine supports QBE as an alternate query language, which is more intuitive to end users than SQL. When the QBE query is directed toward a SQL-based server, the SQL generator module of the QBE engine translates the query into an equivalent SQL query.

Restructure

A restructure service is currently available for Paradox, dBASE, Access, and FoxPro formats. Restructuring enables the application developer to add, drop, or modify fields and drop or modify any structural aspects of a table. This module creates new tables when appropriate, translating and copying data to the new table as necessary.

Batch table functions

A set of generic batch services is available. These include copying data from one format to another, reading and writing blocks of records, and renaming tables.

Data translation service

BDE's data translation service enables many BDE functions and services to do cross-database operations. Given any two compatible formats, the data translation service calculates the optimal conversion. Data is translated from the database's native physical data format to the common BDE logical data format, and vice versa.

Linked cursors

BDE implements linked cursors to automatically support one-to-many relationships between two tables. A linked detail cursor tracks its master cursor using the join key and the records accessible by the detail cursor are constrained by the master record. Developers can use linked cursors to build sophisticated multi-table applications with little programming.

In-memory tables

In-memory tables provide efficient access to unlimited virtual memory in a table format. The sort engine uses in-memory tables to create intermediate batches. SQL drivers use in-memory tables for caching data locally. Developers can create and access in-memory tables by using BDE function calls used for accessing persistent tables. See DbiCreateInMemTable.

Note that in-memory tables cannot be:

- made permanent
- indexed
- moved in batches
- saved to disk

SQL driver services

All SQL-based drivers (including the ODBC connectivity module) are built using SQL driver services. The following driver services are included:

- Mapping navigational BDE calls to SQL, making it possible to upsize Paradox and dBASE applications transparently.
- Local caching of records, making it possible to browse on query results.
- Schema inquiry services.
- BLOB handling services that are built using the BLOB cache module.
- Debugging by using the SQL Trace facility to track SQL statements sent to servers by BDE functions.

System manager

The system manager manages all system-level resources. It loads drivers on demand and keeps track of open databases and cursors. When an application exits, the system manager frees the resources allocated to that application.

Configuration manager

The configuration manager maintains the BDE global system configuration information in the Windows Registry and application-specific information in the BDE configuration file (IDAPI.CFG). At startup time this information is used to customize the BDE environment.

The BDE functions DbiOpenCfgInfoList, DbiAddDriver, DbiDeleteDriver, DbiAddAlias, and DbiDeleteAlias give the application access to the configuration file. You use the BDE Administrator (BDEADMIN.EXE) to register drivers and aliases, set date format options, and customize BDE drivers.

Language drivers

BDE architecture incorporates language drivers to address the needs of the international market. Each language driver encodes the collating sequence, capitalization rules, and

OEM/ANSI translation rules to suit its particular language.

All the native BDE drivers and all BDE shared services support these language drivers, so that the entire BDE environment is automatically "international" enabled. No porting is necessary. Application developers can deploy applications in international markets using the same engine. See International compatibility

Resources

All resources, such as error messages, for a language are placed in a separate dynamic link library. BDE can simultaneously support resources in different languages. An application can register its language at startup time.

{button ,AL(`concepts')} More basic concepts

BDE API functions

The Borland Database Engine includes an API for directly accessing its functionality. The API consists of a set of functions that can be called from any programming language capable of calling Windows DLLs. BDE functions are optimized for calling from C or C++; however, Delphi Pascal syntax is also provided in the function reference.

Over the years, two different types of database systems have developed that traditionally supported different data access approaches:

- PC-based database systems (such as Paradox, dBASE, and B-Trieve) have supported the indexed sequential access method (ISAM) type of data access. However, these systems have supported different kinds of APIs.
- Server-based database systems (such as InterBase, Sybase, Oracle, and DB2) have supported the ANSI standard SQL language. However, an industry standard for an API is just emerging: X/Open SQL Call Level Interface (CLI). This standard addresses only SQL-based database needs, and does not fully address ISAM type data source requirements.

Unified access

BDE functions unify access to both PC-based or ISAM databases and server-based SQL databases with a consistent cursor-based API. BDE supports the basic APIs for both types of databases, extending powerful features of each type to the other. For example, BDE's navigational features are influenced by ISAM databases, and are extended to support server-based databases. Similarly, the Query portion of BDE is influenced by the SQL standard, and is extended to support ISAM databases. Support of these basic API features on both kinds of databases makes BDE unique. For example, Paradox and dBASE exploit these features to support transparent access to SQL data sources.

Through each driver, BDE gives the application developer access to the unique features of each database system, such as data types, primary indexes for Paradox tables, delete flags and expression indexes for dBASE tables, and special processing for SQL databases. For this reason, BDE functions are not a least-common-denominator API.

Purposes

For all supported databases for which a BDE native driver or an ODBC driver is available, BDE API function calls serve the following purposes:

- Opening and closing of databases
- Getting and setting properties of BDE objects: system, clients, sessions, drivers, databases, cursors, and statements
- Accessing and manipulating data stored in each database system
- Defining the structure of a database in each database system, such as creating tables and indexes
- Performing operations across database systems, such as copying and joining tables

---

{button ,AL(`concepts')} More basic concepts

BDE objects

BDE is object-oriented in design. At run time, application developers interact with BDE by creating various BDE objects. These run-time objects are then used to manipulate database entities, such as tables and queries. Programming for BDE involves interaction with the following BDE objects:

- System
- Clients
- Sessions
- Database drivers
- Databases
- Cursors
- Query statements

Each BDE object type is defined by a set of properties. Values are initially assigned to properties when the object is created. For example, the table name CUSTOMER is the value assigned to the table name property of the cursor object when the CUSTOMER table is opened with DbiOpenTable.

The BDE API interface provides a set of functions that the application developer can use to retrieve existing values of properties (DbiGetProp) and reset these values (DbiSetProp).

For a complete list of BDE object types and their properties, see Getting and Setting Properties.

For an overview of persistent objects common to most database systems, see Database Entities..

{button ,AL(`concepts')} More basic concepts

{button ,AL(`bdeobjects')} More BDE objects

System

One system object controls the resources common to all applications running on the same machine. The BDE API system object is automatically created when the first client initializes. At this time, any configurable settings, such as the maximum memory allowed for the buffer pool, are read from the Windows Registry.

{button ,AL(`bdeobjects')} More BDE objects

Clients

A new client object is created when an application calls the BDE initialization function. This first call to DbiInit is necessary before any other BDE call can be made. The client object is maintained automatically by BDE and exists mainly as a context for all the system resources used by the BDE on behalf of each client. The client object has properties which can be set, such as which language is to be used for error messages.

Database Drivers are owned by the client or the system; once a driver is loaded, all other clients registered with the BDE have access to it.

{button ,AL(`bdeobjects')} More BDE objects

Sessions

An application can maintain one or more sessions. Sessions provide the means to isolate a set of database access operations without the need to start another instance of the application. A default session is automatically created when each application initializes. The session object is a container for all other BDE run-time objects that can be created:

- Databases
- Cursors
- Query statements

If you access an object created in another session, the current session will change to the session in which the object was created. The session is also the owner of all table and record locks acquired by all objects within the session. This means that a table or record lock acquired using one cursor in a session is owned by all cursors in the session that are opened on the same table. Any of the cursors on the same table can release such a lock. Additional sessions can be created to allow for different locking contexts.

Another property of the session is the private directory, where BDE places all temporary file-based tables created on behalf of the session. In addition, the session owns two properties specific to the Paradox driver: passwords (for gaining access to password protected tables) and the network control directory, where the PDOXUSRS.NET file is located.

{button ,AL(`bdeobjects')} More BDE objects

Database drivers

Each driver is implicitly loaded by the system when an application first requests a service from that driver. At that time, any configurable settings found in the Windows Registry or the BDE configuration file (IDAPI.CFG) related to this driver are used to initialize it. Examples of configurable settings are the default table level and the language driver to be used when the table is created.

Drivers are owned by the client or the system; once a driver is loaded, all other clients registered with BDE have access to it.

The application developer can also inquire about driver capabilities, such as whether or not the driver supports transactions.

dBASE, Paradox, Access, FoxPro, and text drivers

The standard drivers for Paradox, dBASE, Access, FoxPro, and text databases are shipped with BDE.

SQL drivers

For server-based SQL database systems such as Informix, DB2, InterBase, Oracle, and Sybase separate native BDE SQL drivers are available.

ODBC drivers

Any ODBC driver can be used with BDE, because BDE has an ODBC connectivity socket. The rich features of BDE, such as navigational access to data, bi-directional cursors, and cross-database operations, are also automatically enabled even when an ODBC driver is in use. Enhanced ODBC connectivity. BDE functions like **DbiAddAlias** and **DbiOpenDatabase** automatically add ODBC drivers and data sources as BDE aliases to the active session when they aren't currently stored in the configuration file. The BDE also supports ODBC 3 drivers.

{button ,AL(`bdeobjects')} More BDE objects

Databases

A database is an organized collection of related tables. To access data in a table, the session first must gain access to the database with a DbiOpenDatabase call, which returns a database handle to the database.

Standard databases

BDE classifies file-based databases such as Paradox, dBASE, FoxPro, Access, and text as "standard" databases. Files within a standard database are normally grouped together in the current directory associated with a standard database, although an application can expand its database by referencing, by fully qualified pathname, any accessible file either locally or on the network.

SQL databases

A SQL database usually resides on a server. The client application must first log in, establishing a connection to the database server. This requires supplying the appropriate user name and password. When you call DbiOpenDatabase, BDE logs into the server and establishes a connection, just as with standard databases.

Aliases

An alias is a short name referencing a database. Database references within applications can use alias names, making your applications portable.

You can change the definition of an alias at any time by using the BDE Administrator BDEADMIN.EXE. All references to the alias within the application automatically refer to the new definition of the alias.

Aliases for standard databases

For standard databases, an alias is a name you assign as a shortcut to a directory containing the files you want to access. You can give a long path name a short alias name. When you open a database with such an alias, tables in that directory can be opened by supplying only the table name without supplying the full path.

Aliases for SQL databases

For SQL databases, properties must be defined for the alias. These properties can vary depending on the SQL driver. Alias properties can include:

- User name
- Server name
- Open mode
- Default SQL query mode
- Schema cache size
- Language driver

After a SQL database alias is established, the client application can use it the same way it uses an alias for a standard database.

{button ,AL(`bdeobjects')} More BDE objects

Cursors

BDE provides access to tables or query results through cursors. A cursor provides addressability to a collection of records one at a time. All data manipulation operations (insert, delete, update, and fetch), as well as positioning the cursor in the table (sometimes referred to as navigation) are performed with a cursor.

When the application opens a table with DbiOpenTable or executes a query, a cursor handle is returned. After the cursor handle is returned, you can use it to retrieve data stored in a table as well as information about a table. You can also obtain and set properties of this cursor. The application can close a cursor at any time with DbiCloseCursor. When the cursor is closed, the cursor handle becomes invalid. (Multiple cursors can be created on the same table.)

To access data in a table, the application opens the table and obtains a cursor handle. The table can be opened exclusively or shared. The translation mode can be specified as either xltNONE or xltFIELD. If xltNONE is specified, the data is returned from the table as the untranslated physical type (the native data type as stored by the data source). If xltFIELD is specified, the data is translated into a generic, logical type by BDE. Logical types are compatible with C language data types.

Ordered and unordered cursors

By default, the records returned by a cursor are not in any particular order. Ordered cursors can be obtained by specifying a current active index for a cursor (using DbiOpenTable or DbiSwitchToIndex). A query executed using the ORDER BY clause is also an ordered cursor.

Positioning the cursor

Whenever the application opens a cursor on a table or a query result, the resulting cursor is positioned at the beginning of the result set, before the first row, rather than on the first record of the table. This initial position enables the application to access all the records with the DbiGetNextRecord function.

At any time, the cursor can be positioned on a record or on a crack. A crack is a position between records at the beginning of the table, at the end of the table, or the place left when a record is deleted.

The possible cursor positions are

- At the beginning of the table or result set (the crack before the first record). DbiSetToBegin can be used to explicitly position the cursor here; the cursor is always positioned here when the cursor is opened.
- At the end of the table or result set (the crack after the last record). DbiSetToEnd can be used to explicitly position the cursor here.
- On a record (after a successful call to retrieve, insert, or update a record).
- On a crack between records. DbiSetToKey positions the cursor on the crack before the record of the specified key.
- The cursor is positioned on a crack if it was previously positioned on a record, and that record was deleted.

Bookmarks

A bookmark can be obtained to save the cursor's current position, so that it can be repositioned to that place later. Bookmarks can remember any position: on the current row, at the beginning or end of the table, or on a crack. A call to DbiGetBookMark saves the current position of the cursor as a bookmark. A subsequent call to DbiSetToBookMark positions the cursor to the location saved by DbiGetBookMark. Multiple bookmarks can be placed on a cursor. The positions of two bookmarks can be compared with a call to DbiCompareBookMarks.

{button ,AL(`bdeobjects')} More BDE objects

Query statements

SQL Queries and QBE Queries can be either directly executed or prepared first and then executed. When a query is prepared, BDE checks its validity; if the query is valid, BDE creates a query object and returns a query statement handle.

For a general exposition about query statements, see Querying databases

Certain properties of a query can be changed once the query handle is obtained. For example, if the query has parameter markers, the values of parameters to be used can be set prior to executing a query. See Getting and setting properties.

{button ,AL(`bdeobjects')} More BDE objects

Database entities

Database entities are persistent objects, common to most database systems, and include

- Tables
- Indexes
- Fields
- Queries
- Transactions
- Callbacks
- Cross-database operations

{button ,AL(`concepts')} More basic concepts

{button ,AL(`databaseentities')} More database entities

Tables

Data in a database is organized in tables. In BDE, a table name has meaning only within a database. Tables are accessible to the application in rows (records) and columns (fields). The rows can be ordered by an index.

To create a table, the application calls the BDE function DbiCreateTable passing the completed table descriptor structure CRTblDesc. Alternatively, tables can be created using SQL Data Definition Language (DDL).

Temporary Tables

Certain database operations create temporary tables that last only until you close them or end the BDE session. Your application can create two types of temporary tables:

- Use DbiCreateTempTable to create a temporary table, which can later be saved to disk. If the table becomes too large, it is automatically written to disk. The client application can explicitly save the temporary table to disk by calling the function DbiMakePermanent or DbiSaveChanges. For all practical purposes, these tables behave like regular tables.
- Use DbiCreateInMemTable to create a temporary table never intended to be written to disk. These tables are created by the application for gathering information that is needed temporarily during processing. These tables can be created only with logical types. These tables do not support indexes.

For detailed information, see:

- Accessing and updating tables
- Creating tables
- Modifying table structure

{button ,AL(`databaseentities')} More database entities

Indexes

An index determines the order of the records in a table. Paradox, dBASE, FoxPro, Access, and SQL database systems all let you create indexes to order records. However, there are differences in the way indexes work and the information required to define indexes in each of the database systems.

BDE supports all the native modes of indexing for Paradox, dBASE, FoxPro, Access, and SQL database systems. To enable your application to create an index, BDE provides a generic index descriptor structure, IDXDesc. IDXDesc is a union of all of the fields required to define an index for all of the supported database systems. To add an index, the application supplies the required data in IDXDesc and calls the function DbiAddIndex.

To create an index for a table, your application need only supply data in the index descriptor fields that are applicable to that particular table's database system. For example, when defining an index on an InterBase table, your application ignores fields such as *szTagName* and *bExpIdx*, which are used only in defining dBASE indexes. When required fields are not supplied, an error message is returned by the DbiAddIndex call.

Different types of indexes allowed within the database system may have different requirements. For example, when adding a dBASE maintained index, the field *szTagName* is required. Indexes can also be created using the SQL Data Definition Language.

Types of Indexes

There are three basic types of indexes:

- Traditional indexes on columns. These indexes can be single column indexes or composite indexes on more than one column.
- Expression indexes. These indexes have key values determined by an expression (not necessarily column values). Of the drivers mentioned, only dBASE currently supports expression indexes.
- Pseudo-indexes. For SQL data sources, BDE can create a pseudo-index by using one or more user-specified SQL fields to define the requested order

Characteristics of Indexes

Indexes have three other characteristics:

- Subset indexes do not index every record in a table; instead, they index only those rows that satisfy a given Boolean expression. Of the drivers mentioned, only dBASE uses subset indexes.
- Unique indexes cannot have duplicate key values.
- Indexes can be ascending or descending for drivers that support them.

Driver-Defined Index Requirements

It is important to understand that different drivers support different types and characteristics of indexes. The following sections provide a partial list of rules for the different index types and characteristics supported by each driver:

dBASE

The following rules describe how dBASE supports indexes:

- dBASE supports only expression indexes. (Single-column indexes are treated as a special case of expression indexes.)
- dBASE supports two different physical index formats: .NDX-style and .MDX-style.
- dBASE supports subset indexes in .MDX-style indexes.
- In dBASE, all maintained indexes are .MDX-style indexes.
- dBASE supports FoxPro compressed index .CDX-style indexes.
- dBASE supports Clipper index .NTX-style indexes for importing.

- dBASE does not support primary indexes (or primary keys).

Expression Indexes
When defining an index, dBASE uses expression indexes. The expression index determines how the key is computed when a record is added. Expression indexes can be simply the name of a field or they can be created from field names, operators, and functions.

Multiple indexes
Multiple indexes are stored in a single file with a .MDX or .CDX extension. dBASE stores different indexes in the same physical file. Each index in the multiple index file is called a tag. Tags are identified by the *szTagName* you assigned when you created the index.

One of the multiple index files is used to store all the maintained indexes. The name of this file is of the form <Tbl_Name>.MDX or <Tbl_Name>.CDX. This file is called the production index file; indexes in this file are always maintained.

Single indexes
The dBASE driver also supports the older style dBASE indexes called .NDX indexes. This index is stored in a file with a .NDX extension. Each such file contains only one index; this index is maintained only if the index is explicitly opened.

Paradox
The following rules describe how Paradox supports indexes:

- Paradox supports both single- and multi-column indexes.
- Paradox supports a primary key.
- Paradox supports maintained and non-maintained secondary indexes. Maintained secondary indexes are supported only if the table also has a primary index. If an index is non-maintained, it becomes out of date if any data in the table changes.
- Paradox does not support expression indexes.
- Paradox does not support subset indexes.
- Paradox supports case-sensitive/insensitive secondary indexes.
- Paradox supports descending indexes with level 6 tables.

Primary indexes
A Paradox primary key is defined as a field or group of fields whose values uniquely identify each record of a table. The fields in a key must be contiguous starting with the first field. A primary key requires a unique value for each record (row) of a table. A table’s primary key establishes the default sort order for the table. A Paradox table is sorted based on the values in the fields you define as the table's primary key. Only one record's primary key can be blank. All subsequent blanks are considered as duplicates, and records containing them are not accepted.

Secondary indexes
Paradox supports secondary indexes. A table can have more than one secondary index, and a secondary index can be a composite index. Each secondary index can be maintained or non-maintained. If it is maintained, the index is updated automatically every time the table is changed. Secondary indexes can be case-sensitive or insensitive. If it is case-sensitive, BDE differentiates between uppercase and lowercase letters as it sorts fields. **Maintained secondary indexes** are supported only if the table also has a primary key. If an index is non-maintained, it becomes out of date if any data in the table changes.

SQL
The following rules describe how SQL drivers support indexes:

- All SQL indexes are maintained.
- The rules for index creation are based on SQL server support. SQL drivers support the following indexes if they are supported by your server:
  - Single and multi-column indexes
  - Unique and non-unique indexes
  - Ascending and descending indexes
- If an index is added to any SQL table, then any cursors open on that table must be closed and reopened, to allow for possible changes in the buffer size.

{button ,AL(`databaseentities')} More database entities

Fields

Fields are columns of a table. The properties of each field in a table are defined in a field descriptor structure FLDDesc. When a table is created with DbiCreateTable, the table descriptor CRTblDesc points to an array of FLDDesc structures, each of which defines a field in the table.

Physical data types

Physical data types can vary from one data source to another. For example, floating point numbers are stored differently by Paradox, dBASE, and SQL data sources. Physical data types of one data source might not be compatible with the physical data types of other data sources to store the same data.

Logical data types

Logical data types are the generic data types used by BDE. These generic types are made interchangeable between data sources because BDE automatically translates them into the proper physical data types for each target data source.

Automatic field translation

To facilitate cross-database processing, BDE does not require your application to translate data to make it compatible with each different data source. As long as your application uses BDE logical data types, BDE handles the translation to the correct physical format for each target data source. When BDE returns data to your application, it can translate all data types as they are stored by the data source back to the generic logical data types, depending on the translate mode of the cursor, xltNONE or xltFIELD.

BDE's logical data types are compatible with standard C language data types.

You can make the application override the translation mechanism when accessing a table, so that it receives data in the physical format used by the data source.

{button ,AL(`databaseentities')} More database entities

Queries

The common query engine allows you to specify queries in either the SQL or QBE language on any available data source. Through queries, BDE allows uniform data retrieval across data sources. The local query manager enables you to join data across servers. For example, you can join Oracle to dBASE, Sybase to Paradox, or InterBase to Oracle on two different servers. To run cross-database queries, the table names in a query must be qualified by alias names. Cross-database queries are supported only with standard database handles, even if the query is targeted for SQL servers.

BDE provides a set of query interface functions so that the application developer can query tables across all accessible databases:

- DbiQAlloc obtains a new statement handle
- DbiQGetBaseDescs returns the original database, table, and field names of the fields that make up the result set of a query.
- DbiSetProp sets properties on the new statement handle, such as making the query result set updateable.
- DbiQPrepare prepares a SQL or QBE query for execution.
- DbiQSetParams sets the value of parameter markers in a prepared query before the query executes.
- DbiQExec executes a previously prepared query.
- DbiQFree frees resources acquired during preparation and execution of a query.
- DbiQExecDirect prepares and executes a SQL or QBE query.

BDE allows access to SQL, Paradox, dBASE, FoxPro, or Access data through both SQL queries, a convenient subset of the SQL language, and QBE queries, the Query By Example language defined in Paradox. For both QBE and SQL, a query can be executed as a live result set, resulting in an updateable cursor on the original table.

For detailed information on querying, see:

- Querying databases
- Querying Paradox, dBASE, FoxPro, and Access tables
- Querying different databases
- Executing queries directly
- Executing queries in stages

{button ,AL(`databaseentities')} More database entities

Transactions

A transaction is a group of related operations that must *all* be performed successfully; otherwise no change to the database takes place.

BDE supports transactions on all servers with three BDE function calls:

- DbiBeginTran
- DbiEndTran
- DbiGetTranInfo

The application calls DbiBeginTran, submits SQL statements and BDE function calls to be included in the transaction, and then calls DbiEndTran. DbiGetTranInfo returns status information about a transaction.

BDE supports local transactions for Paradox, dBASE, and Access drivers so that updates to tables can be rolled back (reverted) or committed. Without transaction support, updates would be committed immediately with no way to roll them back and applications might fail to perform updates in a consistent way.

When a local transaction is started on a standard database, updates performed against tables in that database are logged. Each log record contains the old record buffer of the record that is updated. When a transaction is active, the records with updates are locked. These locks are held until the transaction is either committed or rolled back.

- The Commit operation releases all locks that were held when that transaction was active.
- The Rollback operation reapplies the updates to the underlying tables to restore the original state of the database. Once the original state of the database is restored, the locks are released.

Note: For standard transactions there is no automatic crash recovery on DDL-related actions such as table create, restructure, index creation, table/index deletion, and so on.

For more information, see:

- Transactions on Paradox, dBASE, FoxPro, and Access is a guide to specifics.
- Cached updates offers yet another strategy for dealing with locking.
- Transactions and cached updates provides a comparative discussion.

{button ,AL(`databaseentities')} More database entities

Callbacks

Sometimes a client application needs information about the progress of a given function. For example, if a table is being restructured, certain conditions can cause records to be written to a "problems" table rather than the destination table. This situation could warrant termination of the operation, or it could require some other action. A callback enables the application to intercede and evaluate such a situation before any action is taken by BDE. The application registers the callback in advance by calling DbiRegisterCallBack.

After a callback is registered, the occurrence of the specified event triggers the database engine to call the callback function, which in turn alerts the application that the event has occurred. The callback then awaits further instructions from the application.

The client responds to the callback by sending an appropriate return code (cbrABORT, cbrCONTINUE, and so on). The callback mechanism is efficient because BDE can get the application's response without interrupting the normal client process flow.

For detailed information, see Using callbacks

To inspect the callback structures, see Data structures

{button ,AL(`databaseentities')} More database entities

Cross-database operations

BDE query and batch functions can operate on heterogeneous data sources. The following examples illustrate this feature:

- A single SQL or QBE query, can do a three-table join, for example, between InterBase, Oracle, and Paradox tables, and update a Sybase table with join result. For more information, see "Heterogeneous Joins" in the Local SQL Guide
- DbiBatchMove can be used to copy one table type to another; for example, a Paradox table to an Oracle server. All the data types are converted to the appropriate Oracle data types. The table name and all field names are converted to legal Oracle names, and options exist to convert any textual data between the character sets of the two data sources. For more information, see Adding, updating, and deleting records.
- DbiSortTable can be used, for example, to sort an Oracle table and return the result as a Paradox or a dBASE table. For more information, see Sorting tables.
- See Querying databases for a thorough exposition about cross-database operations.

{button ,AL(`databaseentities')} More database entities

Application development

These topics describe the fundamental steps of application development with the Borland Database Engine (BDE). The first topic explains how to get started and provides an introductory tutorial. The remaining topics are guides to the basic tasks.

- Introduction to BDE programming
  - Project setup
  - Basic procedure
  - Chk function
  - Template program
- Accessing and updating tables
- Locking
- Transactions
- Querying databases
- Getting and setting properties
- Retrieving schema and system information
- Creating tables
- Integrity constraints
- Modifying table structure
- Using callbacks
- Data source independence
- Error handling
- Debugging
- Filtering records
- Database driver characteristics
- Improving BDE performance

Introduction to BDE programming

These topics show you how to get started programming with the Borland Database Engine (BDE). After following the steps and examples, you will have a simple EasyWin BDE sample application that gets a record from a table and displays the first two fields. You can inspect the completed program, which demonstrates each of the basic steps in context. This program serves as a template for writing your own BDE programs.

- Project setup
  This section covers the basics of what must be done to set up a Borland Database Engine project or makefile.
- Basic procedure
  An overview of the basic steps required to create a simple application that retrieves fields from a table. From each step you can jump to a detailed description of the procedure with code examples or to the sample template program.
- Chk function
  The complete code for the Chk function, which returns more complete error information about BDE functions than would be returned by the standard error string.
- Template program
  A sample BDE program structure you can use as a convenient template for creating your own applications.

{button ,AL(`applicationdevelopment')} Application development

Project setup

Follow these steps when you begin to write a BDE application:

1. Create a Win32 console project or makefile.

   MAIN.CPP File to contain your code

   IDAPI32.LIB BDE Import Library

   MAIN.DEF Module Definition file

2. For this simple application, set the target to be a console application. This way you don't have to deal with any Windows user interface issues.

3. Install the BDE32.TOK file to support syntax highlighting for BDE functions and types. Directions on how to do this are included in the file \BDE32\DOC\BDE32TOK.TXT.

4 Make certain to compile with "Allocate enums as ints" selected (In the Borland C++ 5 IDE, Options|Project|Compiler|Code Generation). A number of structures, such as CURProps, make use of Enumerations. This error generally manifests itself with stack corruption problems, such as GP faults when calling or returning from a function.

5 Within a module to contain BDE code, include the following header files:

   WINDOWS.H

   IDAPI.H

{button ,AL(`applicationdevelopment')} Application development

Basic procedure

These are the basic steps required to get a record from a table:

1. Initialize the Borland Database Engine
1. Open a database object
1. Set the database object to point to the directory containing the table
1. Set the directory for temporary objects
1. Open a table, creating a cursor object
1. Get the properties of the table
1. Using these properties, allocate memory for a record buffer
1. Position the cursor on the desired record
1. Get the desired record from the cursor (table)
1. Get the desired fields from the record
1. Free all resources

Click on each numbered step to display a detailed explanation and a specific code sample.

At any time, you can refer to the Template program which demonstrates, in the context of a fully functional program, each of the steps in the basic procedure. You can copy and paste from this code to build your own BDE programs.

Note that throughout the short examples unfamiliar variable types are used. These are BDE variable types defined in the IDAPI.H header file, such as: BYTE, BOOL, and CHAR. In addition, the examples for the steps make use of the Chk function, which returns more complete error information about BDE functions than would be returned by the standard error string.

{button ,AL(`applicationdevelopment')} Application development

Step 1: Initialize the Borland Database Engine

Initialize BDE by using the DbiInit function:

```
  CHK(DbiInit(NULL));
```

Chk is a function that handles errors returned from BDE API calls.

{button ,JI(`>example',`templateprogram')} Template program

Step 2: Connect to a database

Now you are ready to connect to a database.

All table access must be performed within the context of a database. Local databases generally use what is referred to as the "STANDARD" database, which is used in this example.

The preferred method is to create an alias to a local directory and use that as the database. This permits easy future modification if one day it is decided to move the application from using dBASE tables to using InterBase tables.

You use the function DbiOpenDatabase to open a database:

```
hDBIDb   hDb = 0;// Handle to the Database

Chk(DbiOpenDatabase(
    NULL,                     // Database name - NULL for standard Database
    NULL,                     // Database type - NULL for standard Database
    dbiREADWRITE,             // Open mode - Read/Write or Read only
    dbiOPENSHARED,            // Share mode - Shared or Exclusive
    NULL,                     // Password - not needed for the STANDARD
  database
    NULL,                     // Number of optional parameters
    NULL,                     // Field Desc for optional parameters
    NULL,                     // Values for the optional parameters
    &hDb));                   // Handle to the database
```

{button ,JI(`>example',`templateprogram')} Template program

Step 3: Set the database object to point to the directory containing the table

Now that the database is open, you must set the table directory.

Although the working directory defaults to the directory that contains the application, most applications place data in a different directory. The working directory is the directory where the BDE expects to find tables when a path is not specified.

While it is possible to open a table in other directories by specifying the absolute path, it is preferable to open tables in the working directory, because a number of operations, such as getting a list of available tables, use the current directory. Use the function DbiSetDirectory to set the working directory (using the default location of the BDE sample tables):

```
Chk(DbiSetDirectory(
    hDb,                      // Handle to the database being modified
    "c:\\bde32\\examples\\tables"));     // The new working directory
```

Note: You must use the full, absolute path. Relative paths are not supported.

{button ,JI(`>example',`templateprogram')} Template program

Step 4: Set the directory for temporary objects

You must create a temporary directory for a client.

Not all BDE applications create temporary objects, but larger applications do sometimes create them. For example, the result set from a query of the records that cause a key violation in a restructure will be placed in a temporary table. By default, this temporary, or "private" directory, is the startup directory. This will cause a problem if the application is running on a network or a CD-ROM, because the directory cannot be shared, and it must writeable.

Use the function DbiSetPrivateDir to set the private directory for a client:

```
Chk(DbiSetPrivateDir(
   "c:\\bdetemp"));     // Select a directory on a local drive
                        // not used by other applications.
```

Note: You must use the full, absolute path. Relative paths are not supported.

{button ,JI(`>example',`templateprogram')} Template program

Step 5: Open a table, creating a cursor object

Now you can open the table.

Upon opening a table, a cursor object is created and returned to the calling application. A cursor object is an abstraction that lets you access queries and tables in the same method:

```
hDBICur  hCur = 0;      // Handle to the cursor (table)
CHAR   szTblName[DBIMAXNAMELEN];
// Table name - DBIMAXNAMELEN is defined in IDAPI.H
CHAR   szTblType[DBIMAXNAMELEN];
// Table Type

strcpy(szTblName, "customer");
// Name of the table
strcpy(szTblType, szPARADOX);
// Type of the tables - szPARADOX is defined in IDAPI.H

Chk(DbiOpenTable(
    hDb,                // Handle to the standard database
    szTblName,                // Name of the table
    szTblType,                // Type of the table - only used for local
  tables
    NULL,                     // Index Name - Optional
    NULL,                     // IndexTagName - Optional. Only used by dBASE
    0,                        // IndexId - 0 = Primary.
    dbiREADWRITE,             // Open Mode - Read/Write or Read Only
    dbiOPENSHARED,            // Shared mode - SHARED or EXCL
    xltFIELD,                 // Translate mode - Almost always xltFIELD
    FALSE,              // Unidirectional cursor movement.
    NULL,               // Optional parameters.
    &hCur));                  // Handle to the cursor
```

{button ,JI(`>example',`templateprogram')} Template program

Step 6: Get the properties of the table

To get record information from the table, you need to determine the size of the record buffer. You can obtain this information from the cursor by using the function DbiGetCursorProps. The Cursor properties include information on the table name, size, type, number of fields, and record buffer size. You can find more information on cursor properties in CURProps.

```
CURProps curProps;            // Properties of the cursor

Chk(DbiGetCursorProps(
    hCur,                     // Handle to the cursor
    &curProps));              // Properties of the cursor (table)
```

curProps.iRecBufSize contains the size of the record buffer.

{button ,JI(`>example',`templateprogram')} Template program

Step 7: Using these properties, allocate memory for a record buffer

You must use the properties you obtained in Step 8 in the following code to allocate memory for a record buffer:

```
pBYTE    pRecBuf;       // Pointer to the record buffer


pRecBuf = (pBYTE) malloc(curProps.iRecBufSize * sizeof(BYTE));
if (pRecBuf == NULL)
{
   // If pRecBuf is NULL, there was not enough memory to allocate
   // a record buffer.
   // Handling of this error is user-determined, but no information
   // from the table can be retrieved.
}
```

{button ,JI(`>example',`templateprogram')} Template program

Step 8: Position the cursor on the desired record

Use the function DbiSetToBegin to position the cursor on the "crack" before the first record in the table.

Crack semantics allow you to set the current cursor position to a point just before the first record, between records, or after the last record. One advantage of crack semantics is that it lets you use a single function to access all records in a table. For example, rather than using DbiGetRecord the first time, and DbiGetNextRecord each subsequent time, you can use DbiGetNextRecord to get all records in a table.

```
Chk(DbiSetToBegin(hCur));     // Position the specified cursor to the crack
                           // before the first record.
```

{button ,JI(`>example',`templateprogram')} Template program

Step 9: Get the desired record from the cursor (table)

To get a record from a table you would normally use the function DbiGetNextRecord. This will set the current record of the cursor to the record returned by this function (the next record in the table):

```
Chk(DbiGetNextRecord(
    hCur,               // Cursor from which to get the record.
    dbiNOLOCK,          // Lock type
    pRecBuf,            // Buffer to store the record
    NULL));       // Record properties - don't need in this case
```

{button ,JI(`>example',`templateprogram')} Template program

Step 10: Get the desired fields from the record

Now you are ready to get the field values out of the record buffer and into some local variables.

In this example, we are making assumptions about which field is at which ordinal position within the table, as well as the size of the field.

The table used by this example is CUSTOMER.DB, included with the BDE SDK.

In general, it is recommended to use DbiGetFieldDescs to get information about a field before retrieving it. Also note that a single function, DbiGetField, is used to get all fields (other than BLOBs) from a table.

```
DFLOAT custNum;
BOOL   isBlank;

Chk(DbiGetField(
    hCur,                    // Cursor which contains the record
    1,                       // Field Number of the "Customer" field.
    pRecBuf,                 // Buffer containing the record
    (pBYTE)&custNum,         // Variable for the Customer Number
    isBlank));               // Is the field blank?
```

{button ,JI(`>example',`templateprogram')} Template program

Step 11: Free all resources

After all desired operations have been performed, you need to clean up the resources allocated on behalf of the application. In addition to any memory explicitly allocated by the application, using malloc or new, all engine objects must also be cleaned up, including the cursor, database, and engine:

```
if (pRecBuf != NULL)
    free(pRecBuf);                  // Free the record buffer

if (hCur != 0)
    Chk(DbiCloseCursor(&hCur));     // Close the cursor

if (hDb != 0)
    Chk(DbiCloseDatabase(&hDb));    // Close the database

DbiExit();                      // Close the BDE.
```

{button ,JI(`>example',`templateprogram')} Template program

Chk function

The Chk function is useful for returning more complete error information about BDE functions than would be returned by the standard error string. Here is the complete code for the Chk function:

```
DBIResult Chk(DBIResult ErrorValue)
{
   char        dbi_status[DBIMAXMSGLEN * 5] = {'\0'};
   DBIMSG      dbi_string = {'\0'};
   DBIErrInfo  ErrInfo;

   if (ErrorValue != DBIERR_NONE)
   {
      DbiGetErrorInfo(TRUE, &ErrInfo);

      if (ErrInfo.iError == ErrorValue)
      {
         wsprintf(dbi_status, "  ERROR %s", ErrInfo.szErrCode);

         if (strcmp(ErrInfo.szContext1, ""))
            wsprintf(dbi_status, "%s\r\n    %s", dbi_status,
  ErrInfo.szContext1);
         if (strcmp(ErrInfo.szContext2, ""))
            wsprintf(dbi_status, "%s\r\n    %s", dbi_status,
  ErrInfo.szContext2);
         if (strcmp(ErrInfo.szContext3, ""))
            wsprintf(dbi_status, "%s\r\n    %s", dbi_status,
  ErrInfo.szContext3);
         if (strcmp(ErrInfo.szContext4, ""))
            wsprintf(dbi_status, "%s\r\n    %s", dbi_status,
  ErrInfo.szContext4);
      }
      else
      {
         DbiGetErrorString(ErrorValue, dbi_string);
         wsprintf(dbi_status, "  ERROR %s", dbi_string);
      }
      MessageBox(NULL, dbi_status, "BDE Error", MB_OK | MB_ICONEXCLAMATION);
   }
   return ErrorValue;
}
```

{button ,AL(`applicationdevelopment')} Application development

Template program

This program demonstrates each of the basic steps described in the "Basic procedure" for BDE application development. You can execute the template program and step through it to see how it works: It opens a BDE sample table and gets two records.

Use this template program as a skeleton on which to build your own BDE programs.

```
#include <idapi.h>
#include <stdio.h>
#include <windows.h>

DBIResult Chk(DBIResult);     // Function Prototype


void main ()
{
   hDBIDb   hDb = 0;          // Handle to the Database
   hDBICur     hCur = 0;      // Handle to the cursor (table)
   CHAR        szTblName[DBIMAXNAMELEN];
   CHAR        szTblType[DBIMAXNAMELEN];
   CURProps    curProps;      // Properties of the cursor
   pBYTE       pRecBuf;             // Pointer to the record buffer
   DFLOAT      custNum;
   BOOL        isBlank;


   printf("\nInitialize engine");
   Chk(DbiInit(NULL));                                                //
  Step 2

printf("\nOpen database");
   Chk(DbiOpenDatabase(                                               //
  Step 3
       NULL,                  // Database name - NULL for standard database
       NULL,                  // Database type - NULL for standard database
       dbiREADWRITE,          // Open mode - Read/Write or Read only
       dbiOPENSHARED,         // Share mode - Shared or Exclusive
       NULL,                  // Password - not needed for the STANDARD
  database
       NULL,                  // Number of optional parameters
       NULL,                  // Field Desc for optional parameters
       NULL,                  // Values for the optional parameters
       &hDb));                // Handle to the database

   printf("\nSet table directory");
   Chk(DbiSetDirectory(                                               //
  Step 5
       hDb,                         // Handle to the database which is being
  modified
       "e:\\bde32\\examples\\tables"));
                              // The new working directory

   printf("\nSet private directory");
   Chk(DbiSetPrivateDir(                                              //
  Step 6
       "c:\\temp"));          // Select a directory on a local drive not
```

```
used
                            // by other applications.

 strcpy(szTblName, "customer");
 strcpy(szTblType, szPARADOX);
 printf("\nOpen table");
 Chk(DbiOpenTable(                                                    //
Step 7
    hDb,              // Handle to the standard database
    szTblName,              // Name of the table
    szTblType,              // Type of the table - only used for local
tables
    NULL,                   // Index Name - Optional
    NULL,                   // IndexTagName - Optional. Only used by dBASE
    0,                      // IndexId - 0 = Primary.
    dbiREADWRITE,           // Open Mode - Read/Write or Read Only
    dbiOPENSHARED,          // Shared mode - SHARED or EXCL
    xltFIELD,               // Translate mode - Almost always xltFIELD
    FALSE,                  // Unidirectional cursor movement.
    NULL,             // Optional Parameters.
    &hCur));                // Handle to the cursor

 printf("\nGet cursor properties");
 Chk(DbiGetCursorProps(                                               //
Step 8
    hCur,                   // Handle to the cursor
    &curProps));            // Properties of the cursor (table)

 printf("\nAllocate a record buffer");
 pRecBuf = (pBYTE) malloc(curProps.iRecBufSize * sizeof(BYTE));       //
Step 9
 if (pRecBuf == NULL)
 {
    // If pRecBuf is NULL, there was not enough memory to allocate a
    // record buffer. Handling this error will be user determined, but
    // no information from the table can be retrieved.
 }
 else
 {
    printf("\nSet cursor to the crack before the first record");
    Chk(DbiSetToBegin(hCur));                                         //
Step 10
                            // Position the specified cursor to the crack
                            // before the first record

    printf("\nGet the next record");
    Chk(DbiGetNextRecord(                                             //
Step 11
       hCur,                // Cursor from which to get the record.
       dbiNOLOCK,           // Lock Type
       pRecBuf,             // Buffer to store the record
       NULL));              // Record properties - don't need in this case

    printf("\nGet a field out of the record buffer");
    Chk(DbiGetField(
       hCur,                // Cursor which contains the record
       1,                   // Field Number of the "Customer" field.
```

```
         pRecBuf,            // Buffer containing the record
         (pBYTE)&custNum,    // Variable for the Customer Number
         &isBlank));         // Is the field blank?

      printf("\nThe retrieved field value is %f", custNum);
   }

   printf("\nClean-up");

   if (pRecBuf != NULL)
      free(pRecBuf);         // Free the record buffer

   if (hCur != 0)
      Chk(DbiCloseCursor(&hCur));
                             // Close the cursor

   if (hDb != 0)
      Chk(DbiCloseDatabase(&hDb));
                             // Close the database

   DbiExit();                // Close the BDE.
}

DBIResult Chk(DBIResult ErrorValue)
{
   char        dbi_status[DBIMAXMSGLEN * 5] = {'\0'};
   DBIMSG      dbi_string = {'\0'};
   DBIErrInfo  ErrInfo;

   if (ErrorValue != DBIERR_NONE)
   {
      DbiGetErrorInfo(TRUE, &ErrInfo);

      if (ErrInfo.iError == ErrorValue)
      {
         wsprintf(dbi_status, "  ERROR %s", ErrInfo.szErrCode);

         if (strcmp(ErrInfo.szContext1, ""))
            wsprintf(dbi_status, "%s\r\n    %s", dbi_status,
  ErrInfo.szContext1);
         if (strcmp(ErrInfo.szContext2, ""))
            wsprintf(dbi_status, "%s\r\n    %s", dbi_status,
  ErrInfo.szContext2);
         if (strcmp(ErrInfo.szContext3, ""))
            wsprintf(dbi_status, "%s\r\n    %s", dbi_status,
  ErrInfo.szContext3);
         if (strcmp(ErrInfo.szContext4, ""))
            wsprintf(dbi_status, "%s\r\n    %s", dbi_status,
  ErrInfo.szContext4);
      }
      else
      {
         DbiGetErrorString(ErrorValue, dbi_string);
         wsprintf(dbi_status, "  ERROR %s", dbi_string);
      }
      MessageBox(NULL, dbi_status, "BDE Error", MB_OK | MB_ICONEXCLAMATION);
   }
```

```
   return ErrorValue;
}
```

{button ,AL(`applicationdevelopment')} Application development

Accessing and updating tables

This table is an overview of the process of accessing and updating tables using BDE:

Phase	Task	BDE function
Preparation	Initialize the database engine	Call DbiInit
Preparation	Open a database	Call DbiOpenDatabase
Preparation	Open a table and get a cursor	Call DbiOpenTable
Preparation	Get the cursor properties	Call DbiGetCursorProps
Preparation	Allocate the record buffer	Responsibility of the application
Preparation	Retrieve field descriptor information into application-supplied memory	Call DbiGetFieldDescs
Preparation	Begin cached updates mode	Call DbiBeginDelayedUpdates
Retrieval	Position the cursor and fetch a record into the record buffer	Call DbiGetNextRecord
Retrieval	Retrieve a field from the record buffer	Call DbiGetField
Update	Update the field and write it to the record buffer	Call DbiPutField
Update	Update the table with the new record	Call DbiModifyRecord
Update	Apply cached updates to the table	Call DbiApplyDelayedUpdates
Exit	End the cached updates mode	Call DbiEndDelayedUpdates
Exit	Close the cursor	Call DbiCloseCursor
Exit	Close the database	Call DbiCloseDatabase
Exit	Exit the database engine	Call DbiExit

{button ,AL(`applicationdevelopment')} Application development topics

{button ,AL(`accessingtables')} Accessing and updating tables

Also See:

- Transactions
- Transactions on Paradox, dBASE, FoxPro, and Access

Preparing to access a table

The steps for preparing to access a table are described in the following topics:

- Initializing BDE
- Opening a database
- Opening a table
- Preparing the record buffer and retrieving field descriptors

Initializing BDE

The first call that the application makes to BDE is always DbiInit, to initialize the database engine and start a new session. DbiInit can optionally be supplied with a pointer to the environment information structure DBIEnv. The NULL pointer is normally passed, which forces BDE to search for the Registry entries and the BDE configuration file (IDAPI.CFG), and to use the default settings. When a NULL pointer to the DBIEnv structure is passed, BDE searches in the following order for the configuration file:

1 BDE checks the Windows Registry for a configuration file defined by an entry of [BDE] with a subentry of CONFIGFILE01.

2 If step 1 is not successful, BDE checks for the configuration file named IDAPI.CFG in the startup directory.

3 If step 2 is not successful, BDE initializes with a default set of configuration settings, predefined for each driver. If initialization takes place after the failure of steps 1 and 2, no SQL driver access is possible.

If the pointer is not NULL, and the configuration file is specified in the DBIEnv structure, BDE uses that configuration file.

Here is a sample DbiInit call:

```
// Initialize IDAPI
rslt = DbiInit(NULL);
```

{button ,AL(`preparingtoaccess')} Preparing to access a table

{button ,AL(`accessingtables')} Accessing and updating tables

See also: Initialization

Opening a database

A database must be opened with a call to DbiOpenDatabase before a table in the database can be opened. A successful call to DbiOpenDatabase returns the database handle, which is then passed in subsequent calls to many other BDE functions.

For SQL databases, a password and user name must be supplied with DbiOpenDatabase to connect to the server.

Specifying a standard database

The following code sample opens a standard database (used to access Paradox, dBASE, FoxPro, Access, and Text tables) by using a NULL database name and database type:

```
rslt = DbiOpenDatabase(NULL, NULL, dbiREADWRITE,
          dbiOPENSHARED, NULL, 0, NULL, NULL, &hDb)
```

To change the current directory for a standard database, call DbiSetDirectory

```
rslt = DbiSetDirectory(hDb, "C:\\DATE");
```

Specifying a SQL database

There are several different methods of specifying a SQL database in the DbiOpenDatabase call:

- The database name can specify a SQL alias, which defines a SQL database in the configuration file. If a SQL alias is specified, the database type is NULL and optional fields are not required.
- The database name can be NULL if the database type specifies one of the SQL driver names (for example, InterBase or Oracle). If optional parameters are not specified, driver-specific defaults are used.

For example, this code sample opens a named database on a SQL server:

```
rslt = DbiOpenDatabase("myalias", NULL, dbiREADWRITE,
                       dbiOPENSHARED, "mypassword", 0, NULL, NULL,
                       &hDb)
```

Specifying an alias

When calling DbiOpenDatabase you can supply an alias referencing a database name in the configuration file.

Specifying access rights

The eOpenMode and eShareMode parameters of the DbiOpenDatabase call, in combination with eOpenMode and eShareMode parameters of the DbiOpenTable call, determine the access rights of users to tables within a database.

Note: For SQL data sources, the OPEN MODE parameter for each alias in the BDE configuration file takes precedence over the open mode parameters passed with DbiOpenDatabase.

If the database open mode is read-only, tables within that database cannot be opened by DbiOpenTable in read-write mode. If the database open mode is read-write, tables within that database can be opened by DbiOpenTable either in read-only or read-write mode.

If the database share mode is exclusive, tables within that database cannot be opened by DbiOpenTable in share mode. If the database was opened in share mode, tables within that database can be opened by DbiOpenTable in either exclusive or share mode.

Specifying optional parameters

Optional database-specific parameters can be passed to the DbiOpenDatabase function. To retrieve a list and description of these optional parameters for a database, the application can call DbiOpenCfgInfoList, supplying the path of the database name in the configuration file. This function returns the handle to a virtual table listing optional parameters for this

database system and default values for these parameters.

OptFields, *pOptFldDesc* and *pOptParams* are the optional parameters, but may actually be required, depending on which driver is being used, and whether enough information has been supplied with other parameters to specify the database. For more on these parameters, see DbiCreateTable

---

{button ,AL(`preparingtoaccess')} Preparing to access a table

{button ,AL(`accessingtables')} Accessing and updating tables

Opening a table

You can open a table by calling DbiOpenTable, and passing appropriate parameters such as table name, driver type, index, type of access, and share mode. After the table is successfully opened, BDE returns a cursor handle to the table.

Specifying the table name and driver type

If the application supplies the fully qualified table name of a Paradox, or dBASE table, it need not specify the driver type parameter, because the driver type can be determined from the table name extension. If the table name does not include a path, the path name defaults to that of the current directory of the database associated with the database handle.

Driver type must be specified if the table name has no extension, or to overwrite the default driver associated with the file extension, or to terminate the table name with a period(.). If the table name does not supply the default extension, and driver type parameter is NULL, DbiOpenTable attempts to open the table with the default file extension designated for each file-based driver listed in the configuration file, in the order that the drivers are listed.

The driver types and their default extensions for Paradox, dBASE, and Text drivers are listed below:

Driver type	Default extension
PARADOX	.DB
dBASE	.DBF
ASCIIDRV	.TXT

For SQL databases, the table name can be a fully qualified name that includes the owner name, in the form

```
<owner>.<tablename>
```

If not specified, <owner> is inferred from the database handle. Driver type is ignored if the database is a SQL database, since driver binding is done when the database is opened.

For Access databases, a driver type and table identifier are required.

Specifying an index

To open a table with an active index, you can use the following parameters, depending on the type of table being opened: *pszIndexName*, *pszIndexTagName*, or *iIndexId*. The active index determines the order of records for this cursor.

Paradox: If all index parameters are NULL, the table is opened in primary key order, if a primary key exists. If a secondary key is specified, the table is opened on that key. Either *pszIndexName* or *iIndexID* can be used to specify a composite or non-composite secondary index.

Access: If all index parameters are NULL, the table is opened in natural order. Either *pszIndexName* or *iIndexID* can be used to specify a composite or non-composite secondary index.

dBASE and FoxPro: If no index is specified, the table is opened in physical order.

- Use the *pszIndexName* parameter in the form <tablename>.MDX or <tablename>.CDX if the index is within a production index.
- Use the *pszIndexTagName* parameter to specify the tag name of the index in an MDX or CDX file. This parameter is ignored if the index given by *pszIndexName* is an NDX index.

SQL: Use the *pszIndexName* parameter to specify the index name. The index name can be qualified or unqualified. An unqualified index name succeeds only if the owner of the index is the current user. (For servers supporting naming conventions with owner qualification, it is not necessary to qualify the index name with the owner.)

Specifying table open mode

A table can be opened in EXCLUSIVE or SHARED mode. When a table is opened in exclusive mode, no other user can access the table. When a table is opened in share mode, other users can access the table at the same time.

Specifying the data translation mode

The *xltFIELD* translation mode is recommended. This mode ensures that BDE automatically translates data from the database's native physical data format to the common BDE logical data format when a field is read from the record buffer. BDE translates the data back into native format when the field is written to the record buffer.

When the translation mode is xltNONE, no data translation takes place when a field is read from the record buffer, or when a field is written to the record buffer.

Note: Data translation occurs only during calls to DbiGetField and DbiPutField; not when the record is read.

{button ,AL(`preparingtoaccess')} Preparing to access a table

{button ,AL(`accessingtables')} Accessing and updating tables

Preparing the record buffer and retrieving field descriptors

A successful call to DbiOpenTable returns a cursor handle to the application. Before it can use the cursor handle to access data in the table, the application must prepare the record buffer. Preparing the record buffer includes allocating memory for it and, in some cases, initializing it.

The application can also set up an array in which to retrieve the field descriptors for each field contained in the table. To determine the required sizes of the record buffer and the array of field descriptors, the application calls DbiGetCursorProps. This call is usually made immediately after the DbiOpenTable call, and returns the required information in the CURProps structure.

Example

The following code sample gets the cursor properties, allocates the record buffer, sets up an array for the field descriptors, and gets the field descriptors:

```
    DBIResult   rslt;
    pCHAR       pRecBuf;
    CURProps    curProps;
    pFLDDesc    pFldArray;
    ...
  // Get the table properties
     rslt = DbiGetCursorProps(hCursor, &curProps);
     if (rslt == DBIERR_NONE)
     {
         // Allocate the record buffer
         pRecBuf = malloc(curProps.iRecBufSize);
              // Check result of malloc
    ...
  // Get an array of field descriptors
         pFldArray = (pFLDDesc) malloc(sizeof(FLDDesc) *
                                       curProps.iFields);
           // Check result of malloc
    ...
  rslt = DbiGetFieldDescs(hCursor, pFldArray);
    ...
  free(pFldArray);
  free(pRecBuf);
    }
```

Getting the cursor properties

When the application calls DbiGetCursorProps, the cursor properties CURProps structure is returned with information describing the most commonly used cursor properties. CURProps contains the following fields:

Type	Name	Description
DBITBLNAME	*szName*	Table name (no extension, if it can be derived)
UINT16	*iFNameSize*	Full file name size
DBINAME	*szTableType*	Table type
UINT16	*iFields*	Number of fields in table
UINT16	*iRecSize*	Record size (logical record)
UINT16	*iRecBufSize*	Record size (physical record)
UINT16	*iKeySize*	Key size
UINT16	*iIndexes*	Number of currently available indexes

UINT16	*iValChecks*	Number of validity checks
UINT16	*iRefIntChecks*	Number of referential integrity constraints
UINT16	*iBookMarkSize*	Bookmark size
BOOL	*bBookMarkStable*	TRUE, if the cursor supports stable bookmarks
DBIOpenMode	*eOpenMode*	dbiREADWRITE, dbiREADONLY
DBIShareMode	*eShareMode*	dbiOPENSHARED, dbiOPENEXCL
BOOL	*bIndexed*	TRUE, if the index is active
INT16	*iSeqNums*	1: Has sequence numbers (Paradox); 0: Has record numbers (dBASE, FoxPro); < 0 (-1, -2. . .): None (SQL and Access)
BOOL	*bSoftDeletes*	TRUE, if the cursor supports soft deletes (dBASE and FoxPro only)
BOOL	*bDeletedOn*	TRUE, if deleted records are seen
UINT16	*iRefRange*	If > 0, has active refresh
XLTMode	*exltMode*	Translate mode: xltNONE (physical types), xltFIELD (logical types)
UINT16	*iRestrVersion*	Restructure version number
BOOL	*bUniDirectional*	TRUE, if the cursor is unidirectional (SQL only)
PRVType	*eprvRights*	Table-level rights
UINT16	*iFmlRights*	Family rights (Paradox only)
UINT16	*iPasswords*	Number of auxiliary passwords (Paradox only)
UINT16	*iCodePage*	Code page; if unknown, set to 0
BOOL	*bProtected*	TRUE, if the table is protected by password
UINT16	*iTblLevel*	Driver-dependent table level
DBINAME	*szLangDriver*	Symbolic name of language driver
BOOL	*bFieldMap*	TRUE, if a field map is active
UINT16	*iBlockSize*	Data block size in bytes, if any
BOOL	*bStrictRefInt*	TRUE, if strict referential integrity is in place
UINT16	*iFilters*	Number of filters
BOOL	*bTempTable*	TRUE, if the table is temporary

Memory allocation elements

The following elements are significant when allocating memory:

iFields

Specifies the number of fields in the table. Use this number to allocate an array to receive the field descriptors for the table. The size of the array is:

```
iFields * sizeof(FLDDesc)
```

iRecSize

Specifies the record size, depending on the translation mode for the cursor. If the translation mode is *xltFIELD*, *iRecSize* specifies the logical record size. In other words, it is the size of the record if all fields were represented as BDE logical types. If the translation mode is xltNONE, *iRecSize* specifies the physical record size, which is the same as *iRecBufSize*.

iRecBufSize
Specifies the physical record size. This is the size of the record buffer that you must allocate in order to retrieve the records by using DbiGetNextRecord, DbiGetPriorRecord, and other functions. For example,

```
pRecBuf = (pBYTE)malloc(curProps.iRecBufSize);
```

Initializing the record buffer

Initialize the record buffer with a call to DbiInitRecord if a new record is to be inserted. This function initializes each field in the record buffer, including BLOB fields, to blanks based on the data type defined. For Paradox tables, default values are used to initialize the fields if default values are specified in the table.

Getting the field descriptors

After memory has been allocated for the array of field descriptors, the application can retrieve the field descriptors with a call to DbiGetFieldDescs. The field descriptors provide the application with information that it needs to address and manipulate each field within the record buffer. DbiGetFieldDescs returns an array of FLDDesc structures, with information describing each field in the table:

Type	Name	Description
UINT16	*iFldNum*	Field number (1 to n)
DBINAME	*szName*	Field name
UINT16	*iFldType*	Field type
UINT16	*iSubType*	Field subtype (if applicable)
UINT16	*iUnits1*	Number of characters or units
UINT16	*iUnits2*	Decimal places
UINT16	*iOffset*	Offset in the record (computed)
UINT16	*iLen*	Length in bytes (computed)
UINT16	*iNullOffset*	For NULL bits (computed)
FLDVchk	*efldvVchk*	Field has validity checks (computed)
FLDRights	*efldrRights*	Field rights (computed)

iFldNum — Specifies a driver-specific field ID. For most drivers, this value is from 1 to curProps.iFields, except for Paradox tables, which can use an invariant field ID For more information about invariant field IDs, refer to DbiDoRestructure

Note: For consistency across drivers, use the ordinal position of the field in the descriptor array. Both DbiGetField and DbiPutField use an ordinal number from 1 to n.

szName
Specifies the name of the field.

iFldType
Specifies the type of the field. Depending on the translate mode property of this cursor the field type returned could be physical or logical. If the translate mode is xltFIELD, the field type returned is a BDE logical type; if the mode is xltNONE, the field type returned is the driver's corresponding physical type. For more information about physical and logical data types, see Using the function reference and Data structures

iSubType
Specifies the subtype of the field. This could be a BDE logical subtype or a driver physical subtype, depending on the translate mode.

iUnits1
Specifies the number of characters, digits, and so on. For logical field types, this number

is consistent across drivers. For physical field types, the interpretation of this field can be dependent on the driver and also on the specific field type. For most drivers, if the field is of the numeric type, *iUnits1* is the precision and *iUnits2* is the scale.

iUnits2
Specifies the number of decimal places, and so on. For logical field types, this number is consistent across drivers. For physical field types, the interpretation of this field can depend on the driver and also on the specific field type. For most drivers, if the field is of the numeric type, *iUnits1* is the precision and *iUnits2* is the scale.

The following three fields together specify the layout of the record buffer:

iOffset
Specifies the offset of this field in the record buffer. The offset depends on the translation mode. If the mode is xltFIELD, it is the offset of the field within a logical record.

iLen
Specifies the length of this field. The length depends on the translation mode; that is, it could be the length of the logical or physical representation of the field. The application developer uses this value to allocate a buffer in which to retrieve the field value.

iNullOffset
Specifies the offset of the NULL indicator for this field in the record buffer. If zero, there is no NULL indicator. Otherwise, iNullOffset is the offset to an INT16 value, which is -1 if the field is NULL (SQL only).

efldvVchk
Specifies whether or not validity checks are associated with this field (Paradox and SQL drivers only).

efldrRights
Specifies the field level rights for this field.

{button ,AL(`preparingtoaccess')} Preparing to access a table

{button ,AL(`accessingtables')} Accessing and updating tables

Positioning the cursor and fetching records

After the record buffer has been prepared, the application can use the record buffer to fetch records from the table.

To fetch records, the application must position the cursor on the record that it wants to fetch. Some BDE functions serve only to position the cursor. Calls to these functions can be followed by a call to a function that fetches the record into the record buffer. Other BDE functions can simultaneously position the cursor and fetch a record into the record buffer.

Positioning the cursor on a crack

Some BDE functions position the cursor before a record, at the beginning of the file or result set, or at the end of the file. When the cursor is positioned at one of these locations, rather than on a record, the cursor is said to be positioned on a crack. The following calls position the cursor on a crack:

- DbiSetToBegin positions the cursor to the beginning of the file (just before the first record). When the cursor is opened, it is at this position.
- DbiSetToEnd positions the cursor to the end of the file (just after the last record).
- DbiSetToKey positions the cursor just prior to the record of the specified key value.

Positioning the cursor on a crack can simplify programming. For example, calling DbiSetToBegin positions the cursor on the crack before the first record in the table. Then, you can set up a loop to process all the records in the table with DbiGetNextRecord. (If the cursor had been positioned on the first record in the table to start with, instead of before the first record, the DbiGetNextRecord loop would have skipped the first record.)

Positioning the cursor on a record and fetching a record

Some BDE functions position the cursor directly on a record. If a record buffer is supplied, these functions can also be used to fetch the record for processing by the application. Most of these calls can optionally lock the record. The record remains locked until it is released explicitly, or the session is closed. For more information about locks, see Locking

DbiGetRecord
This function fetches the current record, and returns an error if the cursor is positioned on a crack.

DbiGetNextRecord
This function positions the cursor on the next record after the current position of the cursor, and also fetches that record.

DbiGetPriorRecord
This function positions the cursor on the record before the current position of the cursor, and also fetches that record.

DbiGetRelativeRecord
This function positions the cursor on the record whose position is specified as an offset (either a positive or a negative number) from the current position of the cursor, and also fetches that record.

DbiGetRecordForKey
This function positions the cursor on the record whose key matches the specified key, and also fetches that record.

Example
The following example shows how to position the cursor to the beginning of file and step through the table:

```
// Position the cursor at the BOF crack
   DbiSetToBegin(hCursor);
   // Step through the table. Read the record each time.
   while (DbiGetNextRecord(hCursor, dbiNOLOCK, pRecBuf, NULL)
```

```
            == DBIERR_NONE)
    {
...
}
```

Repositioning the cursor with bookmarks

Bookmarks provide a convenient way to save the position of the cursor, so that it can be repositioned to that same place later. The bookmark is written to a client-supplied buffer which is allocated by the client.

Note: The size of the bookmark buffer may change after a call to DbiSwitchToIndex.

DbiGetBookmark
This function saves the current position in the supplied bookmark.

DbiSetToBookmark
This function repositions the cursor to a previously saved bookmark position.

Fetching multiple records

The application can fetch multiple records with one call by setting up a buffer large enough to hold the records and calling DbiReadBlock. The specified number of records are fetched beginning with the next record after the current cursor position. This function is equivalent to setting up a loop that makes multiple calls to DbiGetNextRecord.

Retrieving limited record sets

Several BDE functions enable you to force the cursor to return only a limited set of records or fields to the application; that is, the application sees only those records in the table that meet a predefined set of conditions.

Note: Queries provide another way of returning a limited record set.

Using ranges

Use DbiSetRange to force the cursor to return to the application only those records whose keys fall within the defined range. This function can be called only if the cursor has a current active index. (See DbiOpenTable or DbiSwitchToIndex.) Both inclusive and exclusive ranges can be specified. Subsequent BDE calls treat the set of records within the range as the complete table. For example, DbiSetToBegin positions the cursor on the crack before the first record in the range, rather than on the first record in the table.

This function is commonly used to find a set of records between two key values by setting both the upper range limit and the lower range limit. Open-ended ranges can be specified, from the beginning of the file to a specified key, or from a specified key to the end of the file.

For an example, refer to the RANGE code sample in the SNIPIT Code Viewer (\BDE32\EXAMPLES\SNIPIT).

Creating field maps

Use DbiSetFieldMap to force the cursor to return fields in a different order from their order in the table, or to drop fields from view. To set up a field map, the application developer builds an array of field descriptors, including only those fields that are to be made visible by the cursor, and in the order that they are to be returned. Only the fields named in the array are made visible.

Note: Creating field maps can change the size of the record buffer.

For an example, refer to the FLDMAP.C code sample in the SNIPIT Code Viewer (\BDE\EXAMPLES\SNIPIT).

Using filters

An active filter forces the cursor to return a limited record set consisting of only those

records that meet the filter condition. Records that do not meet the filter condition are skipped, and even though they remain in the table, the records are not visible through the cursor. Deactivating the filter brings those records back into view.

A filter condition is defined as an expression returning TRUE or FALSE. When the filter is activated, the filter expression is applied to each record in the table. Only those records that return TRUE are visible to the application. Multiple filters can be defined for one table.

To define a filter, the application calls DbiAddFilter, passing it an existing cursor handle and a pointer to a *CANExpr* structure that contains the expression. The structure is passed in a flat tree format. (For a detailed explanation and an example of how to use filters, see Filtering records.)

The CANExpr structure can include comparison operators, AND, OR, and NOT, and tests for blank fields. Different drivers support different types of expressions, but all drivers support the basic combination of <field> <compare operator> <constant>; for example, "field1 = "CA" and field2 < 30" is supported by all drivers.

When DbiAddFilter completes, it returns a filter handle to the application.

After the filter condition has been defined, it must be activated with DbiActivateFilter in order to take effect. Multiple filters can be activated. Filters can be switched on and off when needed (using DbiActivateFilter and DbiDeactivateFilter). Filters are automatically dropped when the cursor is closed, and can be explicitly dropped with DbiDropFilter. If more than one filter is active, records that fail to meet any active filter condition are filtered out.

Advantages of using filters are that the BDE filtering mechanism is extremely fast, and filters are implemented efficiently by the drivers.

Note: While queries provide a more general way of restricting the result set than filters, filters provide more dynamic control than queries.

{button ,AL(`accessingtables')} Accessing and updating tables

Field-level access

An application usually accesses data in a record at the field level. The BDE functions DbiGetField and DbiPutField let the application retrieve and update the data within each field in a record buffer. These functions allow field access without the need to know the structure of a record buffer.

Field-level access is done through a record buffer:

Reading a record
--> Table [DbiGetRecord] --> Record buffer [DbiGetField] --> Field

Updating a record
--> Field [DbiPutField] --> Record buffer [DbiModifyRecord] --> Table

Retrieving field values

To retrieve a field within the record buffer, the application calls the BDE function DbiGetField, supplying the ordinal number of the field and a buffer to hold the data contents of the field. (The ordinal number is the position of the FLDDesc in the array returned by DbiGetFieldDescs, 1 to n.) Optionally, a Boolean can be returned indicating if the field is blank.

Updating field values

To update a field in the record buffer, the application calls the BDE function DbiPutField, supplying the ordinal number of the field, and a buffer containing the field contents to be written to the record. (The ordinal number is the position of the FLDDesc in the array returned by DbiGetFieldDescs, 1 to n.)

DbiPutField can also be used to set a field to blank, by passing a NULL pointer as the field buffer parameter.

Logical types versus physical types

As a general rule, the application should always use field translation mode xltFIELD. This parameter is set when the table is opened. If the table has already been opened and the translation mode is not set to xltFIELD, it can be changed with the DbiSetProp call.

When field translation mode is in effect, BDE automatically translates a field's data contents. When the field is retrieved, BDE translates the data in the record buffer from the native data type into a generic logical data type. When the field is written back to the record buffer, BDE translates the data back into the native physical data type.

When field translation mode is not in effect, BDE performs no translation of data to logical types. The application must be prepared to accept data from BDE using the data types native to the database system managing the table.

BDE type	C equivalent	Description
fldZSTRING	char[ ]	Zero terminated array of chars
fldUINT16	unsigned int	16-bit unsigned integer
fldINT16	int	16-bit integer
fldUINT32	unsigned long	32-bit unsigned long integer
fldINT32	long	32-bit long integer
fldFLOAT	double	64-bit floating point
fldFLOATIEEE	long double	80-bit floating point
fldBOOL	int	16-bit quantity, TRUE==1; FALSE==0
fldBYTES	unsigned char[ ]	Fixed size (independent of row) array of bytes
fldVARBYTES	unsigned char[ ]	Length-prefixed array of bytes

{button ,AL(`accessingtables')} Accessing and updating tables

Working with BLOBs

Because BLOB fields are variable-sized and can be very large, BDE treats them differently from other fields; they are treated as byte streams. The application developer follows a similar procedure for accessing and updating records containing BLOB fields as with other records.

The following set of BDE functions is designed to work with BLOB fields:

- DbiOpenBlob
- DbiGetBlob
- DbiGetBlobHeading
- DbiGetBlobSize
- DbiFreeBlob
- DbiPutBlob
- DbiTruncateBlob

Opening the BLOB

To write to or read from a BLOB, you must open the BLOB first. To open the BLOB, the record buffer must contain a copy of the record to be modified, or an initialized record, if the record is being inserted. The application calls DbiOpenBlob, passing the cursor handle, the pointer to the record buffer, the field number of the BLOB, and the access rights. (If the BLOB is opened in read-write mode, the table must also be opened in read-write mode.) DbiOpenBlob stores the BLOB handle in the record buffer. DbiOpenBlob must be called prior to calling any other BLOB functions.

Standard: It is advisable to lock the record before opening the BLOB in read-write mode. This ensures that another application does not change the record or lock the record, preventing the record from being updated.

SQL: For SQL servers that do not support BLOB handles for random reads and writes, full BLOB support requires uniquely identifiable rows. Most SQL servers limit a single sequential BLOB read to less than the maximum size of a BLOB. In cases with no row uniqueness and without BLOB handles, an entire BLOB might not be available.

Retrieving BLOB data

DbiGetBlob retrieves BLOB data from the specified BLOB. Any portion of the data can be retrieved, starting from the position specified in *iOffSet*, and extending to the number of bytes specified in *iLen*. Typically, the application does not know the length of the BLOB, and it makes a series of calls to DbiGetBlob to retrieve the entire BLOB. DbiGetBlob returns the number of bytes read when it completes. The application can tell when it has reached the end of the BLOB when the number of bytes specified in iLen is greater than the number of bytes read.

Alternatively, the application can determine beforehand the size of the BLOB by calling DbiGetBlobSize, and then specifying the actual length of the BLOB in the call to DbiGetBlob. That way, the entire BLOB can be retrieved with one DbiGetBlob call, instead of a series of calls.

Updating a BLOB

DbiPutBlob is the equivalent of DbiPutField for a BLOB. DbiPutBlob is used only to write data into a BLOB. The BLOB must be opened in read-write mode. The application passes a pointer to the block of data to be written. The application specifies the length of data to be written, as well as the offset within the BLOB to begin writing the data. The application can make a series of calls to DbiPutBlob to write the entire BLOB.

Updating or adding a record with a blob

To update or add a record, the application follows these steps:

1 Calls DbiAppendRecord or DbiInsertRecord to add a new record with a BLOB to the table or the application calls DbiModifyRecord to modify an existing record containing a BLOB. The pointer to the record buffer containing the new record is passed with the function.

2 Calls DbiFreeBlob to close the BLOB handle and all resources allocated to the BLOB by DbiOpenBlob. (DbiModifyRecord, DbiInsertRecord or DbiAppendRecord do not automatically release BLOB resources after record modification.)

Note: It is important to free the BLOB after adding or modifying the record. If DbiFreeBlob is called prior to DbiModifyRecord, DbiInsertRecord, or DbiAppendRecord, the changes are lost.

Note: Do not use DbiWriteBlock on tables that contain BLOBs.

This example illustrates BLOB processing:

```
DBIResult   rslt;
     pCHAR       blobBuf;
     UINT32      blobSize, bytesRead;
  // Read the current record
  DbiGetRecord(hCursor, dbiNOLOCK, pRecBuf, NULL);
  // Open the BLOB
  rslt = DbiOpenBlob(hCursor, pRecBuf, 3, dbiREADWRITE);
  if (rslt == DBIERR_NONE)
  {
      // Get the size of the BLOB then read it. Note that this
      // example assumes that the BLOB is less than 64k.
      DbiGetBlobSize(hCursor, pRecBuf, 3, &blobSize);
      blobBuf = malloc(blobSize);
      DbiGetBlob(hCursor, pRecBuf, 3, 0, blobSize,
                 (pBYTE) blobBuf, &bytesRead);
  ...
  .    // Free the blob
      DbiFreeBlob(hCursor, pRecBuf, 3);
      // Clean up
      free(blobBuf);
  }
```

{button ,AL(`accessingtables')} Accessing and updating tables

Adding, updating, and deleting records

In order to add, modify or delete a record, the cursor must have write access to the table. The table or record must not be locked by another user. If the application intends to update a record, it can lock the record through the BDE function that fetches the record. The record remains locked until the application explicitly releases it, or the session is closed. For more information about locks, see Locking

Alternatively, you can use the cached updates cursor layer to allow users to retrieve and modify temporarily cached data without immediately writing to the actual underlying database. This minimizes the amount of resource locking.

Adding a record

To add a new record to a table, the application follows these steps:

1 Initializes the client-allocated record buffer with a call to DbiInitRecord.

2 Constructs the record one field at a time, using DbiPutField For information about BLOB fields, see Working With BLOBs

3 Calls DbiAppendRecord or DbiInsertRecord to write the record buffer contents to the table. The application specifies whether or not to keep a record lock on the inserted record (DbiInsertRecord).

Updating a record

To modify an existing record in the table, the application follows these steps:

1 Fetches the record to be modified into the client-allocated record buffer (obtaining a lock, if necessary).

2 Writes the updated fields to the record buffer with DbiPutField For information about BLOB fields, see Working with BLOBs

3 Calls DbiModifyRecord to write the record buffer to the table. The application specifies whether or not to release the record lock on the updated record when DbiModifyRecord completes.

Deleting a record

To delete a record, the application follows these steps:

1 Positions the cursor on the record to be deleted.

2 Calls DbiDeleteRecord. If a record buffer is supplied, the deleted record is copied there.

3 The cursor is left positioned on the crack where the deleted record was.

dBASE and FoxPro

For dBASE and FoxPro tables, a deleted record is not removed from the table until a call to DbiPackTable is made.

Paradox

The record cannot be recalled once it is deleted. The record is not deleted if the deletion would cause violation of referential integrity. For example, if the cursor is validly positioned on a record within the master table, and that record has linked values in a detail table, then the call to DbiDeleteRecord fails, and the position of the cursor remains unchanged.

Deleting a record does not reduce table size. The only way to gain disk space for records that have been deleted is to restructure the table with a call to DbiDoRestructure. Deleted space may be reused by later inserts.

Multiple Record Updating, Adding, And Deleting

BDE provides two functions that enable your application to update, add, or delete multiple records from a table: DbiBatchMove and DbiWriteBlock.

DbiBatchMove

DbiBatchMove can be used in different modes to append, update, append and update, or subtract records from a source table to a destination table. Source and destination tables can be of different driver types. This function supports filters and field maps. It can also copy a table of one driver type to a new table of a different driver type.

This function can be used with the Text driver to import and export data to or from any supported driver type.

This function can optionally create a key violations tables, a changed table, and a problems table to store records that fail to meet the specified criteria for record transfer. A callback can be registered that alerts the application to data transfer between source and destination fields that could result in data loss.

For an example, refer to the BATMOVE.C code sample in the SNIPIT Code Viewer (\BDE\EXAMPLES\SNIPIT).

DbiWriteBlock

To write multiple records to a table, the application creates a record buffer containing the records to be written, and calls DbiWriteBlock, passing the cursor handle of the table to be updated. The entire block of records in the record buffer is written to the specified table. This function is similar to calling DbiAppendRecord for multiple records.

Refer to the BLOCK.C code sample in the SNIPIT Code Viewer (\BDE\EXAMPLES\SNIPIT).

{button ,AL(`accessingtables')} Accessing and updating tables

Linking tables

Linked cursors allow you to create one-to-many (master-detail) relationships between tables. The cursors on two tables can be linked if the tables share a common field, which must be indexed in the detail table. Linking the cursors on a master table and a detail table forces the cursor on the detail table to make visible only those records containing a key value that matches the key value of the current record in the master table.

For example, a CUSTOMER table (master) and an ORDERS table (detail) share a common field called CUSTOMER_NO. If the current record in the master table has a CUSTOMER_NO of 1221, then the only records visible in the detail table are those that have a CUSTOMER_NO of 1221. In other words, the application sees only the orders that are associated with the current customer.

A master table can be linked to more than one detail table; a detail table can be linked to only one master table. A detail table can also be a master table, linked to other detail tables.

Links apply to all available driver types; they can be established between tables of the same or different driver types.

Setting up the link

To link two tables, the application follows these steps:

1 The application opens cursors on both tables. The detail table cursor must have a current active index on the field that will be used to link the cursors.

2 The application calls DbiBeginLinkMode for each cursor to be linked. The function returns a new cursor.

3 The application calls DbiLinkDetail, passing the cursor handles of both the master and detail tables. The data types of linked fields in master and detail records must match. This function links only on indexes that are applied on fields within the detail table (no expression indexes). For expression links in dBASE and FoxPro tables, call DbiLinkDetailToExp.

4 The two cursors are now linked. When the position of the master cursor changes, the corresponding detail cursor changes to show the applicable records.

Breaking the link

To break the link between the cursors, the application follows these steps:

1 The application calls DbiUnLinkDetail, passing the cursor handle of the detail table. The detail table is now unlinked to any master table, and its cursor displays the entire record range again.

2 The application calls DbiEndLinkMode for each linked cursor, passing it the cursor handle. A standard cursor handle is returned.

For an example, refer to the LNKCRSR.C code sample in the SNIPIT Code Viewer (\BDE\EXAMPLES\SNIPIT).

{button ,AL(`accessingtables')} Accessing and updating tables

Sorting tables

The BDE sort function DbiSortTable sorts an opened or closed table, either into itself or into a destination table. There are options to remove duplicates, to enable case-insensitive sorts, to sort on subsets of fields, and to enable special user-supplied comparison functions. The sort can be used with filters and field maps, and it is extremely fast. DbiSortTable is supported by SQL drivers, but a SQL table can serve only as a source table, not as a destination table.

The sort engine uses language driver-defined collating sequences to accommodate the character sets of different languages.

{button ,AL(`accessingtables')} Accessing and updating tables

Cached updates

The cached updates feature allows users to retrieve data from a database and make changes to that temporarily cached data without immediately writing to the actual underlying database. Users can make changes over a prolonged period with a minimum amount of resource locking at the actual database. After modifying the data, users call an update function to save their changes in the actual database. The update function sends to the database a batch of all inserts, deletes, and modifications made since the last update function call.

For a distinguishing comparison of the cached updates feature with transaction processing, see Transactions and cached updates

To support the cached updates feature, a special cursor layer installs on top of any cursor. Implementation and use of the cached updates features is described in subsequent sections

---

{button ,AL(`cachedupdates')} Cached updates topics

{button ,AL(`accessingtables')} Accessing and updating tables

The cached updates layer

The cached updates layer keeps track of all the changes that are made by users by intercepting various table methods such as insert record, modify record, and delete record. As users browse through the table, the cached updates layer recognizes which records are modified, deleted, and inserted. The layer presents those records to the users accordingly. The updates are not immediately sent to the underlying table; instead they are cached by the cached updates layer. No record locks are held until the clients decide to commit the updates. Then the locks are held only during the commit process.

Because no record locks are held before the commit operation, there is a risk that some records might be updated by other users. If a record modified by the cached updates layer is modified by other users before the cached updates layer commits its cached updates, an error is returned, indicating that the record has been modified by a different user.

For standard database tables, every non-blob field is used in determining the record modifications.

After making the required changes, the clients call DbiApplyDelayedUpdates either to commit or rollback the changes. The rollback operation quickly discards the update information from the cache.

If the user decides to commit the changes, the updates are applied to the database. As the updates are applied, referential integrity and data validation checks are made.

A callback mechanism informs the users about data integrity violations. This mechanism can skip a particular failed update or abort the entire commit operation. See Callback functions

Limitations

The cached updates layer works on one cursor at a time. If clients want to support cached updates on a form operating on more than one cursor, it is the client's responsibility to synchronize the updates on various cursors.

A few BDE calls that perform table operations are not supported by the cached updates layer, including data-ordering BDE calls, such as DbiSwitchToIndex. These can be done before entering the cached updates mode. The cached updates layer depends on bookmarks to keep track of modified records. Because bookmarks change when DbiSwitchToIndex is called, this BDE function is disabled.

{button ,AL(`cachedupdates')} Cached updates topics

{button ,AL(`accessingtables')} Accessing and updating tables

Using the cached updates mode

Use of the cached updates mode is a two-phase process.

Phase 1
The command dbiDelayedUpdPrepare causes all changes in the cache to be applied to the underlying data. Unless being used in a single-user environment, this command should always be used within the context of a transaction to allow for error-recovery in the event of an error during the update. Any errors encountered during this phase should be handled through callback functions.

Phase 2
The command dbiDelayedUpdateCommit performs the second phase. After successfully calling dbiDelayedUpdPrepare directly, follow it with the dbiDelayedUpdateCommit command. The internal cache is updated to reflect the fact that the updates were successfully applied to the underlying database (that is, the successfully applied records are removed from the cache).

Procedure
To start the cached updates mode:

1 Create the cached updates layer with a call to DbiBeginDelayedUpdates

```
DBIResult DBIFN EXPORT DbiBeginDelayedUpdates(phDBICur phDbiCur);
```

Note: The record buffer size will be different in cached updates mode. You should reallocate record buffers once the cached updates layer is installed.

2 Apply (commit) changes made to the cached updates cache with a call to DbiApplyDelayedUpdates

```
DBIResult DBIFN EXPORT DbiApplyDelayedUpdates (hDBICur
hDbiCur, DBIDelayedUpdCmd eUpdCmd);
  typedef enum
  {
        dbiDelayedUpdCommit = 0,
        dbiDelayedUpdCancel = 1
        dbiDelayedUpdCancelCurrent = 2
        dbiDelayedUpdPrepare = 3
  } DBIDelayedUpdCmd;
```

Note: When used on inserted, deleted, or modified records, the command dbiDelayedUpdCancelCurrent resets the current record to its original state.

3 Once the changes have been applied to the database, users can resume making changes to the database. They don't have to end the cached updates mode. After completing the next batch of modifications, DbiApplyDelayedUpdates can be called to apply those changes to the database or perform a rollback.

4 End the cached updates mode with a call to DbiEndDelayedUpdates

```
DBIResult DBIFN EXPORT DbiEndDelayedUpdates(phDBICur phDbiCur);
```

{button ,AL(`cachedupdates')} Cached updates topics

{button ,AL(`accessingtables')} Accessing and updating tables

Callback functions

A callback mechanism is provided when a failure to write a modified record to the database occurs. Because updates are not sent to the underlying table until the commit time, no errors (such as integrity constraint violations) are detected before the commit/prepare operation. If an error occurs at commit time, users are prompted with an error message describing the error. Applications should register a callback function for cached updates by using the DbiRegisterCallBack function (*ecbType* for this callback is cbDELAYEDUPD) to be notified of the errors during the commit.

The callback descriptor for cached updates is:

```
// type of delayed update object (cached updates callback)
typedef enum
  {
        delayupdNONE        = 0,
        delayupdMODIFY      = 1,
        delayupdINSERT      = 2,
        delayupdDELETE      = 3
  } DelayUpdErrOpType;
// cached updates callback descriptor.
typedef struct
  {
        DBIResult           iErrCode;
        DelayUpdErrOpType   eDelayUpdErrOpType;
        // Record size (physical record)
        UINT16              iRecBufSize;
        pBYTE               pNewRecBuf;
        pBYTE               pOldRecBuf;
  } DELAYUPDCbDesc;
```

In the callback descriptor, the eDelayUpdErrOpType indicates the operation type (such as insert, delete, or modify) and iErrCode indicates what sort of error has occurred during the eDelayUpdErrOpType operation.

Clients should allocate enough memory for *pNewRecBuf* and *pOldRecBuf*. Each record buffer should be at least as large as the cached update cursor's physical record buffer size. The new (after the update) and old (before the update) record buffers are returned to the clients through *pNewRecBuf* and *pOldRecBuf* record buffers.

Clients can respond to this callback function with the following return codes:

Return code	Resulting action
cbrABORT	The entire commit operation is aborted. cbrABORT is the default return code if no callback function is registered.
cbrSKIP	The failed update operation is skipped and the commit process continues with the remaining updates.
cbrCONTINUE	The failed update operation is skipped and the commit process continues with the remaining updates.
cbrRETRY	The failed update operation is tried again.
cbrPARTIALASSIST	The user-applied changes are kept in the cache. In this case, the user applies the changes to the

original table.

{button ,AL(`cachedupdates')} Cached updates topics

{button ,AL(`accessingtables')} Accessing and updating tables

Locking

The Borland Database Engine locking environment is a hierarchy consisting of three layers:

Session layer (Owns database handles, table cursors, acquired table locks, and record locks)
Database handle layer (Open mode limits the open mode of tables in the database)
Table cursor layer (Cursor open mode can limit access)

See the following topics on the layers and details about table locking:

{button ,AL(`locking')} Locking topics

Session layer

At the top of BDE's locking hierarchy is the session layer. The session indirectly controls some locks because it controls resources including database handles and table cursors. Multiple database handles can be opened in the same session; this is what gives the application access to different databases at the same time. When a session is closed, all resources attached to the session are closed and all locks owned by those resources are released.

The session directly owns table locks and record locks acquired by an application after the table has been opened. This means that if more than one cursor is open on the same table within a session, one cursor can release a lock that was acquired by another cursor. Sessions provide complete isolation from each other.

{button ,AL(`locking')} Locking topics

Database handle layer

One step down in BDE's locking hierarchy is the database handle layer. Although no locks are explicitly owned by the database handle, the share mode assigned to the database when it is opened determines whether tables within that database can be opened exclusively or shared. If the database is opened in share mode, then tables within that database can be opened either in exclusive or share mode. If the database is opened in exclusive mode, then all tables will be opened in exclusive mode, even if other users attempt to open the table in share mode.

When the database is closed, all resources allocated to the database handle are released, including table cursors and table locks owned by these cursors.

Also see Native handles

{button ,AL(`locking')} Locking topics

Table cursor layer

At the bottom of the BDE locking hierarchy is the cursor layer. Only locks placed on the table when it is opened with the DbiOpenTable function are owned by the cursor. If the table is opened in exclusive mode, no other user can access that table. An exclusive lock prevents any other user from accessing the table, or placing any type of lock on it. If the table is opened in share mode, other cursors can access the table and they can acquire read or write locks on the table.

When the cursor is closed, any exclusive lock placed on the table when it was opened is released.

{button ,AL(`locking')} Locking topics

Acquired locks

All locks acquired after the table is opened are owned by the session, rather than the cursor. There are several types of acquired locks:

- Acquired Table Locks
- Acquired Persistent Table Locks
- Record Locks

Checking a table's lock status

To check the acquired lock status of a table use DbiIsTableLocked. The application specifies the type of lock (no lock, read lock, or write lock) and the function returns the number of locks of that type placed on the table.

For dBASE, FoxPro, Access, and Paradox tables, to check whether the table is physically shared on a network or local drive and opened in share mode, use DbiIsTableShared. For SQL tables, this function can be used to check whether the table was opened in SHARE mode.

{button ,AL(`locking')} Locking topics

Acquired table locks

If an application needs to place a lock on a table that was opened in share mode, it calls the BDE function DbiAcqTableLock. If a lock cannot be obtained, an error is returned.

DbiAcqTableLock can place a read or a write lock on the table.

A write lock prevents other users from updating a table, so that updates can be made cleanly and without interference. Only one write lock can exist on a table at a time.

A read lock prevents anyone from updating the table and prevents other users from placing a write lock on the table, so that table data cannot change while you are reading it. Multiple read locks can co-exist.

If a driver does not support read locks, a read lock is upgraded to a write lock. For example, for dBASE tables, read locks are upgraded to write locks. For SQL tables, a write lock is the same as a read lock and behavior varies according to the server.

More than one lock can be acquired on the table.

Releasing acquired table locks

DbiRelTableLock is used to release a table-level lock placed with DbiAcqTableLock. For each lock acquired, a separate call to DbiRelTableLock is required to release it.

{button ,AL(`locking')} Locking topics

Acquired persistent table locks

A persistent lock can be placed even before the table has been created. For Paradox tables, this feature can be used to reserve a table name for future use. For SQL tables, BDE remembers that the lock was placed, and when the table is actually created during that connection, the table is locked (as long as the server supports table locks). These locks are acquired by the DbiAcqPersistTableLock function.

Releasing acquired persistent table locks

To release an acquired persistent lock, use the DbiRelPersistTableLock function.

{button ,AL(`locking')} Locking topics

Record locks

Applications can acquire record locks at record retrieval time. Most BDE functions that are capable of fetching a record provide the option of locking; for example, DbiGetNextRecord, DbiGetPriorRecord, and DbiGetRelativeRecord. The eLock parameter can be used to specify one of the following record locks:

Setting	Description
dbiNOLOCK	No lock; allows other users to read, update, and lock the record
dbiREADLOCK	Upgraded to a write lock
dbiWRITELOCK	Allows other users to read the record, but prevents them from updating the record, or placing a lock on the record

Paradox and dBASE lock managers both upgrade read locks to write locks; so, in effect, a record is either locked or not locked.

Because some BDE record-fetching functions perform operations other than locking, the order in which these operations occur can be significant:

- Cursor movement always occurs first.
- Paradox and dBASE drivers attempt to lock the record before filling the record buffer.
- SQL drivers fill the client's record buffer and then attempt to lock the record.

Note: Cursor movement occurs even if the lock fails. For example, if DbiGetNextRecord is called with a read lock, the cursor moves to the next record, and the lock is then attempted. If the record is already locked by another user, the lock attempt fails, but the cursor has changed position.

Maximum number of record locks for standard tables

Shared dBase table	100
Shared Paradox table	255

Checking a record's lock status

To check the lock status of a record, use DbiIsRecordLocked. This function returns the lock status of the current record; the lock status can be either locked or not locked.

Releasing record locks

The application can call the function DbiRelRecordLock to release the record lock on the current record or release all the record locks acquired in the current session. In addition, DbiModifyRecord provides an option to release the lock after the operation has completed.

{button ,AL(`locking')} Locking topics

Table lock coexistence

Each type of table-level lock placed on a table affects to some degree the access that other users have to the table. You can use a lock aggressively to prohibit other users from accessing a table, or you can use a lock defensively to prevent other users from placing locks that would limit your application's access to the table. The chart below shows the results of User 2's attempts to place table locks after User 1 has successfully placed each type of lock:

	User 2: Attempts to open the table in exclusive mode	Attempts to acquire a write lock	Attempts to acquire a read lock	Attempts to open the table in share mode
User 1: Opens the table in exclusive mode	**Fail**	**Fail**	**Fail**	**Fail**
Acquires a write lock	**Fail**	**Fail**	**Fail**	**Succeed**
Acquires a read lock	**Fail**	**Fail**	**Succeeds for Paradox Fails for dBASE or FoxPro**	**Succeed**
Opens the table in share mode	**Fail**	**Succeed**	**Succeed**	**Succeed**

{button ,AL(`locking')} Locking topics

Locking strategy

In choosing a locking strategy, you must consider both the application's need to keep other users from changing data, and the extent to which locking affects other users. You also need to consider the differences in rules used by the lock managers of each database system being accessed. SQL lock managers use a different set of locking rules from those used by dBASE and Paradox lock managers.

Using BDE, an application can update a table as long as it has read-write access to the table, and no other user has a lock on the table or record to prevent the update. However, it is necessary with dBASE, FoxPro, Access, and Paradox systems to lock the table or record before updating to ensure that the data in the table does not change while the application is in the middle of processing a retrieved record.

Note: With BDE, you can write your application as a multi-user application even if the database resides on a standalone PC, since locking overhead is marginal when data is local. This means that you can write a single application for both single-user and multi-user situations.

{button ,AL(`locking')} Locking topics

SQL-specific locking behavior

With dBASE, Paradox, FoxPro, and Access, a record lock prevents another user from updating the record. However, SQL deals with record locking differently. If a record in a SQL table is not in the record cache, the record is fetched from the server. The client has a local (cached) copy of the record, but that copy can become immediately out-of-date if another client retrieves the same record from the server, and modifies or deletes it before the first client is able to submit changes.

BDE SQL drivers (and some ODBC drivers) use optimistic locking. An optimistic lock actually allows the locked record to be updated by another user, but when the application that placed the lock attempts to update the record, BDE notifies the application that the record has changed and that the requested operation cannot be performed because someone else has modified the data. The application then has the option of inspecting the new record and deciding whether to submit its changes or not.

Optimistic locking avoids the performance and concurrency penalties incurred by a lock that ties up record access for the duration of time that it takes to complete a single user's modifications. At the same time, the application is protected from inadvertently changing data that has never been inspected.

You can use keyed updates to control optimistic locking for improved performance.

{button ,AL(`locking')} Locking topics

Transactions

SQL systems use transaction processing with commit and rollback; either the whole series of operations within the transaction is made permanent when the series completes, or the whole series is undone.

Transactions can be executed on all SQL platforms supported by BDE. A transaction is a series of programming commands that access data in the database. When the last of the series of commands has completed, the entire transaction is either committed or canceled. If it is committed, all changes performed within the transaction against the associated database are made permanent. If it is canceled, all changes performed against the associated database are undone.

Only one transaction can be active per connection to a SQL database. Any attempt to start an additional transaction before the first one terminates results in an error.

Also see SQL transaction control

Default transactions

SQL operations always take place within the context of a transaction. When no explicit transaction occurs, the SQL driver manages the SQL server transactions transparently for the client. Any successful modification of SQL server data is immediately committed to ensure its permanence in the database. Default transaction behavior would apply if you are using BDE with a SQL server, but you are not explicitly using transactions (that is, setting the operations off between DbiBeginTran and DbiEndTran).

Beginning a transaction

The DbiBeginTran function is used to begin a transaction. After a successful DbiBeginTran call, the transaction state is active. The application specifies the isolation level to be used for the transaction when DbiBeginTran is called. Possible values are:

- xilDIRTYREAD: Uncommitted changes can be read.
- xilREADCOMMITTED: Other transactions' committed changes can be read.
- xilREPEATABLEREAD: Other transactions' changes to previously read data are not seen.

Availability and behavior of isolation and read repeatability capabilities vary by SQL server.

Ending a transaction

DbiEndTran ends the transaction. The application specifies the transaction end type. Possible values are

- xendCOMMIT: Commit the transaction.
- xendCOMMITKEEP: For some SQL drivers, commit the transaction and keep cursors.
- xendABORT: Roll back the transaction.

Note: BDE cursors can remain active, even if the underlying SQL cursor is closed. BDE manages the re-opening of server SQL cursors transparently.

xendCOMMIT and xendABORT keep cursors if the driver and the database support keeping cursors. If the database does not support keeping cursors, four possibilities exist for each server cursor opened on behalf of the BDE user:

- A cursor for an open query with pending results is buffered locally. Other than prematurely reading the data, no visible effect remains.
- A cursor opened on a table supporting direct positioning is closed. No other behavior is affected.
- A cursor opened on a table that does not support direct positioning is opened initially in a different transaction or connection context, if the database supports this. This cursor remains open because it exists in a different context from the requested transaction.
- If none of the previous possibilities apply, the cursor is closed and subsequent access to

the BDE objects associated with the server cursor returns an error.

For an example, refer to the TRANSACT.C code sample in the SNIPIT Code Viewer (\BDE\EXAMPLES\SNIPIT).

{button ,AL(`transaction')} Transaction topics

Transactions on Paradox, dBASE, FoxPro, and Access

Transactions for Paradox, dBASE, FoxPro, and Access drivers (local transactions) enable you to roll back (revert) or commit updates to standard tables. This helps ensure that applications will perform updates in a consistent way.

When a local transaction is started on a standard database, updates performed against tables in that database are logged. Each log record contains the old record buffer of the record that is updated. When a transaction is active, the records with updates are locked and these locks are held until the transaction is either committed or rolled back.

- The Commit operation releases all locks that were held when that transaction was active.
- The Rollback operation reapplies the updates to the underlying tables to restore the original state of the database. Once the original state of the database is restored, the locks are released.

Limitations

The following limitations apply to local transactions:

- For standard databases (Paradox, dBASE, FoxPro, Access) there is no automatic crash recovery or DDL-related actions such as table create, restructure, index creation, table/index deletion, and so on.
- To perform transactions on a Paradox table, a valid index must exist. Data cannot be rolled back on Paradox tables lacking an index.
- Inserts rolled back on dBase and FoxPro tables are actually only soft deletes.
- Local transaction do not work for temporary tables.
- Local transactions do not work for the Text driver (ASCII files).
- For Access, if you do not supply a user name and use user-level security you can only have one active transaction. Transactions are occurring at the driver level not at the database level.

{button ,AL(`transaction')} Transaction topics

Transactions and cached updates

When a transaction is active, updates are immediately sent to the underlying tables. Thus errors (such as integrity constraint violations, and so on) are instantly reported to the clients. Because updates are immediately sent to the underlying tables, the updates are visible to other transactions. And because each modified record is locked, other users cannot interfere.

This behavior differs from that of the cached updates layer (batch or burst updates), where updates are not sent to the underlying table until the commit time. Hence no errors are reported until the commit time. No record locks are held until the user decides to commit the updates. The locks are held only during the commit process. If errors occur during the commit process, clients are given an option to abort the commit process. If clients abort a commit process, the original state of the table is restored.

The main advantage of the cached updates feature is that the locks are held only during the commit time, thereby increasing the access time of SQL servers for other system transactions. Transactions lock out other users after record is changed, and local transactions limit the user to changing only the maximum number of records that can be locked. Cached updates avoid these problems, but permit another user to change data underneath you.

These differences are summarized in the following table:

	Advantages	Disadvantages
Transactions	Updates immediately sent to tables. Modified records instantly visible to other users. Modified records are locked. Errors instantly reported.	Lock out other users once a record is modified. Local transactions limit users to changing only the maximum number of records that can be locked.
Cached updates	Locks are held only during commit time, increasing server access time for other transactions. Cached updates can be used with any cursor on a single table. Not limited to the maximum locks for dBASE (100) and Paradox (255) while modifying records. [When committing more than these maximums, an exclusive lock on the table is required to commit them.]	Permits another user to modify the records you are using without your realizing it. If errors occur during commit process, you may abort, reverting table to its original state, losing all modifications

For more information, see Cached updates.

{button ,AL(`transaction')} Transaction topics

Degree of transaction isolation

The degree of isolation provided by transactions on standard databases is Degree 0. This means that a transaction does not overwrite another transaction's dirty data.

Because only Degree 0 isolation is supported, transactions on standard databases are subject to the following limitations:

- **Possible lost updates**
  Two transactions could perform reads without locking records, that is, using dbiNOLOCK protocol. If these two transactions post their updates independently, the final result set might include only one transaction's changes, losing updates of the other transaction.
- **Transaction not isolated from dirty reads**
  A transaction T1 could read a record previously updated by another transaction T2 and make further modifications to that record. The record read by T1 might be inconsistent, because it is not the final update produced by T2. Hence the read of transaction T1 was a dirty read.
- **Unrepeatable reads not prevented**
  A transaction T1 reads a record twice, once before transaction T2 updates it and once after committed transaction T2 has updated it. The two read operations return different values for the record and the first read is not repeatable.

By using the appropriate locking mechanism during the updates, the clients can provide a higher degree of transaction isolation. For example, lost updates can be prevented if a transaction always gets a read lock on a record it is about to modify. No user-requested locks are promoted, that is, if a user requests to read a record by using DbiGetRecord with dbiNOLOCK protocol, that record is not locked and that read operation might be a dirty read. However, in the case of inserts and modifications, records are locked with dbiWRITELOCK and locks are held until that transaction ends.

The function DbiBeginTran supports several transaction isolation levels:

- xilDIRTYREAD (Uncommitted changes read),
- xilREADCOMMITTED (Committed changes, no phantoms), and
- xilREPEATABLEREAD (full read repeatability).

For SQL tables, appropriate transaction isolation levels can be requested depending on the destination SQL server capabilities. The xilREADCOMMITTED isolation level precludes lost updates and dirty reads. The xilREPEATABLEREAD isolation level prevents unrepeatable reads.

Limitation: Because the transaction feature for local (standard) database tables supports Degree 0 isolation, only the xilDIRTYREAD option is accepted in DbiBeginTran. If a higher degree of isolation is requested, an error message is returned. For the same reason, xendCOMMITKEEP is not supported by DbiEndTran.

Because all updates are atomic, users will be informed about the lock conflicts immediately. No deadlock detection is performed. A deadlock occurs when each of two transactions waits for locks held by the other. If there are any lock conflicts between different transactions, an error message is returned to the clients. When a deadlock occurs, it is up to the clients to decide which transactions to rollback.

{button ,AL(`transaction')} Transaction topics

Using transactions

BDE provides two API functions: DbiBeginTran, to begin transactions and DbiEndTran, to end transactions:

```
DBIResult DBIFN DbiBeginTran (      // Begin a transaction
     hDBIDb         hDb,            // Db handle
     eXILType       eXIL,           // Transaction isolation level
     phDBIXact      phXact          // Returned. Xact handle
  );
  // Commit or rollback a basic transaction. If hXact is
  // given, hDb is ignored. If hXact == 0, hDb must be given.

DBIResult DBIFN DbiEndTran (        // End a transaction
     hDBIDb         hDb,            // Database handle
     hDBIXact       hXact,          // Xact handle
     eXEnd          eEnd            // Xact end type
  );
  // The transaction model being discussed here supports only
  // xilDIRTYREAD isolation level.
typedef enum                    // Transaction isolation levels
  {
        xilDIRTYREAD,           // Uncommitted changes read
        xilREADCOMMITTED,       // Committed changes, no phantoms
        xilREPEATABLEREAD       // Full read repeatability
  } eXILType;
typedef enum                    // Transaction end control
  {
        xendCOMMIT,             // Commit transaction
        xendCOMMITKEEP,         // Commit transaction, keep cursors
        xendABORT               // Rollback transaction
  } eXEnd;
```

The following results occur when there are active transactions:

1. If there are active transactions in a session, DbiCloseSession closes that session and its active transactions are rolled back. Similarly, DbiExit rolls back the active transactions present in the system.
2. In the case of standard databases (local transactions), DbiModifyRecord, DbiInsertRecord, and DbiDeleteRecord are intercepted to perform the transaction logging. A separate log is associated with each transaction. The log is maintained as long as the transaction is active. It is destroyed once the transaction commits or rolls back.

{button ,AL(`transaction')} Transaction topics

Querying databases

The BDE API enables the client to use SQL or Query by Example (QBE) to access dBASE, FoxPro, Access, and Paradox tables (standard databases) as well as server-based SQL tables.

A group of BDE query interface functions is provided for passing either SQL Queries or QBE queries to both server-based and PC-based sources.

{button ,AL(`querying')} Querying topics

SQL queries

The common query engine uses a convenient subset of SQL to access dBASE, FoxPro, Access, and Paradox tables. This subset can also be used to join server-based SQL tables with these tables. The appropriate BDE driver must be installed to allow server-based SQL access.

To exploit the full functionality of the server, you can use your server's dialect of SQL. Use passthrough SQL to send native SQL statements directly to your database server to be executed there. Queries executed in the native dialect might not result in updateable cursors. If the appropriate BDE driver is installed, the BDE query interface functions can also be used to pass SQL statements to the server for processing, in the native dialect of a server-based system, such as Oracle or Sybase.

QBE queries

Query By Example (QBE) allows uniform access to data in Paradox, FoxPro, Access, or dBASE tables and tables in server-based databases. BDE supports the full QBE language as defined by Paradox DOS and Paradox for Windows. When QBE is executed with a SQL data source, the QBE query is translated to SQL and sent to the server; the resulting cursor is not updateable.

Querying Paradox, dBASE, FoxPro, and Access tables

The common query engine enables BDE application developers to access tables in standard databases using either the SQL or QBE languages. Two categories of SQL statements ("Local SQL") are supported for tables in standard databases:

- Data Definition Language (DDL)
- Data Manipulation Language (DML)

For more specific information about the BDE implementation of the SQL-92 specification, see the Local SQL Guide.

Naming conventions

When writing SQL statements to be used with dBASE, FoxPro, Access, and Paradox tables, observe the following naming conventions:

Table names

Table names that include a period (.) must be placed in either single or double quotation marks. For example:

```
select * from 'c:\sample.dat\table'
select * from "table.dbf"
```

Table names can include BDE style aliases. For example,

```
select * from ":data:table"
```

Names that are keywords must be placed in quotation marks. For example,

```
select passid from "password"
```

Names that have spaces must be placed in quotation marks. For example,

```
select * from "old table"
```

Field names

Field names that have spaces must be placed in quotation marks. For example,

```
select e."Emp Id" from Employee e
```

Field names that are keywords must be placed in quotation marks. For example,

```
select t."date" from Table t
```

Field names that are placed in quotation marks must have a table reference.

Data Manipulation Language

The following DML clauses are supported:

`SELECT, WHERE, ORDER BY, GROUP BY, UNION, and HAVING`

The following aggregates are supported:

`SUM, AVG, MIN, MAX, COUNT`

Note: The field type returned by aggregator functions is type DOUBLE.

The following operators are supported:

`+, -, *, /, =, <, >, <>, <=, >=, NOT`

UPDATE, INSERT, DELETE operations are fully supported to SQL 92 entry level.

For example:

```
DELETE FROM "Current Cust.db" C
WHERE C."CustID" IN
  (SELECT O."CustID"
  FROM "Old Cust.db" O)
```

Also supported

- Subqueries are supported in SELECT, WHERE, and HAVING clauses. In addition to scalar comparison operators (=, <, > ...), additional predicates IN, ANY, SOME, ALL, and

EXISTS are supported.

- Complex aggregate expressions are supported, including scalar expressions with both aggregation and arithmetic. For example:

```
SUM( Field * 10 )
SUM( Field ) * 10
SUM( Field1 + Field2 )
```

- Constructs such as SUM(MIN(Field)) are supported in projections.
- You can constrain any updateable query by setting the query statement property stmtCONSTRAINED to TRUE before execution. A error will then be returned whenever a modify or insert would cause the new record to disappear from the result set. Refer also to record integrity constraints.

Data Definition Language

The DDL syntax for Paradox, dBASE, FoxPro, and Access tables is restricted to CREATE TABLE (or INDEX), DROP TABLE (or INDEX). For example:

```
create table parts ( part_no char(6), part_name char(20) )
```

The following example demonstrates how SQL DDL can be executed through BDE:

```
hDBICur hCur;
pBYTE szQuery ="create table 'c:\\example\\test.dbf' "
               "( fld1 int, fld2 date)";
rslt = DbiQExecDirect(hDb, langSQL, szQuery, &hCur);
```

For data mappings used by CREATE TABLE and more examples, see the Local SQL Guide.

{button ,AL(`querying')} Querying topics

Querying different databases

Through the BDE interface, the application developer can use SQL to join tables from different data sources (for example, a Paradox, InterBase, and Sybase table could all participate in a SQL query). These are called "heterogeneous joins." See Local SQL Guide

The following SQL statement shows a join of three tables from different platforms, by using aliases:

```
select distinct c.cust_no, c.state, o.order_no, i.price
  from    ':Local_alias:customer.db' c,
          ':IB_alias:order' o,
          ':SYB_alias:lineitem' i
  where         o.cust_no = c.cust_no and
                o.order_no = i.order_no
```

{button ,AL(`querying')} Querying topics

Executing queries directly

Use DbiQExecDirec for simple queries, where no special preparation is necessary. This function immediately prepares and executes a SQL or QBE query and returns a cursor to the result set, if one is generated. The application passes the database handle, specifies whether the query language is QBE or SQL, and passes the formulated query string.

With SQL query language, if the specified database handle refers to a server database, the SQL dialect native to that server is expected. If the database handle refers to a standard database, the SQL statement is limited to the subset supported by the common query engine.

The following example shows how a SQL query is executed with the function DbiQExecDirect:

```
DBIResult   rslt;
hDBICur       hCur;
pBYTE         szQuery =   "Select t.name, t.age "
                                "from EMPLOYEE t "
                                "where t.age > 30 "
                                "and t.salary > 1000000 ";
rslt = DbiQExecDirect(hDb, qrylangSQL, szQuery, &hCur);
```

{button ,AL(`querying')} Querying topics

Executing queries in stages

Some queries require a statement handle and need to be executed in stages. A statement handle is required if the application needs to control the table type of the result set, to express preference over the degree of liveness of data, or to bind parameters for queries. The application uses a separate function call for each stage:

1 To obtain a new statement handle, call DbiQAlloc.

2 To change properties in the statement handle, call DbiSetProp. At this point you can also indicate whether you want the result set to be "live," that is, modifiable.

3 To prepare the query, call DbiQPrepare.

4 To execute the prepared query, call DbiQExec.

5 To free resources bound to the query, call DbiQFree.

DbiQAlloc

This function allocates a statement handle required for prepared query functions. It specifies the database handle and whether the query language is QBE or SQL, returning a statement handle for the prepared query. DbiQAlloc is the necessary first step in all prepared queries.

DbiSetProp

DbiSetProp is used to set a property of an object to a specified value. In this case, the object is the statement handle returned by DbiQAlloc. The property to be set can be the result table type, degree of liveness, or query mode for binding parameters. The following examples show how values are set for these properties:

```
DbiSetProp(hStmt, stmtANSTYPE, (UINT32) szPARADOX);
DbiSetProp(hStmt, stmtLIVENESS, (UINT32) wantLIVE);
```

DbiQPrepare

This function is used to prepare a SQL or QBE query for subsequent execution. It accepts a handle to a statement containing the prepared query.

Live and canned result sets

The last example above shows how you can specify your preference for live or canned result sets during query execution. A canned result set is like a snapshot or a copy of the original data selected by the query. In contrast, a live result set is a view of the original data; specifically, if you modify a live result set, the changes are reflected in the original data.

When you specify your preference for a live result set, the Query Manager attempts to give you a live result set. However, no guarantee can be made that the resulting result set will indeed be live. After the query has executed and a result set has been returned, you can check to see if it is live by examining the cursor property *bTempTable*. If TRUE, the result set is a temporary table, hence a copy (canned); otherwise, the result set is live.

SQL queries against SQL servers return an error if the result cannot be made live. *bTempTable* is valid for local queries.

The possible values for liveness are:

Value	Description
wantCANNED	Indicates preference for a canned result set (this request is always honored)
wantLIVE	Indicates preference for a live result set
wantSPEED	Directs the query manager to decide, based on which method is probably fastest
wantDEFAULT	Same as wantCANNED

DbiQExec

DbiQExec executes the previously prepared query identified by the supplied statement handle and returns a cursor to the result set, if one is generated.

For all queries, remote and local, the same prepared query can be executed several times, but only after any pending results have been read or discarded (by using DbiCloseCursor on the answer set cursor).

DbiQFree

This function is always used as the final step in executing prepared queries to free all system resources allocated during preparation and use of a query. If cursors are associated with an outstanding result set produced by execution of the statement, the cursors remain valid and the dependent statement resources are not released until the last cursor has been closed or the result set is read to completion, whichever happens first.

{button ,AL(`querying')} Querying topics

Getting and setting properties

Each BDE object is defined by a set of properties. The properties defining an object depend on the object's type. For example, a session is a BDE object, and its properties include sesMAXPROPS, sesSESSIONNAME, and sesCFGMODE2. Each type of object has its own set of properties, as listed in Object Properties.

Values are initially assigned to properties when an object is created. For example, the name of the table is assigned to the curTABLENAME property of the cursor object when the table is opened with DbiOpenTable.

Values of some properties can be changed with the BDE function DbiSetProp. To reset a property, the application passes the object handle, the name of the property to be changed, and the new value of the property.

To retrieve an object's current property settings, use DbiGetProp.

To retrieve an object's handle, use DbiGetObjFromName.

To retrieve a cursor's database handle, use DbiGetObjFromObj.

This example illustrates a method for getting the table name/type when all that is available is the table cursor:

```
UINT16      iLen;
DBITBLNAME  tblName;
DBINAME     tblType, dbName;
// The table cursor gives you access to the table
// name and the table type.
DbiGetProp(hCursor, curTABLENAME, (pVOID) tblName,
           sizeof(tblName), &iLen);
DbiGetProp(hCursor, curTABLETYPE, (pVOID) tblType,
           sizeof(tblType), &iLen);
// You can also access database properties (such as
// the name of the database associated with the cursor).
DbiGetProp(hCursor, dbDATABASENAME, (pVOID) dbName,
           sizeof(dbName), &iLen);
```

Object properties

Each BDE object is defined by its own set of properties as described in the following table.

Note: Not all drivers support all properties. Also, some properties are valid only at certain times. For example, stmtLIVENESS is valid only before DbiQPrepare.

Properties	System	Session	Database	Driver	Cursor	Statement
sysMAXPROPS	X	X	X	X	X	X
sysLOWMEMUSAGE	X	X	X	X	X	X
sesMAXPROPS		X	X		X	X
sesSESSIONNAME		X	X		X	X
sesNETFILE		X	X		X	X
sesCFGNAME		X	X		X	X
sesCFGUPDATE		X	X		X	X
sesCFGMODE2		X	X		X	X
dbBATCHCOUNT			X			
dbBLOBCOUNT			X			
dbBLOBSIZE			X			
dbMAXPROPS			X		X	X
dbDATABASENAME			X		X	X
dbDATABASETYPE			X		X	X
dbASYNCSUPPORT			X			
dbPROCEDURES			X			
dbDEFAULTTXNISO			X			
dbNATIVEHNDL			X			
dbNATIVEPASSTHRUHNDL				X		
dbUSESCHEMAFILE			X			
dbSERVERVERSION			X	X		
dbTRACEMODE						
drvMAXPROPS			X	X	X	
drvDRIVERTYPE				X	X	
drvDRIVERVERSION				X	X	
cfgREFRESH	X	X				
curGETEXTENDEDINFO					X	
curMAXPROPS					X	
curMAXROWS			X	X	X	
curTABLENAME					X	
curTABLETYPE					X	
curTABLELEVEL					X	
curFILENAME					X	
curXLTMODE					X	
curSEQREADON					X	

curONEPASSON	X	
curUPDATETS	X	
curSOFTDELETEON	X	
curLANGDRVNAME	X	
curPDXMAXPROPS	X	
curDBMAXPROPS	X	
curINEXACTON	X	
curNATIVEHNDL	X	
curUPDLOCKMODE	X	
stmtMAXPROPS		X
stmtPARAMCOUNT		X
stmtUNIDIRECTIONAL		X
stmtANSTYPE		X
stmtLIVENESS		X
stmtQRYMODE		X
stmtBLANKS		X
stmtDATEFORMAT		X
stmtNUMBERFORMAT		X
stmtAUXTBLS		X
stmtTBLVECTOR		X
stmtALLPROPS		X
stmtALLPROPSSIZE		X
stmtANSNAME		X
stmtNATIVEHNDL		X
stmtCURSORNAME		X
stmtROWCOUNT		X
stmtCONSTRAINED		X
stmtFIELDDESCS		X
stmtCURPROPS		X

Descriptions

sesCFGUPDATE	When set to ON, the session receives a copy of any alias or driver additions that are applied to other sessions. Set to OFF to disable this behavior.
sesCFGNAME	Read only property that returns the name of the configuration file in use by the session.
dbBLOBCOUNT	Read only property that returns the current setting of BLOBS TO CACHE.
dbBLOBSIZE	Read only property that returns the current setting of BLOB SIZE.
CfgREFRESH	Specifies whether the BDE retrieves virtual ODBC information each time it is needed (TRUE) or only at the start of the BDE session (FALSE), which improves performance.

Retrieving schema and system information

A set of BDE functions return schema or system information. Some functions, in the format DbiOpen*xxx*List, can be used to return a cursor to an in-memory table whose records contain the requested information. Other functions in the format DbiGet*xxx*Descs return information directly to descriptor structures and arrays supplied by the application. In each of the topics below you will find a chart of record structures of the virtual table returning the information.

- DbiOpenList Functions
  Return a cursor handle to an in-memory table listing the requested information. This topic includes an example that illustrates the use of a static structure as the record buffer
- DbiGetDescs Functions
  These function calls return descriptive information. Inquiry function structures are supplied by the application. This topic includes an example showing how to retrieve all the index descriptors with one function call.

DbiOpenList functions

One series of inquiry function calls, in the form DbiOpen*xxx*List, return a cursor handle to an in-memory table listing the requested information. The cursor to an in-memory table is read-only, so that the application is prohibited from updating the table. Information can be retrieved from the in-memory table in the normal way, by preparing the record buffer, positioning the cursor, fetching each record into the record buffer, and using DbiGetField. Or each record can be read into the predefined structures assigned to the function. These structures are listed in the IDAPI.H file.

List function	Record structure of the virtual table returning the information
DbiOpenDatabaseList	DBDesc
DbiOpenDriverList	The virtual table contains only one CHAR field.
DbiOpenFamilyList	FMLDesc
DbiOpenFieldList	FLDDesc
DbiOpenFieldTypesList	FLDType
DbiOpenFileList	FILEDesc
DbiOpenFunctionArgList	DBIFUNCArgDesc
DbiOpenFunctionList	DBIFUNCDesc
DbiOpenIndexList	IDXDesc
DbiOpenIndexTypesList	IDXType
DbiOpenLdList	LDDesc
DbiOpenLockList	LOCKDesc
DbiOpenRintList	RINTDesc
DbiOpenSecurityList	SECDesc
DbiOpenTableList	TBLBaseDesc, TBLExtDesc, TBLFullDesc
DbiOpenTableTypesList	TBLType
DbiOpenUserList	USERDesc
DbiOpenVchkList	VCHKDesc

Example

This example illustrates the use of a static structure as the record buffer:

```
DBIResult   rslt;
hDBICur     hListCur;
IDXDesc     idxDesc;
// Open a schema table which will contain 1 record for each
// index currently available for the given table.
```

```
rslt = DbiOpenIndexList(hDb, "Sample", szPARADOX, &hListCur);
if (rslt == DBIERR_NONE)
{
    // Use a loop to retrieve each index descriptor
    while (DbiGetNextRecord(hListCur, dbiNOLOCK,
                            (pBYTE) &idxDesc, NULL)
           == DBIERR_NONE)
    {
...
}
    // Close the index list
    DbiCloseCursor(&hListCur);
}
```

DbiGetDescs functions

Inquiry function structures are supplied by the application. These function calls return descriptive information.

List function	Record structure of the virtual table returning the information
DbiGetDatabaseDesc	DBDesc structure
DbiGetDriverDesc	DRVType structure
DbiGetFieldDescs	Array of FLDDesc structures
DbiGetFieldTypeDesc	FLDType structure
DbiGetIndexDesc	IDXDesc structure
DbiGetIndexDescs	Array of IDXDesc structures
DbiGetIndexTypeDesc	IDXType structure
DbiGetTableTypeDesc	TBLType structure
DbiQGetBaseDescs	STMTBaseDesc structure

Example

The following example shows how to retrieve all the index descriptors with one function call:

```
    DBIResult   rslt;
         hDBICur     hCursor;
         CURProps    curProps;
         pIDXDesc    pIdxArray;
       // Open the table
       rslt = DbiOpenTable(hDb, "Sample", szPARADOX, NULL, NULL, 0,
                           dbiREADWRITE, dbiOPENSHARED, xltFIELD,
                           TRUE, NULL, &hCursor);
       if (rslt == DBIERR_NONE)
       {
           // Get the properties for the cursor
           DbiGetCursorProps(hCursor, &curProps);
           // Allocate the buffer for the index descriptors
           pIdxArray = (pIDXDesc) malloc(sizeof(IDXDesc) *
                                         curProps.iIndexes);
  // Get the indexes
           rslt = DbiGetIndexDescs(hCursor, pIdxArray);
           if (rslt == DBIERR_NONE)
           {
          ...
          }
          // Clean up
           free((pCHAR) pIdxArray);
           DbiCloseCursor(&hCursor);
       }
```

Creating tables

The application can create permanent tables by using the BDE function DbiCreateTable. It can also create temporary tables with DbiCreateTempTable and in-memory tables with DbiCreateInMemTable. To see code samples of creating tables, run the SnipIt Code Viewer and select Table: Create dBASE or Table: Create Paradox.

Permanent tables

Permanent tables are named and are saved to disk. To create a permanent table, the application first creates a field descriptor structure FLDDesc for each field in the table and an index descriptor structure IDXDesc for each index. For SQL and Paradox tables, the application can also define a descriptor structure for each validity check VCHKDesc. For Paradox and SQL tables, the application can define a descriptor structure for each referential integrity check RINTDesc, and each security check SECDesc to be enforced.

Next, the application creates a table descriptor structure CRTblDesc defining general attributes of the table, and supplying pointers to arrays of field, index, validity, referential integrity and security descriptor structures previously created. Finally, the application calls DbiCreateTable, passing the CRTblDesc structure.

Specifying optional parameters

When creating a Paradox, dBASE, FoxPro, or Access table, optional driver-specific parameters may be included in the last three fields of the CRTblDesc structure. To retrieve a list and description of these optional parameters for a driver, the application can call DbiOpenCfgInfoList, supplying the path of the driver's table create options in the configuration file. This function returns an in-memory table with information about relevant optional parameters, as well as the default values for these parameters. For example, the Table Level is an optional parameter for dBASE and Paradox tables.

Temporary tables

A temporary table is deleted when the cursor is closed. The application can create a temporary table in the same way it creates a permanent table except that it calls DbiCreateTempTable instead of DbiCreateTable. See "Permanent Tables" above for a description of the descriptor structures used to create a table.

For Paradox, dBASE, and FoxPro only, a temporary table can be made into a permanent table by calling DbiMakePermanent while the cursor is still open and supplying a table name, or calling DbiSaveChanges.

In-memory tables

An in-memory table cannot be saved as a permanent table. The application can create an in-memory table by calling DbiCreateInMemTable, and supplying an array of field descriptor structures FLDDesc. The table descriptor CRTblDesc is not used. Only BDE logical types are supported.

{button ,AL(`creatingtables')} Creating tables topics

Integrity constraints

When creating a table by using the BDE function DbiCreateTable, you can use integrity constraints to ensure that references in the key fields of secondary tables (in the same database) or foreign tables (in another database) are maintained to key fields in a primary table. For example, if several tables have keys referencing the primary key Customer ID in the Customer table, then this dependency must be checked so that referenced customer IDs cannot be deleted, thereby orphaning records in secondary or foreign tables.

Primary key and foreign key integrity constraints are implemented wherever supported by SQL servers such as:

- Sybase system 10
- Microsoft SQL server 6.0
- InterBase 5
- Oracle 6.0 (syntax only, not enforced)
- Oracle 8
- DB2 2.1.1
- Informix 7.11
- Informix 9

Primary key support

1. Decide which fields or set of fields will act as the primary key for the table to be created. For a dBASE table, choose the index that will act as the primary key for the table to be created.
2. Put this information in an IDXDesc structure with these columns.
3. Set IDXDesc.bPrimary = TRUE;
4. Attach the IDXDesc structure to a CRTBLDesc.pidxDesc pointer.

Primary key columns must be NOT NULL, which means you should have VCHKDesc for each column with VCHKDesc.bRequired = TRUE. The exception is Paradox which can have one blank record.

There can be only one primary key per table.

A table with primary key constraint (table level) is created and an unique index (ascending) on these columns is also created. For dBASE, any index can be used, whether ascending or descending. For remote databases, this index can neither be added nor dropped by using CREATE INDEX or DROP INDEX. The index will be created when table is created and will go away when the table is dropped. (In Local SQL, you can drop the primary index by using this statement: "DropIndex TABLENAME.PRIMARY".)

Foreign key support

1 Decide which table (the *other* Table) is going to be referenced by the table (*this* table) being created. It could be the same table if supported by the server.

2 Decide which columns of *this* table reference the *other* tables columns

3 Decide what should be the referential action for Delete. If cascading is required, set RINTDesc. eDelOp = rintCASCADE. (This is supported by ORACLE 7.x and Sybase System 10.)

4 Put this information in the RINTDesc structure.

5 Attach the RINTDesc to CRTBLDesc.printDesc pointer

There can be more than one referential (foreign key) integrity constraint.

Note: Some servers, such as InterBase 4.0, create an index on referencing columns of this table.

Schema retrieval/integrity constraints

Primary key: Any of index retrieval functions and check if pIdxDesc->bPrimary = TRUE.

Foreign Key: Use BDE function DbiGetRINTDesc.

{button ,AL(`creatingtables')} Creating tables topics

Modifying table structure

After a table has been created, the application can modify it using BDE functions in the following ways:

- Add, delete, or regenerate indexes
- Restructure the table

Adding indexes

The application can add an index to a table by calling DbiAddIndex and supplying the IDXDesc structure, with the appropriate fields filled in (the fields required vary by driver and index type). For a complete description of these fields by driver and index type, see DbiAddIndex.

Deleting and regenerating indexes

The application can delete an index by calling DbiDeleteIndex. The application can either specify the table by name or by opening a cursor on the table. The index to be deleted cannot be active.

The application can bring dBASE, FoxPro, Access, or Paradox indexes up to date by calling either of two BDE functions. DbiRegenIndex regenerates a single out-of-date index. DbiRegenIndexes regenerates multiple out-of-date indexes on a table. The application specifies the index name.

Restructuring a table

Currently, for Paradox, dBASE, FoxPro, and Access tables only, the application can call DbiDoRestructure to modify existing field types or sizes, add new fields, delete a field, rearrange fields, change indexes, security passwords, or referential integrity.

The application passes the table descriptor structure, CRTblDesc.

{button ,AL(`creatingtables')} Creating tables topics

Using callbacks

Sometimes an application needs to be notified of a specific type of database engine event in order to complete an operation or to provide the user with information. The advantage of using callbacks is that BDE can get a user's response without interrupting the normal application process flow.

The following rules must be strictly followed in a callback function:

- No other BDE calls can be made inside the callback function.
- BDE is not re-entrant during the callback function. The application must not yield to Windows within the callback function. For example, if the application displays a dialog box in Windows inside a callback function, the dialog box must be System Modal.

Types of callbacks

The application can choose to be notified of many different types of events, depending on which callback type it registers. The application can specify the following callback types in a call to DbiRegisterCallback.

Callback	Description
cbGENPROGRESS	Informs applications about the progress made during large batch operations.
cbRESTRUCTURE	Supplies information about an impending action and requests a response from the caller.
cbBATCHRESULT	Batch processing results.
cbTABLECHANGED	Notifies user that table has changed.
cbCANCELQRY	Allows user to cancel a Sybase query.
cbINPUTREQ	A BDE driver requests input from user.
cbDBASELOGIN	Enables clients to access encrypted dBASE tables.
cbFIELDRECALC	Field(s) recalculation
cbTRACE	Trace
cbDBLOGIN	Database login
cbDELAYEDUPD	Cached updates callback
cbNBROFCBS	Number of callbacks

Callback function declarations and associated parameter lists, function return types, and callback data types are defined in the header file IDAPI.H, which is the application interface to Borland Database Engine.

Return codes

The application responds to a callback by issuing a return code that commands an appropriate action:

Return code	Action description
cbrUSEDEF	Take default action
cbrCONTINUE	Continue
cbrABORT	Abort the operation
cbrCHKINPUT	Input given

cbrYES	Take requested action
cbrNO	Do not take requested action
cbrPARTIALASSIST	Assist in completing the job

Registering a general progress report callback

Suppose that an application must copy a million-record table, and you want to periodically display a progress report on screen indicating the progress of the copy operation. You would use the following procedure:

1 Write the body of the of the progress callback function, declaring it with an associated predefined parameter list:

```
typedef CBRType far *pCBRType;
 typedef CBRType (DBIFN * pfDBICallBack)
 (
 CBType          ecbType,             // Callback type
 UINT32          iClientData,         // Client callback data
 pVOID           pCbInfo              // Call back info/Client
 Input
 );
```

2 The application allocates memory for the buffer *pCbBuf* to be used for passing data back and forth from the application to the function, and pointing to a CBPROGRESSDesc structure.

```
typedef struct
 {
 INT16          iPercentDone;        // Percentage done
 DBIMSG         szMsg;               // Message to display
 } CBPROGRESSDesc;
 typedef CBPROGRESSDesc far * pCBPROGRESSDesc;
```

3 To register a callback, the application calls DbiRegisterCallback passing cbGENPROGRESS as the value for ecbType.

4 The application issues a call to DbiBatchMove.

5 BDE returns either a percentage done (in the *iPercentDone* parameter of the CBPROGRESSDesc structure), or a message string to display on the status bar. The application can assume that if the *iPercentDone* value is negative, the message string is valid; otherwise, the application needs to consider the value of *iPercentDone*. The message string format is <Text String><:><Value> to allow easy international translations. For example: `Records copied: 250`

6 To continue processing the application returns the code cbrUSEDEF. The application can abort the BDE function call in progress by returning cbrABORT.

Data source independence

You can use these techniques to achieve data source independence:

- Qualify table names through aliases defined in the configuration file (or by supplying fully qualified path names).
- Use only BDE logical data types.
- Use the generic subset of SQL supported by the shared query engine.

The application can determine which aliases are available to it by calling the BDE function DbiOpenDatabaseList. This function lists all of the database aliases in the configuration file (IDAPI.CFG).

Filtering records

This section is an overview of how to create an expression tree used in DbiAddFilter. This is a fairly complex undertaking. You would want to write an expression tree only when you need to efficiently generate a highly constrained view of the data in a table, by qualifying multiple, *unindexed* fields. (If the fields were all indexed, you might be able to use DbiSetToKey more easily.)

A filter is a mechanism that lets you qualify the data that a cursor displays, relieving the application of the task of testing each record. For a basic example, let's say you want to open a customer table but display only those customers living in California. To use a filter to accomplish this, you can write your application to define a filter for a cursor open on the Customer table, where customer.state= CA. When the filter is activated, the BDE retrieves only those records that meet this condition, so your application can view and process only those records. For example, when your application calls DbiGetNextRecord, any records where the customer is not a resident of California are skipped.

To define a filter, the application calls DbiAddFilter, passing the cursor handle and the filter condition specification. The function returns the filter handle to the application. The DbiAddFilter parameter *pcanExpr* points to an expression tree of type pBYTE. The application can use the expression tree to specify the filter condition.

The advantage of using an expression tree to define a filter condition is that BDE can use it to optimize the filtering operation. The level of optimization depends on the driver's level of support for parsing the expression tree.

After defining the filter, the filter must be activated with DbiActivateFilter.

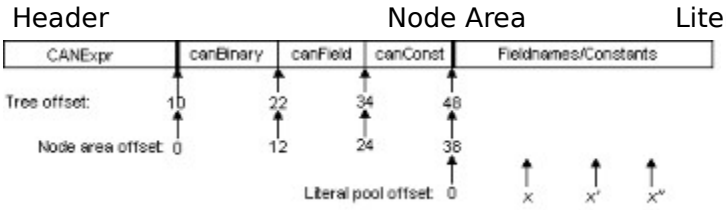
{button ,AL(`filtering')} Filtering topics

Using an expression tree

An expression tree is a filter expression of type pBYTE cast as a pCANExpr. It is a three-part block of memory, consisting of:

- Header
  A CANExpr structure defining size, number of nodes, and offsets.
- Node Area
  A series of conditional (operator and operand) branches in the tree, "nodes," defining the filter's tree of conditions. Operand nodes point to offset locations of field names or constants stored in the literal pool area.
- Literal Pool Area
  Used to store the field names pointed to by each field node and the constant values pointed to by each constant node.

Note that the header consists of a CANExpr structure 10 bytes in length, hence the Node Area begins at offset 10:



The first node in the node area is the canBinary specifying the operand:

canExpr.iFirstNode = 10 (where 10 is offset for entire expression tree)

canExpr.iLiteralStart = 48 (where 48 is offset for entire expression tree)

canBinary.Operand1 = 12 (where 12 is offset into Node Area)

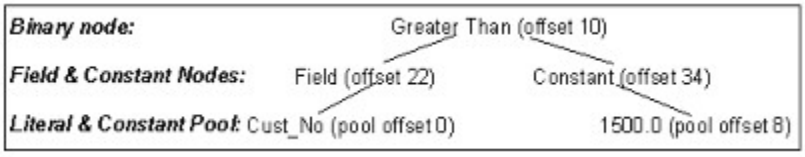
canBinary.Operand2 = 24 (where 24 is offset into Node Area)

canField.iNameOffset = 0 (where 0 is offset into Literal Pool)

canConst.iOffset = strlen(<fieldName>)+1 -- (where the constant value appears just after the field name in the Literal Pool)

Example

Normally you would use an expression tree to tightly focus a view by using a tree of conditions that could be quite complex. For the sake of clarity, in this example, we want a simple filter to display only those records where "CUST_NO>1500". Our task is to create an expression tree CUST_NO > 1500.00 to pass to DbiAddFilter. The following chart represents this expression tree:



The same expression tree is defined in C as a parameter to be passed to DbiAddFilter. The following example assumes that the compiler allocates consecutively declared variables in physically contiguous memory:

```
void
  Filter (void)
  {
      hDBIDb          hDb = 0;            // Handle to the database.
      hDBICur         hCur = 0;           // Handle to the table.
      DBIResult       rslt;               // Return value from IDAPI
  functions.
```

```
    pBYTE           pcanExpr;           // Structure containing filter
info.
    hDBIFilter      hFilter;            // Filter handle.
    UINT16          uSizeNodes;         // Size of the nodes in the tree.
    UINT16          uSizeCanExpr;       // Size of the header information.
    UINT32          uSizeLiterals;      // Size of the literals.
    UINT32          uTotalSize;         // Total size of the filter
expression.
    UINT32          uNumRecs = 10;      // Number of records to display.
    CANExpr         canExp;             // Contains the header information.
    struct {
        CANBinary BinaryNode;
        CANField  FieldNode;
        CANConst  ConstantNode;
    }
    Nodes = {                           // Nodes of the filter tree.
    {
        // Offset 0
        nodeBINARY,                     // canBinary.nodeClass
        canGT,                          // canBinary.canOp
        sizeof(Nodes.BinaryNode),       // canBinary.iOperand1
        sizeof(Nodes.BinaryNode) + sizeof(Nodes.FieldNode),
                                        // canBinary.iOperand2
                                        //   Offsets in the Nodes array
    },
    {
        // Offset sizeof(Nodes.BinaryNode)
        nodeFIELD,                      // canField.nodeClass
        canFIELD,                       // canField.canOp
        1,                              // canField.iFieldNum
        0,                              // canField.iNameOffset: szField is
the
                                        //   literal at offset 0
    },
    {
        // Offset sizeof(Nodes.BinaryNode) + sizeof(Nodes.FieldNode)
        nodeCONST,                      // canConst.nodeClass
        canCONST,                       // canConst.canOp
        fldFLOAT,                       // canConst.iType
        sizeof(fConst),                 // canConst.iSize
        8,                              // canConst.iOffset: fconst is the
                                        // literal at offset
strlen(szField) + 1
    }};
static const char szTblName[] = "cust";     // Name of the table
static const char szTblType[] = szDBASE;    // Type of table
static const char szField[]   = "CUST_NO";  // Name of the field for the
                                            //   third node of the tree.
static const DFLOAT fConst    = 1500.0;     // Value of the constant for
                                            //   the second node of the
tree.
```

{button ,AL(`expressiontree')} Expression tree topics

Expression tree header

The expression tree header defines:

- the version tag of the expression
- the size of the tree structure
- the number of nodes in the node area
- the offset locations of the first node and the beginning of the literal pool.

The header is in this form:

```
#define CANEXPRVERSION 2
typedef struct{
    UINT16 iVer;
    UINIT16iTotalSize;
    UINT16 iNodes;
    UINT16 iNodeStart;
    UINT16 iLiteralStart;
  } CANExpr;
  typedef CANExpr far *pCANExpr;
  typedef pCANExpr far *ppCANExpr;
```

{button ,AL(`expressiontree')} Expression tree topics

Expression tree node area

Each node forms a branch of the tree and defines a condition. Nodes can define either operators or operands.

Operand nodes store the offset of field names or constants within the Literal Pool Area. The values are stored in the literal pool. A field node points to the offset location of a field name containing a literal, that is, the actual character string of the field name, which must be zero-terminated. A constant node points to a constant value within the literal pool.

Operator nodes are of different types:

- Relational
- Logical
- Arithmetic
- Miscellaneous

{button ,AL(`expressiontree')} Expression tree topics

Operator nodes, relational

Enumerated type	Description
canISBLANK	Unary; blank operand
canNOTBLANK	Unary; non-blank operand
canEQ	Binary; equal to
canNE	Binary; not equal to
canGT	Binary; greater than
canLT	Binary; less than
canGE	Binary; greater than or equal to
canLE	Binary; less than or equal to

Operator nodes, logical

Enumerated type	Description
canNOT	Unary; NOT
canAND	Binary; AND
canOR	Binary; OR

Operator nodes, arithmetic

Enumerated type	Description	SQL support
canMINUS	Unary; minus	Not supported by all SQL drivers
canADD	Binary; addition	Not supported by all SQL drivers
canSUB	Binary; subtraction	Not supported by all SQL drivers
canMUL	Binary; multiplication	Not supported by all SQL drivers
canDIV	Binary; division	Not supported by all SQL drivers
canMOD	Binary; modulo division	Not supported by all SQL drivers
canREM	Binary; remainder of division	Not supported by all SQL drivers

Operator nodes, miscellaneous

Enumerated type	Description
canCONTINUE	Unary; stops evaluating records when operand evaluates to false (provides a stop at the high range of the filter value)

Operator nodes point to the offsets of their operand nodes. See the sample expression tree in Literal Pool Area where binary operands cause the tree to branch.

Literal pool area

The literal pool is used to store the field names pointed to by each field node and the constant values pointed to by each constant node. Field names contain literals. Constant values must be represented in BDE logical types only.

For example, the following Boolean condition is represented as an expression tree parameter, and then as a chart:

```
CUST_NO <= 1500 AND CUST_NO >= 1300
```

Expression Tree

The following example assumes that the compiler allocates consecutively declared variables in physically contiguous memory:

```
  static const char szTblName[] = "cust";     // Name of the table
  static const char szTblType[] = szDBASE;    // Type of table
  static const char szField[]   = "CUST_NO";  // Name for the first field
  node
  static const char szField2[]  = "CUST_NO";  // Name for the second field
  node
  static const DFLOAT fConst    = 1500.0;     // Value of the first constant
  node
  static const DFLOAT fConst2   = 1300.0;     // Value of the second constant
  node


void
  Filter (void)
  {
      hDBIDb          hDb = 0;            // Handle to the database.
      hDBICur         hCur = 0;           // Handle to the table.
      DBIResult       rslt;               // Return value from IDAPI
  functions.
      pBYTE           pcanExpr;           // Structure containing filter
  info.
      hDBIFilter      hFilter;            // Filter handle.
      UINT16          uSizeNodes;         // Size of the nodes in the tree.
      UINT16          uSizeCanExpr;       // Size of the header information.
      UINT32          uSizeLiterals;      // Size of the literals.
      UINT32          uTotalSize;         // Total size of the filter
  expression.
      UINT32          uNumRecs = 10;      // Number of records to display.
      CANExpr         canExp;             // Contains the header information.
      struct {
          CANBinary MainNode;
          CANBinary BinaryNode1;
          CANField  FieldNode1;
          CANConst  ConstantNode1;
          CANBinary BinaryNode2;
          CANField  FieldNode2;
          CANConst  ConstantNode2;
      }
      Nodes = {                           // Nodes of the filter tree.
      {
          // Offset 0
          nodeBINARY,                     // canBinary.nodeClass
          canAND,                         // canBinary.canOp
```

```
        sizeof(Nodes.MainNode),         // canBinary.iOperand1
        sizeof(Nodes.MainNode)
          + sizeof(Nodes.BinaryNode1)
          + sizeof(Nodes.FieldNode1)
          + sizeof(Nodes.ConstantNode1),// canBinary.iOperand2
                                        //   Offsets in the Nodes array
    },
    {
        // Offset sizeof(Nodes.MainNode)
        nodeBINARY,                     // canBinary.nodeClass
        canLE,                          // canBinary.canOp
        sizeof(Nodes.MainNode)
          + sizeof(Nodes.BinaryNode1),  // canBinary.iOperand1
        sizeof(Nodes.MainNode)
          + sizeof(Nodes.BinaryNode1)
          + sizeof(Nodes.FieldNode1),   // canBinary.iOperand2
                                        //   Offsets in the Nodes array
    },
    {
        // Offset sizeof(Nodes.MainNode) + sizeof(Nodes.BinaryNode1)
        nodeFIELD,                      // canField.nodeClass
        canFIELD,                       // canField.canOp
        1,                              // canField.iFieldNum
        0 ,                             // canField.iNameOffset: szField is
the
                                        // literal at 0 (start of literal
pool)
    },
    {
        // Offset sizeof(Nodes.MainNode) + sizeof(Nodes.BinaryNode1)
        //   + sizeof(Nodes.FieldNode1)
        nodeCONST,                      // canConst.nodeClass
        canCONST,                       // canConst.canOp
        fldFLOAT,                       // canConst.iType
        sizeof(fConst),                 // canConst.iSize
        sizeof(szField),                // canConst.iOffset: fConst is the
                                        // literal at offset
sizeof(szField)
    },
    {
        // Offset sizeof(Nodes.MainNode) + sizeof(Nodes.BinaryNode1)
        //   + sizeof(Nodes.FieldNode1) + sizeof(Nodes.ConstantNode1)
        nodeBINARY,                     // canBinary.nodeClass
        canGE,                          // canBinary.canOp
        sizeof(Nodes.MainNode)
          + sizeof(Nodes.BinaryNode1)
          + sizeof(Nodes.FieldNode1)
          + sizeof(Nodes.ConstantNode1)
          + sizeof(Nodes.BinaryNode2),  // canBinary.iOperand1
        sizeof(Nodes.MainNode)
          + sizeof(Nodes.BinaryNode1)
          + sizeof(Nodes.FieldNode1)
          + sizeof(Nodes.ConstantNode1)
          + sizeof(Nodes.BinaryNode2)
          + sizeof(Nodes.FieldNode2),   // canBinary.iOperand2
                                        //   Offsets in the Nodes array
    },
```

```
    {
        // Offset sizeof(Nodes.MainNode) + sizeof(Nodes.BinaryNode1)
        //   + sizeof(Nodes.FieldNode1) + sizeof(Nodes.ConstantNode1)
        //   + sizeof(Nodes.BinaryNode2)
        nodeFIELD,                      // canField.nodeClass
        canFIELD,                       // canField.canOp
        2,                              // canField.iFieldNum
        sizeof(szField)+sizeof(fConst), // canField.iNameOffset: szField2
is
                                        //   the literal at sizeof(fConst)
                                        //   + size of the first field
    },
    {
        // Offset sizeof(Nodes.MainNode) + sizeof(Nodes.BinaryNode1)
        //   + sizeof(Nodes.FieldNode1) + sizeof(Nodes.FieldNode1)
        //   + sizeof(Nodes.BinaryNode2) + sizeof(Nodes.FieldNode2)
        nodeCONST,                      // canConst.nodeClass
        canCONST,                       // canConst.canOp
        fldFLOAT,                       // canConst.iType
        sizeof(fConst2),                // canConst.iSize
        sizeof(szField)
          + sizeof(fConst)
          + sizeof(szField2),           // canConst.iOffset: fconst is the
                                        // literal at sizeof(fConst)+size
of
                                        //   the first field + second field
    }};
```

Chart

The chart below represents the same Boolean expression: CUST_NO <= 1500 AND CUST_NO >= 1300
(Note that the offsets are shown in parentheses.)

Header:	----------			
Binary node:		AND (0)		
Binary nodes:		LE (12)	GE (50)	
Constant & field nodes:	FIELD (24)	CONST (36)	FIELD (62)	CONST (74)
Literal / constant pool:	CUST_NO (0)	1500 (8)	CUST_NO (16)	1300 (24)

{button ,AL(`expressiontree')} Expression tree topics

Database driver characteristics

The Borland Database Engine (BDE) requires a separate driver to support each database format or data source. Standard database drivers for dBASE, FoxPro, Access, Paradox, and Text databases are included with BDE. To extend BDE to support additional SQL database systems, you must install the appropriate Borland SQL Links driver.

These sections provide additional information about specific driver types that you may use.

- SQL Drivers
- Paradox Driver
- Access Driver
- FoxPro Driver
- Text Driver

SQL drivers

All BDE drivers for SQL servers share common services including record navigation, record caching, record editing, and server query management. Only about twenty percent of the services are driver specific, addressing driver capabilities, data types and data translations, transaction control, server specific query creation and server calls.

All SQL drivers are fully described in SQL Links Guide

Click here for topics on using BDE with SQL drivers:

{button ,AL(`sqldrivers')} SQL driver topics

Informix driver

This topic discusses features unique to the Informix SQL Link driver.

Stored procedure support

The Informix driver supports stored procedures. Please note the following points:

1. Informix stored procedures have input parameters but no output parameters.
2. DbiOpenSPParamList returns all input parameters and sets *SPParamDesc.uParamNum* and *SPParamDesc.szName* starting from 1.

   SPParamDesc[0].uParamNum = 1

   SPParamDesc[0].szName = "1"

   and SPParamDesc[0].eParamType = paramIN;

   .

   .

   .

   The rest of the information for SPParamDescs (such as *uFldType*, *usubType*, *iUnits1*, *iUnits2*, *uOffset*, *uLen*, and *uNullOffset*) must be set by user.

Retrieving SQLCA information

The Informix SQL Link driver includes an improved passthrough property that contains native SQLCA information. Users can use the drvNATIVESQLCA property with DbiGetProp to retrieve SQLCA information. SQLCA information gives detailed data on Informix server errors and exceptions. When an Informix error occurs, the Informix Global SQLCA information for that error is retrieved and retained by the SQL driver until the next time the database server is accessed. General SQLCA information is returned whenever an error hasn't occurred.

The following table shows the information that is made available.

*ppropValue	*pilen
SQLCA	sizeof(struct sqlca_s)

An example:

```
// Informix SQLCA structure from Informix sqlca.h header file
struct sqlca_s
     {
     long sqlcode;
     char sqlerrm[72]; /* error message parameters */
     char sqlerrp[8];
     long sqlerrd[6];
               /* 0 - estimated number of rows returned */
               /* 1 - serial value after insert or  ISAM error code */
               /* 2 - number of rows processed */
               /* 3 - estimated cost */
               /* 4 - offset of the error into the SQL statement */
               /* 5 - rowid after insert  */
     struct sqlcaw_s
     {
     char sqlwarn0; /* = W if any of sqlwarn[1-7] = W */
     char sqlwarn1; /* = W if any truncation occurred or
                       database has transactions */
     char sqlwarn2; /* = W if a null value returned or
                       ANSI database */
     char sqlwarn3; /* = W if no. in select list != no. in into list or
                       turbo backend */
```

```
     char sqlwarn4; /* = W if no where clause on prepared update,
                       delete or incompatible float format */
     char sqlwarn5; /* = W if non-ANSI statement */
     char sqlwarn6; /* reserved */
     char sqlwarn7; /* reserved */
     } sqlwarn;
     };

struct sqlca_s mySqlca;

int main()
{
     // Initialize engine

     // Connect to database.

      //get the sqlca (on no exception)
      unsigned int len;
      DbiGetProp(hDb,drvNATIVESQLCA, &mySqlca, sizeof(sqlca_s),
                      &len);

     //get the sqlca (on an exception)
      DbiOpenTable(hDb, "non existing table", ...)

      DbiGetProp(hDb,drvNATIVESQLCA, &mySqlca, sizeof(sqlca_s),
                      &len);
       return 0;
}
```

Retrieving Informix database information

The Informix SQL Link driver has three properties that you can access using DbiGetProp to determine the type of database the BDE is connected to

Property	Type	Description
dbONLINE	BOOL	TRUE if the database connected is ONLINE type otherwise FALSE
dbTRANALLOWED	BOOL	TRUE if the database connected permits transactions otherwise FALSE
dbANSI	BOOL	TRUE if the database connected is ANSI type otherwise FALSE

{button ,AL(`sqldrivers')} SQL driver topics

DB2 driver

This topic discusses features unique to the DB2 SQL Link driver.

DBCLOB data type

The DB2 physical type DBCLOB is not currently supported within the BDE even though schema information indicates that it is an available type. It is suggested that you use either CLOB or BLOB physical types.

Stored procedures

The SQL Links driver for DB2 translates the logical BDE string type (fldZSTRING) to a physical DB2 SQL_CHAR data type when it is passed as a parameter to a stored procedure. This can cause problems for stored procedures that are hard-coded to expect a physical SQL_VARCHAR as a parameter.

Programmers should write stored procedures that examine the sqltype member of the SQLDA structure (for example, input_sqlda->sqlvar[i].sqltype) to determine which data type the client has actually bound, instead of expecting a specific data type.

Creating indexes on AS/400 servers

Due to a problem with the IBM server software version 2.1.1, if you create a new index by specifying an index name that includes lowercase characters and is enclosed in double quotes ("<index_name>"), the index cannot be accessed by the BDE.

For example, the following statement creates an accessible index,

```
CREATE INDEX CustNdx ON ....
```

but the following statement creates an index that the BDE can't access,

```
CREATE INDEX "CustNdx" ON ....
```

If creating an index through a call to DbiAddIndex, the *szName* member in the IDXDesc Parameter should only be uppercase.

Creating BLOB or CLOB columns

When using DbiCreateTable, BLOB or CLOB columns created in a DB2 table by the function are set by default to a size of 1MB. To create BLOB columns of different sizes, use Passthrough SQL.

VARBINARY output parameters in stored procedures

In a stored procedure parameter description (*SPParamDesc.uFldType*), normally a corresponding BDE logical data type is specified. There is one exception. A varbinary output parameter should specify fldBYTES instead of fldVARBYTES.

Calling by name in stored procedures

DB2 only supports calling by number in stored procedures. Always use calling by number instead of calling by name.

{button ,AL(`sqldrivers')} SQL driver topics

Sybase CT-Lib driver

This topic discusses features unique to the Sybase CT-Lib SQL Link driver.

DBIERR_MULTIPLEUNIQRECS on dead tables

If the error DBIERR_MULTIPLEUNIQRECS occurs while using a dead table (no index at all), then the cursor for that table should be closed and reopened, regardless of whether DbiBeginTran had been called to start a transaction.

Multiple active stored procedure support

The Sybase CT-Lib SQL Link driver can now perform row fetches from multiple stored procedures simultaneously using the new property stmtEXECASCURSOR.

The stmtEXECASCURSOR property allows users to ask for a CT-Lib cursor instead of a CT-Lib command. CT-Lib cursors let the user have multiple cursors open and fetch rows from those simultaneously. With CT-Lib commands, all pending results must be processed before executing the next operation. This property is mainly for users who execute Sybase stored procedures that return a result set. They can pass the SQL string as "EXECUTE proc_name" or "EXECUTE proc_name :1, :2, :3 ..." (if there are parameters), prepare the statement, then set stmtEXECASCURSOR to True. This makes the driver open cursors on the stored procedure instead of a command.

Some limitations to this property:

1. The stored procedure should not have any BDE output parameters or return status.
2. Input parameters must be place holders in the SQL string and must be bound before execution. (Literal substitution of parameter values in the string doesn't work.)
3. The stored procedure body should contain a single SELECT statement returning a single result set.

An example:

```
int main ()
{
    // Initialize engine

    // Connect to database

    //Prepare a statement
    DbiQPrepare(hDb, .., .. , phStmt);

   // Set the property
   DbiSetProp(hStmt,  stmtEXECASCURSOR, TRUE);

   // Bind parameters if any

   //DbiQExec(hStmt, phCur);

     .
     .
     .

     return 0;
}
```

{button ,AL(`sqldrivers')} SQL driver topics

Passthrough SQL

The native SQL dialect of the SQL server can be passed directly to the server, as long as the appropriate BDE driver is installed. These passthrough SQL queries can be executed directly by using DbiQExecDirect or in stages. See Querying Databases

The SQLPASSTHRU MODE parameter of the BDE configuration file allows you to specify whether passthrough and non-passthrough SQL operations can share the same connection. It also allows you to specify whether you want passthrough SQL to be autocommitted or not (if the connection is shared). When passthrough and non-passthrough SQL operations share the same connection, transaction control statements should not be executed in passthrough SQL. Instead, use DbiBeginTran and DbiEndTran.

Update of Simple Unidirectional SQL Passthrough Queries

Certain SQL servers support these dynamic SQL statements:

```
UPDATE ... WHERE CURRENT of CursorName
DELETE ... WHERE CURRENT of CursorName
```

BDE supports this syntax, provided that it is also supported by the server.

Use the statement property stmtCURSORNAME (defined in the header file IDAPI.H) to set or get the cursor name from the passthrough SELECT statement and use it in the UPDATE statement. For example:

```
 ...
DbiQPrepare(hDb,
            qrylangSQL,
            "SELECT * FROM FOO FOR UPDATE OF f1",
            &hStmt);

// set the cursor name for the SELECT statement
DbiSetProp(hStmt,
           stmtCURSORNAME,
           pszCursorName);

// set unidirectional cursor
DbiSetProp(hStmt,
           stmtUNIDIRECTIONAL,
           TRUE);

// execute the SELECT stmt
DbiQExec(hStmt,
         &hCur);

// fetch a record
DbiGetNextRecord(hCur,
                 dbiNOLOCK,
                 pRecBuf,
                 NULL);

// Note that we use DbiQExecDirect to execute the UPDATE
// statement in this example.
// DbiQPrepare/DbiQExec/DbiQFree can be used instead of
// DbiQExecDirect to execute the UPDATE

sprintf(pszQuery,
        "UPDATE foo SET f1 = 'X' WHERE CURRENT of %s",
        pszCursorName);
```

```
// update the current record
DbiQExecDirect(hDb,
               qrylangSQL,
               pszQuery,
               NULL);

// free the SELECT stmt
DbiQFree(&hStmt);

// close the SELECT cursor
DbiCloseCursor(&hCur);
...
```

Certain drivers require that you set the cursor name BEFORE the SELECT statement is executed (as in the above example). Other drivers do not require you to explicitly set the cursor name and will generate one for you. If the server generates a cursor name, you can retrieve that name by calling DbiGetProp AFTER the SELECT statement has been executed. As always, when using passthrough SQL, you must know the native syntax supported by the back end server.

Where not supported, the function DbiSetProp with stmtCURSORNAME will return DBIERR_NOTSUPPORTED.

InterBase

By default the InterBase SQL Link driver must close cursors when transactions end (COMMIT/ABORT occurs). When this happens, the remaining rows are read from the server and cached locally. This means that a COMMIT/ABORT can cause you to lose your current cursor position, and a subsequent UPDATE ... WHERE CURRENT can update the WRONG row. For this reason, you must be certain that a COMMIT/ABORT does not cause SQL Link to prematurely close the server cursor.

There are two ways to guarantee this:

1 Set your SQLPASSTHRU MODE to NOT SHARED. In this mode, all passthrough statements are performed on a separate connection and will NOT be autocommitted.

2 If your SQLPASSTHRU MODE is either SHARED AUTOCOMMIT or SHARED NOAUTOCOMMIT, passthrough and non-passthrough statements share the same connection. Operations performed within an explicit transaction (that is, within the DbiBeginTran/DbiEndTran block) are never autocommitted.

By adding 4096 to the setting of DRIVER FLAGS in the BDE configuration, you can specify that the InterBase SQL Links driver should use soft commits. Soft commits are a feature of InterBase that let the driver retain the cursor when commiting changes. Soft commits improve performance on updates to large sets of data. When not used, the BDE must re-fetch all the records, even for a single record change. With soft commit the cursor is retained, and a re-fetch isn't needed. Soft commits are never used in explicit transactions started by BDE client applications.

DRIVER FLAGS	Isolation level and commit type
0	Read committed, hard commit
512	Repeatable read, hard commit
4096	Read committed, soft commit
4068	Repeatable read, soft commit

{button ,AL(`sqldrivers')} SQL driver topics

SQL transaction control

To control explicit transactions, use DbiBeginTran and DbiEndTran. Except for explicit transactions, the BDE isolation level is Read Committed, with auto-committed modifications. Some SQL drivers support only the server default isolation level inside of an explicit transaction. To verify the actual isolation level used, call DbiGetTranInfo after a successful call to DbiBeginTran.

Example 1: No explicit transaction

The SQL driver automatically starts a server transaction if necessary:

```
DbiGetNextRecord (hCursor, dbiWRITELOCK, &myRecBuff, NULL);
```

The application changes the record buffer data:

```
DbiModifyRecord (hCursor, &myRecBuff, TRUE);
```

If the record modification succeeds, it is automatically committed to the database.

Example 2: Explicit transaction used

The application uses a transaction:

```
DbiBeginTran (hDb, xilREADCOMMITTED, NULL);
```

The SQL driver starts a server transaction:

```
DbiGetNextRecord (hCursor, dbiWRITELOCK, &myRecBuff, NULL);
```

The application changes the record buffer data:

```
DbiModifyRecord (hCursor, &myRecBuff, TRUE);
```

The application can make more changes in the transaction:

```
DbiEndTran (hDb, NULL, xendCOMMIT);
```

The SQL driver commits the server transaction.

InterBase: By default, when the InterBase SQL Link driver's SQLPASSTHRUMODE is set to SHAREDAUTOCOMMIT, the BDE uses the InterBase API call isc_commit_transaction to commit transactions, which close all cursors. When using implicit transactions, running a new query makes the BDE pre-fetch all records of the previous query, if it is still open. When using an explicit transaction, a pre-fetch of all open queries will occur when the transaction is committed. If the result sets of the opened queries are large, then slower performance can occur.

When using Interbase 4 with the InterBase SQL Links driver, you can improve performance for implicit transactions by setting DRIVER FLAGS to 4096. This makes the BDE use isc_commit_retaining to keep cursors open; this avoids having to pre-fetch records when a commit occurs. Setting DRIVER FLAGS to 4096 does not effect explicit transaction behavior; when an explicit commit occurs, any open queries on that connection will be pre-fetched.

NOTE: The driver places interest locks on any relation touched during a transaction, and interest locks are maintained across isc_commit_retaining calls. Therefore any DDL-related operations for the locked relations are blocked for all users, including your session, when DRIVER FLAGS is 4096. To avoid this, perform periodic "hard" commits by using DbiBeginTran and DbiEndTran to start and commit an explicit transaction.

Transaction isolation levels

Extended transaction isolation levels are supported. If an unsupported isolation level is specified in DbiBeginTran, the next-highest supported isolation level is used. If the requested isolation level is higher than any supported isolation level, then an error is returned (DBIERR_NOTSUPPORTED). The highest level (most isolated) level is Repeatable Read, then Read Committed, and finally Dirty Read. As always, you can verify the actual isolation level that was used by calling DbiGetTranInfo.

This database property is used with DbiGetProp to retrieve the server's default transaction

isolation level:

```
dbDEFAULTTXNISO, ro eXILType     Server's default transaction isolation
level
```

Compatibility

InterBase

Supports Repeatable Read and Read Committed. The wait mode is set to NO WAIT.

Sybase

Supports only the server default, Read Committed.

Oracle

Supports Read Committed and Repeatable Read. However, a Repeatable Read transaction is always READ ONLY.

Informix

In some cases, when connecting with your Informix database, your BDE application overrides the current Informix transaction isolation settings. The following table shows under which circumstances these overrides occur.

DB2

Supports all BDE transaction isolation levels. Any DB2 isolation levels not supported by the BDE are converted to Read Committed.

Database	**Default isolation level: Informix**	**Default isolation level: SQL Link**
ANSI	RepeatableRead	CommittedRead
Logged	CommittedRead	CommittedRead
Non-logged	DirtyRead	DirtyRead

The following table shows the changes from previous versions of BDE.

REQUESTED ACTUAL ISOLATION LEVEL USED

Isolation Level	**pre-BDE 2.5**	**BDE 2.5**
Sybase:		
DirtyRead	ReadCommited	ReadCommitted
ReadCommitted	ReadCommitted	ReadCommitted
RepeatableRead	ReadCommitted	DBIERR_NOTSUPPORTED
Oracle:		
DirtyRead	ReadCommitted	ReadCommitted
ReadCommitted	ReadCommitted	ReadCommitted
RepeatableRead	ReadCommitted	RepeatableRead (READ ONLY)
InterBase:		
DirtyRead	RepeatableRead	ReadCommitted
ReadCommitted	RepeatableRead	ReadCommitted
RepeatableRead	RepeatableRead	RepeatableRead

You can maintain compatibility with pre-BDE 2.5 behavior by setting the DRIVER FLAGS parameter in the BDE configuration file. All SQL drivers have a field called DRIVER FLAGS in the DRIVER INIT section. To obtain pre-BDE 2.5 transaction behavior, set the bit corresponding to 0x0200 (512 decimal).

{button ,AL(`sqldrivers')} SQL driver topics

SQL connection

BDE connects to the SQL server database by using the following guidelines:

- BDE uses the server authorization scheme. The password is used in DbiOpenDatabase to connect to the server.
- Most BDE features require an open database, with the exception of retrieving driver capabilities, such as data-types information.
- Transactions and passthrough operations are done in the database context.

SQL record caching

Two caching mechanisms are used:

- Live Caching
  Done for a cursor, if possible.
- Dead Caching
  Used if live caching cannot be done.

{button ,AL(`sqldrivers')} SQL driver topics

Live caching

Live caching provides fuller BDE support than dead caching. It can be fast or slow, depending on other factors. Live caching is used by default if an index or row ID exists, but only for tables, not queries. With DbiOpenTable, iIndexId can be set to NODEFAULTINDEX to force dead caching even though an index or row ID exists.

The following general rules apply to live caching:

- Data tends to be fresh. The fastest index is chosen automatically if none is specified during table open.
- A partial cache is kept, ordered by index. The cache contains the current cursor row, plus the last several rows fetched.
- Live caching allows cache refresh. Refresh can be done manually via DbiForceReread and is done automatically if the cursor moves around.
- Live caching allows key-oriented operations, such as DbiSetRange and DbiSetToKey.

Record caching example: live

A Customer table with unique or non-unique index on ID field.

ID	Name
10000	John
11001	Mary
12321	Harry
12345	Beth
12666	Joe

The SQL driver finds some basic information about the table structure, but no data is retrieved:

```
DbiOpenTable (
  hDb,
  "Customer",
  NULL,
  "IdIndex",
  ...,
  &hCursor ...);
```

The SQL driver sets up for data retrieval:

```
UINT16 myKey = 12321;
DbiPutField (hCursor,
   1,
  &myRecBuff,
  &myKey);
DbiSetToKey (hCursor,
  keySEARCHGEQ,
  FALSE,
  1,
  0,
  &myRecBuff);
```

The SQL driver query:

```
SELECT Id, Name
FROM Customer
WHERE Id >= 12321
ORDER BY Id
DbiGetNextRecord (...)
```

The SQL driver caches a row:

ID	Name
12321	Harry

```
DbiGetNextRecord (...)
DbiGetNextRecord (...)
```

The SQL driver caches more rows:

ID	Name
12321	Harry
12345	Beth
12666	Joe

```
DbiGetPriorRecord (...)
```

The SQL driver uses a cache, rather than the server:

```
DbiSetToBegin (...)
```

The SQL driver terminates the query and clears the cache:

ID	Name

No data in the cache

Dead caching

Dead caching may be used when live caching is not possible. With dead caching, the data may not be fresh. The following rules apply to dead caching:

- Dead caching is used
  - for passthrough queries; or
  - if no ordering exists; and
  - provided that the statement property stmtLIVENESS is not set to wantLIVE
- Dead caching is used for DbiOpenTable if no index is available and the server does not support row IDs, or if iIndexId is set to NODEFAULTINDEX with DbiOpenTable.
- Dead caching keeps a full client snapshot cache. As records are read from the server, they are stored locally in case they are needed again.
- Dead caching provides no cache refresh. You must close and re-open the table, or re-execute a query to see new data.
- Since there is no key, key operations (such as DbiSetRange and DbiSetToKey) are not supported. Other navigation functions such as DbiSetToBookMark are supported.

Record caching example: dead

The SQL driver finds some basic information about the Customer table structure, but no data is retrieved:

ID	Name
11001	Mary
10000	John
12666	Joe
12321	Harry
12345	Beth

```
DbiOpenTable (
  hDb,
  "Customer",
  NULL,
  NULL,
  ...,
  &hCursor ...);
DbiGetNextRecord (...)
```

The SQL driver executes a query:

```
SELECT Id, Name
FROM Customer
```

The SQL driver caches a row:

ID	Name
11001	Mary

```
DbiGetNextRecord (...)
```

The SQL driver caches another row:

```
DbiGetPriorRecord (...)
```

The SQL driver uses a cache:

```
DbiSetToBegin (...)
```

The SQL driver leaves the query and cache alone:

ID	Name
11001	Mary
10000	John
12666	Joe

iIndexId

iIndexId Type: UINT16 (Input)

Specifies the index identifier, which is the number of the index to be used. The range for the index identifier is 1 to 511. Used for Paradox tables only and is ignored if pszIndexName is specified.

SQL record modification requirements

The following requirements must be met to modify a record:

- The server must allow each operation. Security and capability are important: server views may not allow changes, and different types of modification are authorized separately.
- Views support insert, modify, and delete if allowed by the server. Queries do not support modifications.
- Record modifications performed within an explicit client transaction may require that a unique index or server ROWID exists on the table. For example, both DbiSetRange and DbiGetRecordForKey require a current index. However, BDE supports the ability of SQL data sources to order records by any field without using an index on the server. A current index (for SQL data sources) can be defined as any group of fields from a specific table, whether or not a corresponding index exists on the server. BDE creates a pseudo-index by using one or more user-specified SQL fields to define the requested order.

  For information on implementing pseudo-indexes, see DbiOpenTable or DbiSwitchToIndex.

{button ,AL(`sqldrivers')} SQL driver topics

SQL record modification behavior

The following characteristics describe record modification behavior:

- All current record modifications use optimistic locking. An optimistic lock must be explicitly requested, but the lock request does not attempt to explicitly lock the record on the server.
- Except for an explicit client transaction, all modifications are singleton operations. This means that upon successful completion, each modification is autocommitted.
- Transaction or batch request overrides singleton behavior.

Record Modification Example

The SQL driver saves a copy of the record as an optimistic lock. The application changes the record buffer data:

```
DbiGetNextRecord (hCursor, dbiWRITELOCK, &myRecBuff, NULL);
```

The SQL driver uses the saved record copy to find and modify the data:

```
DbiModifyRecord (hCursor, &myRecBuff, TRUE);
UPDATE Customer
SET Name = "Harold"
WHERE Id = 12321 AND Name = "Harry"
```

Then the SQL driver verifies the resulting rows changed:

- If one row changed, optimism has paid off.
- If no rows changed, the optimistic lock was broken.
- If more than one row changed, there was no unique index and the optimistic lock was broken.

{button ,AL(`sqldrivers')} SQL driver topics

SQL record-locking behavior

SQL servers automatically and transparently lock data as required, although different SQL servers vary in the type of lock used, and how granular the lock is. For example, some servers provide individual record locks, while others can only lock a group, or page, of records. Also, some servers provide automatic record versioning or database snapshots so that other copies of data being modified can be read by clients instead of waiting for a modification to finish.

In addition to the automatic locking that SQL servers provide, SQL drivers provide a particular type of record locking called optimistic locking. Optimistic locking allows a client to make changes to a local copy of the record without the performance and concurrency penalty incurred by asking the server for a lock over the modification duration. When the client modifications are finished, the current SQL server record is first checked to make sure no changes have occurred to the record, then the modifications are completed. The operation is said to be optimistic because it assumes that no other client will change the record, but then makes sure of that as the final change is sent to the SQL server.

If the record was changed, an optimistic lock failure occurs. The client is notified that the requested operation cannot be performed because someone else has changed the data. The client can then inspect the new data and decide whether or not to make changes at that time.

Because server data cached on the client can immediately become out of date at the server, SQL drivers always perform optimistic locking. This protects the client against inadvertently changing data that has never been inspected.

Keyed Updates

Keyed updates give you more control over optimistic record locking for improved performance. You can control which columns are placed in the WHERE clause of an UPDATE or DELETE statement generated by calls to DbiModifyRecord or DbiDeleteRecord.

You can set and retrieve the SQL-specific cursor property curUPDLOCKMODE by using DbiGetProp and DbiSetProp. This property is valid for all SQL Link drivers and the ODBC Socket.

The following enumeration defines the options:

```
typedef enum
   {
    updWHEREALL,
    updWHEREKEYCHG,
    updWHEREKEY
   } UPDLockMode;
```

updWHEREALL
All fields (except BLOBs) are placed in the WHERE clause of the update or delete statement for DbiModifyRecord or DbiDeleteRecord. This is the default when a cursor is returned. The behavior is identical to current "optimistic record locking" behavior.

updWHEREKEY
If a unique index exists, only those fields in the key are placed in the WHERE clause of the update or delete statement for DbiModifyRecord and DbiDeleteRecord. The key that is used is based on the active index. If the active index is a unique index, then it will be used. Otherwise the driver will pick the "best" unique index. (Note: For Oracle, it will pick the special column, ROWID). If there is no unique index, then all fields are placed in the WHERE clause and the behavior is identical to updWHEREALL.

updWHEREKEYCHG
Similar to updWHEREKEY except that changed fields (as well as indexed fields) are placed in the WHERE clause.

WARNING: When using updWHEREKEY or updWHEREKEYCHG, it is possible to overwrite other users' updates. Therefore you should use this feature only when you **know** that overwrites will not be a problem.

{button ,AL(`sqldrivers')} SQL driver topics

SQL table-locking behavior

The SQL driver provides a degree of support for table locking if the SQL server supports it. Different SQL servers provide different levels of support for table locking. Some servers provide no table locking support at all. Others only provide support for read-only locking (many clients can share a lock and all can read). Some SQL servers provide support for locking, but require the client to wait until a lock is granted, rather than letting the client know immediately if the lock could not be achieved. For information on locking support provided by your SQL server, see your server documentation.

SQL servers that support table locks maintain a lock within the context of a transaction: a lock can only be acquired within a transaction, and only released by terminating the transaction. This is sometimes referred to as a two-phase locking protocol. When the SQL driver is asked to acquire a table lock, it automatically starts a transaction if necessary. When asked to release a table lock, the SQL driver must commit the transaction in order to release the lock. Because a transaction commit releases all locks, the SQL driver automatically re-acquires any remaining locks.

Note: If a table lock is held when a commit becomes necessary, a time window exists in which the lock is not held and unanticipated changes can occur. For this reason, it is recommended that all table locks be released together when the last lock is needed, or that explicit SQL transactions be used instead of table locking.

{button ,AL(`sqldrivers')} SQL driver topics

SQL asynchronous queries

SQL Links can cancel long-running queries if the server supports asynchronous query submission. Verify that your SQL Link driver currently supports asynchronous query execution on Windows.

Use the dbASYNCSUPPORT database property with DbiGetProp to inquire whether a driver supports asynchronous queries:

```
dbASYNCSUPPORT , ro BOOL   Does the driver support
                                asynchronous query execution?
```

There are two options to asynchronous query submission/cancel:

1) The query cancels because it exceeds the maximum time allowed.

2) The query completes normally.

The parameter MAX QUERY TIME in the BDE configuration file (IDAPI.CFG) is a DB OPEN parameter. It is available for SQL Links drivers that support this feature, such as those for Sybase and Microsoft SQL Server.

Note: By default, SQL statements sent to MS SQL and Sybase servers using DB-Lib are now sent using the synchronous query submission API. Earlier versions of SQL Links used the asynchronous query submission API. To use the asynchronous query submission API, add 2048 to the current value of DRIVER FLAGS or set it to 2048 if it's blank.

You can use the BDE Administrator to set MAX QUERY TIME for the maximum amount of time (seconds) you want to wait for a query to finish executing. (The default value is 3600 seconds, or one hour.) If this time limit is exceeded, the query is canceled. When a query is successfully canceled, DBIERR_CANCEXCEPT "Query canceled" is returned.

{button ,AL(`sqldrivers')} SQL driver topics

SQL performance tips

The following tips are suggested to help reduce unnecessary processing, and speed up performance:

- Use passthrough SQL for complex queries or stored procedures.
- You can bypass BDE functions and make direct calls using the native SQL API. Use DbiGetProp to get native handles.
- Use the server to minimize the size of the returned result set.
- Return results into a local table for processing.
- Use DbiAddFilter, DbiSetRange, and DbiSetFieldMap before data access to limit the number of records accessed.
- Create a descending index if backwards navigation is done frequently.
- Avoid moving toward the beginning of the table except within a small cache range.
- Avoid using DbiSetToEnd and DbiSetToKey in the middle of large tables or when the table is ordered on a composite index.

All options mentioned below are configurable using the BDE Administrator:

- For Microsoft SQL Server and Sybase: increase PACKET SIZE to at least 4096. You must also need to set the Packet Size option on the Microsoft of Sybase Server to match. Make sure that DRIVER FLAGS is 0. If it is 2048, queries will execute in asynchronous mode, which is slow.
- For Oracle, DB2, and the ODBC socket: try adjusting ROWSET SIZE. This specifies how many rows you fetch or insert in a single server operation.
- Set TRACE MODE to 0. This option is used only for debugging and can slow down your application.
- If your client/server applications TTables in Delphi, consider using TQueries along with cached updates to improve the performance of your overall application. TTables give you an easy model for programming and provide adequate performance but are not designed with speed in mind.
- Set SQLPASSTHRU MODE to SHARED NOAUTOCOMMIT and use explicit Begin Transaction and End Transaction statements in your application instead of relying on SQL Links to do AUTOCOMMIT.

See also:

Improving BDE performance
A few general suggestions for maximizing BDE's performance in accessing tables.

{button ,AL(`sqldrivers')} SQL driver topics

Paradox driver

The Paradox driver supports both descending and unique secondary index types and default values ("NOW") for the time stamp and time fields.

Descending and unique secondary index types

You can use composite indexes where some fields are ascending and others are descending.

The descending and unique secondary index types share these characteristics :

- Both index types require Table Level 7.
- At table creation time the table level is automatically upgraded to level 7 if:
  a) any of these index types are requested, *and*
  b) no lower table level is specified as an optional parameter (DBIERR_TABLELEVEL).
- If adding the index to a table with a lower table level, the error returned is DBIERR_TABLELEVEL.
- DBIERR_TABLELEVEL error code replaces DBIERR_OLDVERSION whenever a higher table level is required.
- Both index types share the characteristics of "Composite secondary indexes." They require a unique index name and so on.
- The Descending and Unique options can be mixed with each other, as well as with the *bMaintained*, and *bCaseInsensitive* options.
- In case of duplicate fields, the indexes must be distinguished by at least one of these options:

  bUnique

  bCaseInsensitive

  bDescending (not *bMaintained*)

Note: If two indexes are distinguished only by the bUnique option, they will in effect be equivalent, but is still accepted.

Descending secondary indexes

The IDXDesc structure includes this boolean array :

```
BOOL16          abDescending[DBIMAXFLDSINKEY];
```

The abDescending array indicates which fields in a composite secondary index are descending (TRUE). Thus, in a composite key, if abDescending[i] is TRUE--where 'i' is the index to aiKeyFlds[]--then the aiKeyFld[i] is descending.

This array is ignored if:

- The existing boolean *bDescending* is FALSE.
- The index has only one field.
  If you have an index with only one field, (and if and *bDescending* is TRUE), then you do not need to specify TRUE in *abDescending*[0]. In that case, bDescending counts for everything and *abDescending* is ignored. This rule is consistent with previous semantics (for dBASE and FoxPro descending indexes).

There are no other restrictions on how ascending and descending can be mixed. You may improve performance slightly if you do not specify a descending index.

Unique secondary indexes

The *bUnique* option enables the creation of unique secondary indexes.

- At table create time: upgrades the table to level 7.
- At add index time: requires the fields in the unique index to be unique already, otherwise it will fail. No error table is generated for the records that contain the non-unique field

values.

- At restructure time: If the fields are not unique, restructure will fail, and will *not* generate an error table for records containing non-unique field values.

Valchecks

Valchecks are default values for TIME and TIMESTAMP. Similar to specifying "TODAYVAL" for a default value for a date field. You can specify 'NOW' for the default value for a TIME and TIMESTAMP field.

- This has an effect only at DbiInitRecord time, where the 'current' time or timestamp will be substituted.
- This feature is supported only for table level 7 or higher.
- At create table or restructure table time: if the table level is less than 6, the function will fail with DBIERR_TABLELEVEL.
- The table will NOT be automatically upgraded.
- For table level 7 tables, up to 255 VCHKDesc's may be supplied at table create time. For lower table levels, the number is 64. If more are supplied, the function returns DBIERR_TABLELEVEL.

Record locks

The maximum number of record locks on a shared Paradox table is 255.

Access driver

If you have a version of the Microsoft JET engine (included with Microsoft Access and FoxPro) installed on your system you can use the BDE to open or create Microsoft Access tables using the MSACCESS driver. In the BDE API, use the constant szMSACCESS when creating Access tables or checking the table type. See Logical types and driver-specific physical types for a table listing the new physical data types for MSACCESS and their BDE logical equivalents:

Two drivers are available: driver IDDAO32.DLL for Access 95 and Jet Engine 3.0, and driver IDDA3532.DLL for Access 97 and Jet Engine 3.5. Use the BDE Administrator (BDEADMIN.EXE) to specify which Access driver to use. The default is IDDA3532.DLL.

FoxPro driver

The included FoxPro driver allows BDE clients to access FoxPro tables natively. Previous versions of the BDE used the native dBASE driver to access FoxPro data. When opening a table, the BDE detects if you are opening a FoxPro table and uses the appropriate driver.

Text driver

The text driver allows BDE clients to access text files. The text driver allows BDE clients to access text data directly without first importing into a table format. By using this driver, the application developer can build a more efficient import/export utility. Filters can be set on the cursors that are opened on the text files to import/export only those records that satisfy the filter's criteria.

When you open a text table, you can provide the field descriptor information by calling the function DbiSetFieldMap to set a field map or you can bind external schema to text tables:

- Field Maps
- Binding External Schema to Text Tables

Creating a text file with DbiCreateTable

A text file can be created by using DbiCreateTable. The developer supplies only table name and driver type values in the CRTblDesc descriptor; the rest of the field values are ignored. DbiCreateTable creates a file with the given name; no field descriptions are necessary.

Opening, importing and exporting text files

DbiOpenTable can be used to open a text file for import/export. The file can be opened as a delimited text file or as a fixed length text file.

Example 1: Opening a delimited text file

In this example, the text file dBASE.txt is opened as a delimited text file. The quotation mark (") is the delimiter character and comma is the field separator character.

```
DbiOpenTable (hDb, "DBASE.TXT", "ASCIIDRV-\"-,", NULL, NULL, 0,
dbiREADWRITE, dbiOPENEXCL, xltNONE, FALSE, NULL, &hCursor);
```

The *pszDriverType* argument of DbiOpenTable is used to indicate the field separator and the delimiter characters. The field separator and delimiter characters are passed through the *pszDriverType* argument as shown below:

```
"ASCIIDRV-<Delimiterchar>-<FieldSeparator>"
```

The field separator character separates the text file field values. The delimited character surrounds the text field types (alphanumeric or character) in the text file.

Example 2: Opening a fixed length text file

In this example, the text file dBASE.txt is opened as a fixed length text file:

```
DbiOpenTable (hDb, "DBASE.TXT", "ASCIIDRV", NULL, NULL, 0,
dbiREADWRITE, dbiOPENEXCL, xltNONE, FALSE, NULL, &hCursor);
```

When opening a fixed-length text file, no delimiter and separator characters are passed along with the *pszDriverType* argument.

{button ,AL(`textdriver')} Text driver topics

Field maps

Because no description of the fields is available when a text file is created, it is a good practice to set a field map on the cursor that is opened on that text file. The text driver uses this field map to interpret the data types of the fields in that text file.

When you open a text table, you can provide the field descriptor information by using the DbiSetFieldMap call. dBASE, FoxPro, Access, and Paradox go through the following steps in setting the field description information for a text table.

1 Obtains the field descriptors of the source/target table by using the function call DbiGetFieldDescs.

2 Obtains equivalent physical field descriptors of the text driver by using the call DbiTranslateRecordStructure.

3 Sets the field descriptor information on the text table by using the call DbiSetFieldMap.

If no field maps are set, the following behavior is expected:

The text file exists and has records:

Fixed-length Text	Delimited Text
iFlds = 1	iFlds = Calculated using the first record
fldType = CHAR	fldType = CHAR
fldLen = Calculated using first record	fldLen= (4k/iFlds) && less than 255.

The text file does not exist and has no records:

Fixed -length Text	Delimited Text
iFlds = 1	iFlds = 1
fldType = CHAR	fldType = CHAR
fldLen = 255	fldLen = 255

When a field map is set on a cursor that is opened as a text table, the source field descriptors (or destination field descriptors when importing) must be converted into text driver type descriptors. This step is necessary because some data types (for example, DBIDATE) have different field lengths in different driver types (for example, in Paradox, a DATE field is of four bytes long, while in dBASE a DATE field is eight bytes long).

The DbiTranslateRecordStructure call can be used to convert the logical or physical fields of a given driver type (that is, Paradox, dBASE, FoxPro, or Access) to the physical fields of the text driver. Then those physical text fields should be used in the DbiSetFieldMap call. When a field map is set on a text table, the *iFldType*, *iFldNum*, *iUnits1*, *iUnits2* and *iLen* elements should be set correctly in all the field descriptors.

Note: The Text driver supports files with field names in the first record because many applications export ASCII files with field names in the first line.

After a field map is set on the Text driver, DbiBatchMove can be used to import and export data to and from the text files. Refer to the online SnipIt code Import and Export examples.

Alternatively, you can bind schema information to a text table by storing the schema information of that text table in another text file. See Binding External Schema to Text Tables

---

{button ,AL(`textdriver')} Text driver topics

Binding external schema to text tables

Although you can set the field descriptors on text tables for use with export/import utilities, the BDE text driver can bind an external schema information to text tables. You bind schema information to a text table by storing the schema information of that text table in another text file.

The extension of the text file containing the schema information will be "sch". Thus, the name of the text file containing the schema information of the text table zzz.txt will be zzz.sch. If the text table has an extension other than "txt", extension of the schema file would still be "sch".

Schema File

All information in the schema file is case-insensitive.

Here is a sample schema file:

```
[CUSTOMER]                  // File name with no extension.
FILETYPE = VARYING          // Format: VARYING or FIXED
CHARSET = ascii       // Language driver name.
DELIMITER = "         // Delimiter for char fields.
SEPARATOR = ,               // Separator character
Field1 = Name,CHAR,12,0,0   // Field information
Field2 = Salary,FLOAT,8,2,12
```

The schema file has a format similar to Windows INI files. The file begins with the name of the table in brackets. The second line specifies the file format following the keyword FILETYPE: FIXED or VARYING.

FIXED format file
Each field always takes up a fixed number of characters in the file, and the data is padded with blanks as needed.

VARYING format file
Each field takes a variable number of characters, each character field is enclosed by DELIMITER characters, and the fields are separated by a SEPARATOR character. The DELIMITER and SEPARATOR must be specified for a VARYING format file, but not for a FIXED format file.

The CHARSET attribute specifies the name of the language driver to use. This is the base filename of the .LD file used for localization purposes.

The DELIMITED character surrounds the text field types (alphanumeric or character) in the text file. Delimited fields must be of character type.

The field SEPARATOR character separates the text file field values.

The remaining lines specify the attributes of the table's fields (columns). Each line must begin with "Fieldx = ", where x is the field number (that is. Field1, Field2, and so on).

Next appears a comma-delimited list specifying:

- Field name. Same restrictions as Paradox field names.
- Data type. The field data type. See below.
- Number of characters or units. Must be <= 20 for numeric data types. Total maximum number of characters for date/time data types (including / and : separators).
- Number of digits after the decimal (FLOAT only).
- Offset. Number of characters from the beginning of the line that the field begins. Used for FIXED and DELIMITED formats.

The following data types are supported:

```
CHAR      - Character
FLOAT     - 64-bit floating point
NUMBER    - 16-bit integer
BOOL      - Boolean (T or F)
LONGINT   - 32-bit long integer
DBIDATE   - Date field. Format specified in Registry
TIME      - Time field. Format specified in Registry
TIMESTAMP - Date/Time field. Format specified in Registry
```

Note: You specify date and time formats by using the BDE Administrator.

Example 1: VARYING format file

```
CUSTOMER.SCH:

[CUSTOMER]
Filetype=VARYING
Delimiter="
Separator=,
CharSet=ascii
Field1=Customer No,Float,20,04,00
Field2=Name,Char,30,00,20
Field3=Phone,Char,15,00,145
Field4=First Contact,Date,11,00,160

CUSTOMER.TXT:
1221.0000,"Kauai Dive Shoppe","808-555-0269",04/03/1994
1231.0000,"Unisco","809-555-3915",02/28/1994
1351.0000,"Sight Diver","357-6-876708",04/12/1994
1354.0000,"Cayman Divers World Unlimited","809-555-8576",04/17/1994
1356.0000,"Tom Sawyer Diving Centre","809-555-7281",04/20/1994
```

All the BDE API functions work with the text driver. To support external schema binding, the text driver includes the database property dbUSESCHEMAFILE applicable only to the text driver.

If the dbUSESCHEMAFILE property is set to true at the time of an export to a text table, the schema information of that text table is stored in a schema file. The DbiBatchMove function is used in exporting data to a text file. DbiBatchMove automatically stores the schema information while copying the data to a text table.

If the dbUSESCHEMAFILE flag is set to TRUE at the time of an import and a schema file exists for the text table, the text driver gets the field descriptors from the schema text file and sets them as the default fields for that text table. If the dbUSESCHEMAFILE flag is not set, you should define the field descriptions of the text table by using the function DbiSetFieldMap.

{button ,AL(`textdriver')} Text driver topics

Error handling

BDE functions return error codes to inform the calling program if the function succeeded or failed. The return value is DBIERR_NONE when the function was successful. If an error occurs during the execution of a BDE call, any of the BDE subsystems may push an error context onto the common BDE error stack. This error context allows the application to examine potentially more detailed information about the cause of any error.

Several BDE functions enable the application to retrieve different levels of information about errors:

Error function	Level of information returned
DbiGetErrorEntry	Allows any entry on the error stack to be returned. This is the only function that returns native server error codes for SQL drivers.
DbiGetErrorString	When the application passes the error code, this function returns a more detailed message; for example, "At end of table."
DbiGetErrorContext	Pass it an error context type, such as "ecTABLENAME," and it returns specific information; in this case, the full path name of the table involved in the error.
DbiGetErrorInfo	Returns the error code, descriptive error message, and error contexts for the first four error messages on the error stack.

For more specific instructions on using error messages, see the following topics:

{button ,AL(`errorhandling')} Error handling topics

Using DbiGetErrorEntry to access the error stack

Every error generated as a result of a BDE function call goes onto an error stack. Error stack entries begin with 1. Each stack entry contains a DBIERR code, and possibly a native server error code and a native server error message. (The only way for the application to get native server errors is to access the error stack.)

The application can access the error stack by calling DbiGetErrorEntry. This function returns the error code and description of a specified error stack entry. The application can optionally pass a pointer to a buffer to receive the native error code and the native error message.

DbiGetErrorEntry returns the error code DBIERR_NONE for stack entries beyond the current error stack, so this successful return can be used as a loop termination. For example, if error entry 1 returns an error code of DBIERR_NONE, there are no errors on the stack. The stack may be traversed multiple times or combined with other error interface calls, but non-error routine BDE calls reset the error stack.

{button ,AL(`errorhandling')} Error handling topics

Using DbiGetErrorString to get a detailed error message

DbiGetErrorString returns a more detailed message for the error code returned by DbiGetErrorEntry. The application passes the error code and receives the error message. For example, if DbiGetErrorString is called with the error code DBIERR_EOF, it returns the string "At End of Table." BDE keeps the error strings as Windows string resources in the .DLL file with the IDR prefix. This way the application developer can translate or customize them as needed by using a product such as Resource Workshop.

{button ,AL(`errorhandling')} Error handling topics

Using DbiGetErrorContext to get more specific information

DbiGetErrorContext returns more specific error information about the context of an error, such as the name of the offending table or field. When an error occurs, the error context is logged by the BDE engine. Other error contexts can be logged as well, so rather than force the user to scan each error context individually, DbiGetErrorContext searches for a particular context type. The application inputs the error context type and the function returns a character string.

Error Context Types

Error contexts can be one of the following types:

Type	Description
ecTOKEN	Token (For QBE)
ecTABLENAME	Table name
ecFIELDNAME	Field name
ecIMAGEROW	Image row (For QBE)
ecUSERNAME	For example, in lock conflicts, user involved
ecFILENAME	File name
ecINDEXNAME	Index name
ecDIRNAME	Directory name
ecKEYNAME	Key name
ecALIAS	Alias
ecDRIVENAME	Drive name ('c:')
ecNATIVECODE	Native error code
ecNATIVEMSG	Native error message
ecLINENUMBER	Line number
ecCAPABILITY	Capability

For example, if the application attempts to open a nonexistent table by using DbiOpenTable, it receives an error code of DBIERR_NOSUCHFILE. To determine which table name is associated with the error condition, the application calls DbiGetErrorContext (ecTABLENAME, buffer), which returns the full path name of the table. If there is no table name associated with the error, the buffer is empty.

{button ,AL(`errorhandling')} Error handling topics

Using DbiGetErrorInfo to get immediate information

DbiGetErrorInfo provides immediate descriptive error information about the last error that occurred. This information consists of the DBIResult error code, an error message in ANSI characters corresponding to the code, and up to four associated error contexts. For example, if the error message is "Table Not Found," the user might want to know the table name. The BDE engine logged the table name with the error context ecTABLENAME, which can be found in one of the contexts contained in the DBIErrInfo structure.

The application calls DbiGetErrorInfo which returns relevant error information in the provided DBIErrInfo structure. These structure types are shown in the following table.

DbiErrorInfo Structure

Type	Name	Description
DBIResult	*iError*	Last error code returned
DBIMSG	*szErrCode*	More descriptive information
DBIMSG	*szContext1*	Context 1
DBIMSG	*szContext2*	Context 2
DBIMSG	*szContext3*	Context 3
DBIMSG	*szContext4*	Context 4

This function immediately displays up to four error contexts to the user, while the function DbiGetErrorContext returns only the specific error context requested by the user.

If all that is required is a formatted error message for the end user, DbiGetErrorInfo is a more convenient way to get it.

These examples shows how to get information about an error when a BDE function returns a value other than DBIERR_NONE:

```
hDBIDb      hDb;
     DBIResult   rslt;
     DBIMSG      dbiStatus;
 // Open a STANDARD database
  rslt = DbiOpenDatabase(NULL, NULL, dbiREADWRITE,     dbiOPENSHARED,
                         NULL, 0, NULL, NULL, &hDb);
  if (rslt != DBIERR_NONE)
  {
      // An error occurred. Retrieve the error string.
      DbiGetErrorString(rslt, dbiStatus);
  }
```

{button ,AL(`errorhandling')} Error handling topics

Debugging

BDE provides SQL Trace to track SQL statements sent to the servers when BDE functions execute.

SQL Trace

The SQL Trace facility is a useful debugging tool that opens the SQL "black box," allowing you to track the SQL statements sent to the servers when their BDE function calls are executed. SQL Trace is implemented as the SQL TRACE option in the Windows Registry and as a callback to return trace information.

Configuration option

To set the SQL trace mode, use the SQLTRACE option in the DRIVER\INIT section of the Windows Registry (settings\driver*driver_name*\init\TRACE MODE) for the appropriate driver. The option takes a numeric value (actually a bit mask) that determines how much information to log. The Windows OutputDebugString call is used to output the requested information to the debug window. The following table shows which information is logged based on bit settings:

Bit Settings	Logged Information
0x0001	prepared query statement
0x0002	executed query statements
0x0004	vendor errors
0x0008	statement ops (that is: allocate, free)
0x0010	connect / disconnect
0x0020	transaction
0x0040	BLOB I/O
0x0080	miscellaneous
0x0100	vendor calls

Examples

1. To trace only prepared and executed query statements, sets bits 0x0001 and 0x0002 (that is, set SQLTRACE to 3)
2. To trace only vendor calls, set bit 0x0100 (that is, set SQLTRACE to 256)

Because the value of SQLTRACE is evaluated as an unsigned integer, a value of -1 will turn on all bits, and therefore all of the above events will be traced.

dbTRACEMODE

To programatically override the DRIVER option for any database, use the database property dbTRACEMODE with DbiSetProp. TRACE MODE is a DRIVER option in the Registry that determines the trace behavior for all database operations associated with the driver.

cbTRACE

To retrieve trace information, use the system-level callback cbTRACE. The trace string retrieved through the callback is the same as that which goes to the debug window via OutputDebugString.

The TraceDESC structure is used to return trace information to the callback:

```
typedef struct       // trace callback info
       {
          TRACECat    eTraceCat;     // trace category
          UINT16      uTotalMsgLen;  // total message length
          CHAR        pszTrace[];    // trace string
                                     // (recommended size = DBIMAXTRACELEN
  (8192))
       } TRACEDesc;
```

```
typedef enum         // trace categories
     {
        traceUNKNOWN   = 0x0000,
        traceQPREPARE  = 0x0001,   // prepared query statements
        traceQEXECUTE  = 0x0002,   // executed query statements
        traceERROR     = 0x0004,   // vendor errors
        traceSTMT      = 0x0008,   // statement ops (i.e. allocate, free)
        traceCONNECT   = 0x0010,   // connect / disconnect
        traceTRANSACT  = 0x0020,   // transaction
        traceBLOB      = 0x0040,   // blob i/o
        traceMISC      = 0x0080,   // misc.
        traceVENDOR    = 0x0100,   // vendor calls
     } TRACECat;
```

The TRACECat enums have the same bit sequence used to set the TRACE MODE configuration option, and can also be used (singularly or piped together) as input to the dbTRACEMODE database property. You can use the uTotalMsgLen field of the TRACEDesc structure to determine whether the returned string (in pszTrace) has been truncated.

Example: Registering a cbTRACE Callback:

Note: Before calling DbiRegisterCallBack() for SQLTRACE, the argument pTraceInfo must be allocated for the size of (TRACEDESC) plus DBIMAXTRACELEN.

```
DbiRegisterCallBack
       (NULL,
       cbTRACE,
       iClientData,
       sizeof (TRACEDesc) + DBIMAXTRACELEN,
       (pVOID)pTraceInfo,          // ptr to client-allocated TRACEDesc
       (pfDBICallBack) lpfnTrace);
```

Improving BDE performance

Here are a few general programming practices to help improve overall BDE performance in accessing tables:

1 Keep the number of maintained secondary indexes to a minimum; sometimes it is better to delete the index and recreate it than to perform a number of table operations with the indexes in place.

2 If possible, increase the size of the swap buffer and the number of file handles that BDE has available to it. This will decrease BDE's need to swap resources.
Note: Be sure to increase the file handles available to your application by using SetHandleCount. Also, in IDAPI.CFG, increase the number of file handles available to BDE.

3 Open the table exclusively.

4 Batch as many operations as possible--do not read or write records one at a time. Use DbiBatchMove, DbiCopyTable, DbiReadBlock, and/or DbiWriteBlock.

5 When using DbiWriteBlock, try to work in multiples of the physical block size, usually 2K or 4K.

6 If you are opening and closing one or more tables repeatedly, consider calling DbiAcqPersistTableLock on a non-existent file after you initialize the BDE. This will create the .LCK file so that it will not have to be created each time a table is opened, created, and so on. (**Note:** You'll also want to call DbiRelPersistTableLock before calling DbiExit). This applies to Paradox tables only.

7 Work with in-memory tables when possible.

8 When working with remote data sources that support transactions, use explicit transactions. For example, each insert to a table on an SQL server will force a transaction to be started and committed. This adds a lot of overhead when inserting a large group of records. Instead, start a transaction, insert a group of records, and then commit the changes as a group.

All options mentioned below are configurable using the BDE Administrator:

1 Set LOCAL SHARE to False. This option should only be TRUE if both BDE and non-BDE applications are accessing dBASE or Paradox tables simultaneously. Borland products all use the BDE to access Paradox and dBASE tables, so this option can be FALSE if you're using only Borland applications.

2 To improve performance of opening tables and updateable queries for a Server BDE Alias, set ENABLE SCHEMA CACHE to TRUE. Do not use the schema cache if your application is constantly creating tables or altering the structures of existing tables.

3 Try adjusting BATCH COUNT. This is the amount of records processed in a single transaction in a BatchMove operation. This also affects the performance of the Data Migration Expert.

See also: SQL performance tips

{button ,AL(`applicationdevelopment')} Application development

Using the function reference

You can find a complete description of each BDE function by looking in the task-related tables in Function Reference, Categorical.

Alternatively, you can quickly access the topic for any function by searching the complete list in Function Reference, Alphabetical.

Each BDE function name begins with the prefix Dbi. The remainder of the name describes the function's use. For example, DbiGetClientInfo is the name of the BDE function that retrieves information about the client application environment.

Syntax is provided in both C and in Delphi (Pascal) languages.

Next to the title of each function topic you'll see two Examples buttons, one in C code, the other in Delphi (Pascal). Click here to display code examples that you can copy and paste into your applications.

See the following topics for general conventions and definitions that will assist you in understanding and making effective use of the BDE function reference:

{button ,AL(`usingfuncref')} Using the function reference

Syntax conventions

The C syntax for BDE function calls is:

```
DBIResult DBIFN DbiFunctionName (argument1, argument2, argument3
                                 ...);
```

Each function definition includes the elements described in this table:

Element	Description
Function name	Name of function
Examples buttons	Click to display a window of code examples (either C or Delphi) that you can copy and paste into your application.
Description	Summary description of function
Syntax	Diagram of the function and parameters in both C and Delphi coding styles.
Parameters	Descriptions of each parameter
Usage	Detailed information about using the function
Prerequisites	State required before function is called
Completion state	State after the function completes
DBIResult return values	Description of possible values returned after the function completes, if any
See Also	Cross references to other related functions

Each function definition observes these typographical conventions:

Convention	Purpose	Example
`Courier font`	Keywords that must be typed exactly as they appear when used (case-sensitive).	DBIResult DBIFN DbiInit();
italic	Variables and parameters passed to the function, returned from the function, or both.	(*hCursor*, *piRecords*, *pBuf*)
[]	Brackets enclose optional parameters. Optional parameters can be set to NULL.	*iPosOffset*, [*eLock*]

{button ,AL(`usingfuncref')} Using the function reference

Variable names

Each variable name used in this reference begins with a standard prefix and appears italicized in text.. These prefixes indicate the variable's type or use, as described in the following table:

Prefix	Variable type or use
a	The declared variable is an array.
b	The declared variable is of the boolean type.
dt	The declared variable is of the datetime type.
e	The content of the declared variable is of the enumerated type.
h	The declared variable is used as a handle.
i	The declared variable is an integer.
p	The declared variable is a pointer.
sz	The declared variable is a null-terminated character string.
tm	The declared variable is of the timestamp type.

Prefixes can be combined to more completely describe the variable's use. For example, the prefix psz in the variable name *pszIndexName* indicates that the variable is a pointer to a null-terminated character string, in this case, where the name of the index is stored.

{button ,AL(`usingfuncref')} Using the function reference

Constants

The following table lists the constants used to define maximum limits throughout this reference:

Constant	Limit	Description
DBIMAXBOOKMARKLEN	4104	Maximum bookmark length
DBIMAXDRIVELEN	2	Maximum drive length (if Win32 not defined)
DBIMAXDRIVELEN	127	Maximum drive length (if Win32 defined)
DBIMAXDRSQLSTR	8192	Max size of SQL constraint
DBIMAXEXTLEN	3	Maximum file extension length, not including the extension delimiter "."
DBIMAXFLDSINKEY	16	Maximum number of fields in a key
DBIMAXFLDSINSEC	256	Maximum fields in security specification
DBIMAXFUNCNAMELEN	255	Max function name length
DBIMAXKEYEXPLEN	220	Maximum key expression length
DBIMAXMSGLEN	127	Maximum message length (allocate 128)
DBIMAXNAMELEN	31	Maximum object name limit (such as, table, field)
DBIMAXPATHLEN	81	Maximum path plus filename length. Allocate 80. (If Win32 not defined.)
DBIMAXPATHLEN	260	Maximum path plus filename length, excluding zero termination. (If Win32 defined.)
DBIMAXPICTLEN	175	Maximum picture length
DBIMAXSCFIELDS	32	Maximum number of fields in an optional parameter list
DBIMAXSCFLDLEN	128	Maximum field length in an optional parameter list
DBIMAXSCRECSIZE	2048	Maximum record size in an optional parameter list, computed as DBIMAXSCFIELDS*DBIMAXSCFLDLEN.
DBIMAXSPNAMELEN	64	Maximum stored procedure name length
DBIMAXTBLNAMELEN	127	Maximum table name length (if Win32 not defined)
DBIMAXTBLNAMELEN	260	Maximum table name length (if Win32 defined)
DBIMAXTRACELEN	8192	Maximum trace message length
DBIMAXTYPEDESC	127	Maximum type description size
DBIMAXUSERNAMELEN	14	Maximum user name (general)
DBIMAXVCHKLEN	255	Maximum validity check length
DBIMAXXBUSERNAMELEN	12	Maximum user name length for xBASE

{button ,AL(`usingfuncref')} Using the function reference

#defines

The following table lists the #defines used throughout this reference:

#define	Definition
NULL	(0)
VOID	void
INT8	char
CHAR	char
BYTE	unsigned char
UINT8	unsigned char
INT16	int (if defined FLAT); short
UINT16	unsigned short (if defined FLAT); unsigned int
INT32	long
UINT32	unsigned long
BOOL	short (if defined FLAT) int
DFLOAT	double
DBIDATE	long
TIME	long
TIMESTAMP	double
DBIFN	pascal far
UINT16	DBIResult

{button ,AL(`usingfuncref')} Using the function reference

Typedefs

The following table lists the typedefs used throughout this reference:

typedefs	Definition
VOID	far *pVOID
pVOID	far *ppVOID
CHAR	far *pCHAR
BYTE	far *pBYTE
INT8	far *pINT8
UINT8	far *pUINT8
INT16	far *pINT16
UINT16	far *pUINT16
INT32	far *pINT32
UINT32	far *pUINT32
DFLOAT	far *pFLOAT
DBIDATE	far *pDATE
TIME	far *pTIME
BOOL	far *pBOOL
TIMESTAMP	far *pTIMESTAMP
pBYTE	far *ppBYTE
pCHAR	far *ppCHAR
pBOOL	far *ppBOOL
DBIResult	far *pDBIResult

{button ,AL(`usingfuncref')} Using the function reference

Object definitions

The following objects are defined:

Type	Object	Description
UINT32	hDBIObj	Generic object handle
hDBIObj	hDBIDb	Database handle
hDBIObj	hDBIQry	Query handle
hDBIObj	hDBIStmt	Statement handle ("new query")
hDBIObj	hDBICur	Cursor handle
hDBIObj	hDBISes	Session handle
hDBIObj	hDBIXlt	Translation handle
UINT32	hDBIXact	Transaction handle
hDBIObj	far *phDBIObj	Pointer to generic object handle
hDBICfg	far *phDBICfg	Pointer to configuration handle
hDBIDb	far *phDBIDb	Pointer to database handle
hDBIQry	far *phDBIQry	Pointer to query handle
hDBIStmt	far *phDBIStmt	Pointer to statement handle
hDBICur	far *phDBICur	Pointer to cursor handle
hDBISes	far *phDBISes	Pointer to session handle
hDBIXlt	far *phDBIXlt	Pointer to translation handle
hDBIXact	far *phDBIXact	Pointer to transaction handle

{button ,AL(`usingfuncref')} Using the function reference

Buffer Typedefs

The following typedefs for buffers of various common sizes are defined:

Type	typedef	Description
DBIPATH	CHAR[DBIMAXPATHLEN+1]	Holds a DOS path
DBINAME	CHAR[DBIMAXNAMELEN+1]	Holds a name
DBIEXT	CHAR[DBIMAXEXTLEN+1]	Holds a file extension
DBIDOTEXT	CHAR[DBIMAXEXTLEN+2]	Holds a file extension including "."
DBIDRIVE	CHAR[DBIMAXDRIVELEN+1]	Holds a drive name
DBITBLNAME	CHAR[DBIMAXTBLNAMELEN+1]	Holds a table name
DBIUSERNAME	CHAR[DBIMAXUSERNAMELEN+1]	Holds a user name
DBIKEY	UINT16[DBIMAXFLDSINKEY]	Holds a list of fields in a key
DBIKEYEXP	CHAR[DBIMAXKEYEXPLEN+1];	Holds a key expression
DBIVCHK	BYTE[DBIMAXVCHKLEN+1]	Holds a validity check
DBIPICT	CHAR[DBIMAXPICTLEN+1]	Holds a picture clause
DBIMSG	CHAR[DBIMAXMSGLEN+1]	Holds an error message

{button ,AL(`usingfuncref')} Using the function reference

Function reference, categorical

Each of the BDE functions documented in this reference fall into one of the categories listed in the table below:

Function Type	Purpose
Capability or schema	Returns information about database schema.
Cursor	Returns information or affects cursors and bookmarks.
Data access	Performs specific data access operations.
Database	Returns information or performs related tasks.
Date/time/number	Handles formats for the session.
Environment	Returns information or affects the client application environment.
Error handling	Returns information or performs related tasks.
Index	Returns information or affects indexes.
Locking	Returns information or affects locks.
Query	Performs query tasks.
Session	Returns information or affects a session.
Table	Returns information or performs table-wide operations.
Transaction	Returns information or performs related tasks.

Environment functions

Each BDE function listed below returns information about the client application environment such as the supported table, field and index types for the driver type, or the available driver types. Also listed are functions that perform a task that affects the client application environment, such as loading a driver.

Function	Description
DbiAddAlias	Adds an alias to the BDE configuration file (IDAPI.CFG).
DbiAddDriver	Adds a driver to the BDE configuration file (IDAPI.CFG).
DbiAnsiToNative	Multipurpose translate function.
DbiDeleteAlias	Deletes an alias from the BDE configuration file (IDAPI.CFG).
DbiDeleteDriver	Deletes a driver from the BDE configuration file (IDAPI.CFG).
DbiDllExit	Prepares the BDE to be disconnected within a Dll.
DbiExit	Disconnects the client application from BDE.
DbiGetClientInfo	Retrieves system-level information about the client application environment.
DbiGetDriverDesc	Retrieves a description of a driver.
DbiGetLdName	Retrieves the name of the language driver associated with the specified object name (table name).
DbiGetLdObj	Retrieves the language driver object associated with the given cursor.
DbiGetNetUserName	Retrieves the user's network login name. User names should be available for all networks supported by Microsoft Windows.
DbiGetProp	Returns a property of an object.
DbiGetSysConfig	Retrieves BDE system configuration information.
DbiGetSysInfo	Retrieves system status and information.
DbiGetSysVersion	Retrieves the system version information, including the BDE version number, date, and time, and the client interface version number.
DbiInit	Initializes the BDE environment.
DbiLoadDriver	Loads a given driver.
DbiNativeToAnsi	Translates a string in the native language driver to an ANSI string.
DbiOpenCfgInfoList	Returns a handle to an in-memory table listing all the nodes in the configuration file accessible by the specified path.
DbiOpenDriverList	Creates an in-memory table containing a list of driver names available to the

	client application.
DbiOpenFieldTypesList	Creates an in-memory table containing a list of field types supported by the table type for the driver type.
DbiOpenFunctionArgList	Returns a list of arguments to a data source function.
DbiOpenFunctionList	Returns a description of a data source function.
DbiOpenIndexTypesList	Creates an in-memory table containing a list of all supported index types for the driver type.
DbiOpenLdList	Creates an in-memory table containing a list of available language drivers.
DbiOpenTableList	Creates an in-memory table with information about all the tables accessible to the client application.
DbiOpenTableTypesList	Creates an in-memory table listing table type names for the given driver.
DbiOpenUserList	Creates an in-memory table containing a list of users sharing the same network file.
DbiSetProp	Sets the specified property of an object to a given value.

Session functions

Each BDE function listed below returns information about a session, or performs a task that affects the session, such as adding a password.

Function	Description
DbiAddPassword	Adds a password to the current session.
DbiCheckRefresh	Checks for remote updates to tables for all cursors in the current session, and refreshes the cursors if changed.
DbiCloseSession	Closes the session associated with the given session handle.
DbiDropPassword	Removes a password from the current session.
DbiGetCallBack	Returns a pointer to the function previously registered by the client for the given callback type.
DbiGetCurrSession	Returns the handle associated with the current session.
DbiGetDateFormat	Gets the date format for the current session.
DbiGetNumberFormat	Gets the number format for the current session.
DbiGetSesInfo	Retrieves the environment settings for the current session.
DbiGetTimeFormat	Gets the time format for the current session.
DbiRegisterCallBack	Registers a callback function for the client application.
DbiSetCurrSession	Sets the current session of the client application to the session associated with *hSes*.
DbiSetDateFormat	Sets the date format for the current session.
DbiSetNumberFormat	Sets the number format for the current session.
DbiSetPrivateDir	Sets the private directory for the current session.
DbiSetTimeFormat	Sets the time format for the current session.
DbiStartSession	Starts a new session for the client application.

Error handling functions

Each BDE function listed below returns error handling information, or performs a task that relates to error handling.

Function	Description
DbiGetErrorContext	After receiving an error code back from a call, enables the client to probe BDE for more specific error information.
DbiGetErrorEntry	Returns the error description of a specified error stack entry.
DbiGetErrorInfo	Provides descriptive error information about the last error that occurred.
DbiGetErrorString	Returns the message associated with a given error code.

Locking functions

Each BDE function listed below returns information about lock status, or acquires or releases a lock at the table or record level.

Function	Description
DbiAcqPersistTableLock	Acquires an exclusive persistent lock on the table preventing other users from using the table or creating a table of the same name.
DbiAcqTableLock	Acquires a table-level lock on the table associated with the given cursor.
DbiGetRecord	Record positioning functions have a lock parameter.
DbiIsRecordLocked	Checks the lock status of the current record.
DbiIsTableLocked	Returns the number of locks of a specified type acquired on the table associated with the given session.
DbiIsTableShared	Determines whether the table is physically shared or not.
DbiOpenLockList	Creates an in-memory table containing a list of locks acquired on the table.
DbiOpenUserList	Creates an in-memory table containing a list of users sharing the same network file.
DbiRelPersistTableLock	Releases the persistent table lock on the specified table.
DbiRelRecordLock	Releases the record lock on either the current record of the cursor or only the locks acquired in the current session.
DbiRelTableLock	Releases table locks of the specified type associated with the current session (the session in which the cursor was created).
DbiSetLockRetry	Sets the table and record lock retry time for the current session.

Cursor functions

Each BDE function listed below returns information about a cursor, or performs a task that performs a cursor-related task such as positioning of a cursor, linking of cursors, creating and closing cursors, counting of records associated with a cursor, filtering, setting and comparing bookmarks, and refreshing all buffers associated with a cursor.

Function	Description
DbiActivateFilter	Activates a filter.
DbiAddFilter	Adds a filter to a table, but does not activate the filter (the record set is not yet altered).
DbiApplyDelayedUpdates	When cached updates cursor layer is active, writes all modifications made to cached data to the underlying database.
DbiBeginDelayedUpdates	Creates a cached updates cursor layer so that users can make extended changes to temporarily cached table data without writing to the actual table, thereby minimizing resource locking.
DbiBeginLinkMode	Converts a cursor to a link cursor. Given an open cursor, prepares for linked access. Returns a new cursor.
DbiCloneCursor	Creates a new cursor (clone cursor) which has the same result set as the given cursor (source cursor).
DbiCloseCursor	Closes a previously opened cursor.
DbiCompareBookMarks	Compares the relative positions of two bookmarks in the result set associated with the cursor.
DbiDeactivateFilter	Temporarily stops the specified filter from affecting the record set by turning the filter off.
DbiDropFilter	Deactivates and removes a filter from memory, and frees all resources.
DbiEndDelayedUpdates	Closes a cached updates cursor layer ending the cached updates mode.
DbiEndLinkMode	Ends linked cursor mode, and returns the original cursor.
DbiExtractKey	Retrieves the key value for the current record of the given cursor or from the supplied record buffer.
DbiForceRecordReread	Rereads a single record from the server on demand, refreshing one row

	only, rather than clearing the cache.
DbiForceReread	Refreshes all buffers associated with the cursor, if necessary.
DbiFormFullName	Returns the fully qualified table name.
DbiGetBookMark	Saves the current position of a cursor to the client-supplied buffer called a bookmark.
DbiGetCursorForTable	Finds the cursor for the given table.
DbiGetCursorProps	Returns the properties of the cursor.
DbiGetExactRecordCount	Retrieves the current exact number of records associated with the cursor.
DbiGetFieldDescs	Retrieves a list of descriptors for all the fields in the table associated with the cursor.
DbiGetLinkStatus	Returns the link status of the cursor.
DbiGetNextRecord	Retrieves the next record in the table associated with the cursor.
DbiGetPriorRecord	Retrieves the previous record in the table associated with the given cursor.
DbiGetProp	Returns a property of an object.
DbiGetRecord	Retrieves the current record, if any, in the table associated with the cursor.
DbiGetRecordCount	Retrieves the current number of records associated with the cursor.
DbiGetRecordForKey	Finds and retrieves a record matching a key and positions the cursor on that record.
DbiGetRelativeRecord	Positions the cursor on a record in the table relative to the current position of the cursor.
DbiGetSeqNo	Retrieves the sequence number of the current record in the table associated with the cursor.
DbiLinkDetail	Establishes a link between two tables such that the detail table has its record set limited to the set of records matching the linking key values of the master table cursor.
DbiLinkDetailToExp	Links the detail cursor to the master cursor using an expression.
DbiMakePermanent	Changes a temporary table created by DbiCreateTempTable into a permanent table.
DbiOpenTable	Opens the given table for access and associates a cursor handle with the opened table.
DbiResetRange	Removes the specified table's limited range previously

	established by the function DbiSetRange.
DbiSaveChanges	Forces all updated records associated with the cursor to disk.
DbiSetFieldMap	Sets a field map of the table associated with the given cursor.
DbiSetProp	Sets the specified property of an object to a given value.
DbiSetRange	Sets a range on the result set associated with the cursor.
DbiSetToBegin	Positions the cursor to BOF (just before the first record).
DbiSetToBookMark	Positions the cursor to the location saved in the specified bookmark.
DbiSetToCursor	Sets the position of one cursor (the destination cursor) to that of another (the source cursor).
DbiSetToEnd	Positions the cursor to EOF (just after the last record).
DbiSetToKey	Positions an index-based cursor on a key value.
DbiSetToRecordNo	Positions the cursor of a dBASE or FoxPro table to the given physical record number.
DbiSetToSeqNo	Positions the cursor to the specified sequence number of a Paradox table.
DbiUnlinkDetail	Removes a link between two cursors.

Index functions

Each BDE function listed below returns information about an index or indexes, or performs a task that affects an index, such as dropping it, deleting it, or adding it.

Function	Description
DbiAddIndex	Creates an index on an existing table.
DbiCloseIndex	Closes the specified index on a cursor.
DbiCompareKeys	Compares two key values based on the current index of the cursor.
DbiDeleteIndex	Drops an index on a table.
DbiExtractKey	Retrieves the key value for the current record of the given cursor or from the supplied record buffer.
DbiGetIndexDesc	Retrieves the properties of the given index associated with the cursor.
DbiGetIndexDescs	Retrieves index properties.
DbiGetIndexForField	Returns the description of any useful index on the specified field.
DbiGetIndexSeqNo	Retrieves the ordinal number of the index in the index list of the specified cursor.
DbiGetIndexTypeDesc	Retrieves a description of the index type.
DbiOpenIndex	Opens the index for the table associated with the cursor.
DbiRegenIndex	Regenerates an index to make sure that it is up-to-date (all records currently in the table are included in the index and are in the index order).
DbiRegenIndexes	Regenerates all out-of-date indexes on a given table.
DbiSwitchToIndex	Allows the user to change the active index order of the given cursor.

Query functions

Each BDE function listed below performs a query task.

Function	Description
DbiGetProp	Returns a property of an object.
DbiQAlloc	Allocates a new statement handle for a prepared query.
DbiQExec	Executes the previously prepared query identified by the supplied statement handle and returns a cursor to the result set, if one is generated.
DbiQExecDirect	Executes a SQL or QBE query and returns a cursor to the result set, if one is generated.
DbiQExecProcDirect	Executes a stored procedure and returns a cursor to the result set, if one is generated.
DbiQFree	Frees the resources associated with a previously prepared query identified by the supplied statement handle.
DbiQGetBaseDescs	Returns the original database, table, and field names of the fields that make up the result set of a query.
DbiQInstantiateAnswer	Creates a permanent table from the cursor to the result set.
DbiQPrepare	Prepares a SQL or QBE query for execution, and returns a handle to a statement containing the prepared query.
DbiQPrepareProc	Prepares and optionally binds parameters for a stored procedure.
DbiQSetParams	Associates data with parameter markers embedded within a prepared query.
DbiQSetProcParams	Binds parameters for a stored procedure prepared with DbiQPrepareProc.
DbiSetProp	Sets the specified property of an object to a given value.
DbiValidateProp	Validates a property.

Database functions

The BDE functions listed below return information about a specific database, available databases, or perform a database-related task.

Function	Description
DbiCloseDatabase	Closes a database and all tables associated with this database handle.
DbiGetDatabaseDesc	Retrieves the description of the specified database from the configuration file.
DbiGetDirectory	Retrieves the current working directory or the default directory.
DbiOpenDatabase	Opens a database in the current session and returns a database handle.
DbiOpenDatabaseList	Creates an in-memory table containing a list of accessible databases and their descriptions.
DbiOpenFileList	Opens a cursor on the virtual table containing all the tables accessible by the client application and their descriptions.
DbiOpenIndexList	Opens a cursor on an in-memory table listing the indexes on a specified table, along with their descriptions.
DbiOpenTableList	Creates an in-memory table with information about all the tables accessible to the client application.
DbiSetDirectory	Sets the current directory for a standard database.

Table functions

Each BDE function listed below returns information about a specific table, such as all the locks acquired on the table, all the referential integrity links on the table, the indexes open on the table, or whether or not the table is shared. Or, it performs a table-wide operation, such as copying and deleting.

Function	Description
DbiBatchMove	Appends, updates, subtracts, and copies records or fields from a source table to a destination table.
DbiCopyTable	Duplicates the specified source table to a destination table.
DbiCreateInMemTable	Creates a temporary, in-memory table.
DbiCreateTable	Creates a table.
DbiCreateTempTable	Creates a temporary table that is deleted when the cursor is closed, unless the call is followed by a call to DbiMakePermanent.
DbiDeleteTable	Deletes a table.
DbiDoRestructure	Changes the properties of a table.
DbiEmptyTable	Deletes all records from the table associated with the specified table cursor handle or table name.
DbiGetTableOpenCount	Returns the total number of cursors that are open on the specified table.
DbiGetTableTypeDesc	Returns a description of the capabilities of the table type for the driver type.
DbiIsTableLocked	Returns the number of locks of a specified type acquired on the table associated with the given session.
DbiIsTableShared	Determines whether the table is physically shared or not.
DbiMakePermanent	Changes a temporary table created by DbiCreateTempTable into a permanent table.
DbiOpenFamilyList	Creates an in-memory table listing the family members associated with a specified table.
DbiOpenFieldList	Creates an in-memory table listing the fields in a specified table and their descriptions.
DbiOpenIndexList	Opens a cursor on an in-memory table listing the indexes on a specified table, along with their descriptions.
DbiOpenLockList	Creates an in-memory table containing a list of locks acquired on the table

	associated with the cursor.
DbiOpenRintList	Creates an in-memory table listing the referential integrity links for a specified table, along with their descriptions.
DbiOpenSecurityList	Creates an in-memory table listing record-level security information about a specified table.
DbiOpenTable	Opens the given table for access and associates a cursor handle with the opened table.
DbiPackTable	Optimizes table space by rebuilding the table associated with the cursor and releasing any free space.
DbiQInstantiateAnswer	Creates a permanent table from a cursor handle.
DbiRegenIndexes	Regenerates all out-of-date indexes on a given table.
DbiRenameTable	Renames the table and all of its resources to the new name specified.
DbiSaveChanges	Forces all updated records associated with the table to disk.
DbiSortTable	Sorts an opened or closed table, either into itself or into a destination table. There are options to remove duplicates, to enable case-insensitive sorts and special sort functions, and to control the number of records sorted.

Data access functions

Each BDE function listed below accesses data in a table.

Function	Description
DbiAppendRecord	Appends a record to the end of the table associated with the given cursor.
DbiDeleteRecord	Deletes the current record of the given cursor.
DbiFreeBlob	Closes the BLOB handle located within the specified record buffer.
DbiGetBlob	Retrieves data from the specified BLOB field.
DbiGetBlobHeading	Retrieves information about a BLOB field from the BLOB heading in the record buffer.
DbiGetBlobSize	Retrieves the size of the specified BLOB field in bytes.
DbiGetField	Retrieves the data contents of the requested field from the record buffer.
DbiGetFieldDescs	Retrieves a list of descriptors for all the fields in the table associated with the cursor.
DbiGetFieldTypeDesc	Retrieves a description of the specified field type.
DbiInitRecord	Initializes the record buffer to a blank record according to the data types of the fields.
DbiInsertRecord	Inserts a new record into the table associated with the given cursor.
DbiModifyRecord	Modifies the current record of table associated with the cursor with the data supplied.
DbiOpenBlob	Prepares the cursor's record buffer to access a BLOB field.
DbiPutBlob	Writes data into an open BLOB field.
DbiPutField	Writes the field value to the correct location in the supplied record buffer.
DbiReadBlock	Reads a specified number of records (starting from the next position of the cursor) into a buffer.
DbiSaveChanges	Forces all updated records associated with the cursor to disk.
DbiSetFieldMap	Sets a field map of the table associated with the given cursor.
DbiTruncateBlob	Shortens the size of the contents of a BLOB field, or deletes the contents of a BLOB field from the record, by shortening it to zero.
DbiUndeleteRecord	Undeletes a dBASE or FoxPro record that has been marked for deletion (a "soft" delete).
DbiVerifyField	Verifies that the data specified is a valid data type for the field specified, and that all validity checks in place for the field are satisfied. It can

also be used to check if a field is blank.

DbiWriteBlock Writes a block of records to the table associated with the cursor.

Capability or schema functions

Each BDE function listed below returns information about a data source's capabilties, or about its schema.

Function	Description
DbiOpenCfgInfoList	Returns a handle to an in-memory table listing all the nodes in the configuration file accessible by the specified path.
DbiOpenDatabaseList	Creates an in-memory table containing a list of accessible databases and their descriptions.
DbiOpenDriverList	Creates an in-memory table containing a list of driver names available to the client application.
DbiOpenFamilyList	Creates an in-memory table listing the family members associated with a specified table.
DbiOpenFieldList	Creates an in-memory table listing the fields in a specified table and their descriptions.
DbiOpenFieldTypesList	Creates an in-memory table containing a list of field types supported by the table type for the driver type.
DbiOpenFunctionArgList	Returns a list of arguments to a data source function.
DbiOpenFunctionList	Returns a description of a data source function.
DbiOpenIndexList	Opens a cursor on an in-memory table listing the indexes on a specified table, along with their descriptions.
DbiOpenIndexTypesList	Creates an in-memory table containing a list of all supported index types for the driver type.
DbiOpenLockList	Creates an in-memory table containing a list of locks acquired on the table.
DbiOpenRintList	Creates an in-memory table listing the referential integrity links for a specified table, along with their descriptions.
DbiOpenSecurityList	Creates an in-memory table listing record-level security information about a specified table.
DbiOpenTableList	Creates an in-memory table with information about all the tables accessible to the client application.
DbiOpenTableTypesList	Creates an in-memory table listing table type names for the given driver.

DbiOpenVchkList	Creates an in-memory table containing records with information about validity checks for fields within the specified table.

Date/time/number functions

Each BDE function listed below sets or retrieves date, time or number formats for the current session, or decodes or encodes date and time into or from a timestamp.

Function	Description
DbiBcdFromFloat	Converts FLOAT data to binary coded decimal (BCD) format.
DbiBcdToFloat	Converts binary coded decimal (BCD) data to FLOAT format.
DbiDateDecode	Decodes DBIDATE into separate month, day and year components.
DbiDateEncode	Encodes separate date components into date for use by DbiPutField and other functions.
DbiGetDateFormat	Gets the date format for the current session.
DbiGetNumberFormat	Gets the number format for the current session.
DbiGetTimeFormat	Gets the time format for the current session.
DbiSetDateFormat	Sets the date format for the current session.
DbiSetNumberFormat	Sets the number format for the current session.
DbiSetTimeFormat	Sets the time format for the current session.
DbiTimeDecode	Decodes time into separate components (hours, minutes, milliseconds).
DbiTimeEncode	Encodes separate time components into time for use by DbiPutField and other functions.
DbiTimeStampDecode	Extracts separate encoded date and time components from the timestamp.
DbiTimeStampEncode	Encodes the encoded date and encoded time into a timestamp.

Transaction functions

The BDE functions listed below begin, end, or return information about a transaction.

Function	Description
DbiBeginTran	Begins a transaction.
DbiEndTran	Ends a transaction.
DbiGetTranInfo	Retrieves the transaction state.

Function reference, alphabetical

Function	Description
DbiAcqPersistTableLock	Acquires an exclusive persistent lock on the table preventing other users from using the table or creating a table of the same name.
DbiAcqTableLock	Acquires a table-level lock on the table associated with the given cursor.
DbiActivateFilter	Activates a filter.
DbiAddAlias	Adds an alias to the BDE configuration file (IDAPI.CFG).
DbiAddDriver	Adds a driver to the BDE configuration file (IDAPI.CFG).
DbiAddFilter	Adds a filter to a table, but does not activate the filter (the record set is not yet altered).
DbiAddIndex	Creates an index on an existing table.
DbiAddPassword	Adds a password to the current session.
DbiAnsiToNative	Multipurpose translate function.
DbiAppendRecord	Appends a record to the end of the table associated with the given cursor.
DbiApplyDelayedUpdates	When cached updates cursor layer is active, writes all modifications made to cached data to the underlying database.
DbiBatchMove	Appends, updates, subtracts, and copies records or fields from a source table to a destination table.
DbiBcdFromFloat	Converts FLOAT data to binary coded decimal (BCD) format.
DbiBcdToFloat	Converts binary coded decimal (BCD) data to FLOAT format.
DbiBeginDelayedUpdates	Creates a cached updates cursor layer so that users can make extended changes to temporarily cached table data without writing to the actual table, thereby minimizing resource locking.
DbiBeginLinkMode	Converts a cursor to a link cursor. Given an open cursor, prepares for linked access. Returns a new cursor.
DbiBeginTran	Begins a transaction.
DbiCheckRefresh	Checks for remote updates to tables for all cursors in the current session, and refreshes the cursors if changed.
DbiCloneCursor	Creates a new cursor (clone cursor) which has the same result set as the given cursor (source cursor).
DbiCloseCursor	Closes a previously opened cursor.
DbiCloseDatabase	Closes a database and all tables associated with this database handle.
DbiCloseFieldXlt	Closes a field translation object.
DbiCloseIndex	Closes the specified index on a cursor.
DbiCloseSession	Closes the session associated with the given session handle.
DbiCompareBookMarks	Compares the relative positions of two bookmarks in the result set associated with the cursor.
DbiCompareKeys	Compares two key values based on the current index of the cursor.

DbiCopyTable Duplicates the specified source table to a destination table.

DbiCreateInMemTable Creates a temporary, in-memory table.

DbiCreateTable Creates a table.

DbiCreateTempTable Creates a temporary table that is deleted when the cursor is closed, unless the call is followed by a call to DbiMakePermanent.

DbiDateDecode Decodes DBIDATE into separate month, day and year components.

DbiDateEncode Encodes separate date components into date for use by DbiPutField and other functions.

DbiDeactivateFilter Temporarily stops the specified filter from affecting the record set by turning the filter off.

DbiDeleteAlias Deletes an alias from the BDE configuration file (IDAPI.CFG).

DbiDeleteDriver Deletes a driver from the BDE configuration file (IDAPI.CFG).

DbiDeleteIndex Drops an index on a table.

DbiDeleteRecord Deletes the current record of the given cursor.

DbiDeleteTable Deletes a table.

DbiDllExit Prepares the BDE to be disconnected within a Dll.

DbiDoRestructure Changes the properties of a table.

DbiDropFilter Deactivates and removes a filter from memory, and frees all resources.

DbiDropPassword Removes a password from the current session.

DbiEmptyTable Deletes all records from the table associated with the specified table cursor handle or table name.

DbiEndDelayedUpdates Closes a cached updates cursor layer ending the cached updates mode.

DbiEndLinkMode Ends linked cursor mode, and returns the original cursor.

DbiEndTran Ends a transaction.

DbiExit Disconnects the client application from BDE.

DbiExtractKey Retrieves the key value for the current record of the given cursor or from the supplied record buffer.

DbiForceRecordReread Rereads a single record from the server on demand, refreshing one row only, rather than clearing the cache.

DbiForceReread Refreshes all buffers associated with the cursor, if necessary.

DbiFormFullName Returns the fully qualified table name.

DbiFreeBlob Closes the BLOB handle located within the specified record buffer.

DbiGetBlob Retrieves data from the specified BLOB field.

DbiGetBlobHeading Retrieves information about a BLOB field from the BLOB heading in the record buffer.

DbiGetBlobSize Retrieves the size of the specified BLOB field in bytes.

DbiGetBookMark Saves the current position of a cursor to the client-supplied buffer called a bookmark.

DbiGetCallBack Returns a pointer to the function previously registered by the client for the given callback type.

DbiGetClientInfo	Retrieves system-level information about the client application environment.
DbiGetCurrSession	Returns the handle associated with the current session.
DbiGetCursorForTable	Finds the cursor for the given table.
DbiGetCursorProps	Returns the properties of the cursor.
DbiGetDatabaseDesc	Retrieves the description of the specified database from the configuration file.
DbiGetDateFormat	Gets the date format for the current session.
DbiGetDirectory	Retrieves the current working directory or the default directory.
DbiGetDriverDesc	Retrieves a description of a driver.
DbiGetErrorContext	After receiving an error code back from a call, enables the client to probe BDE for more specific error information.
DbiGetErrorEntry	Returns the error description of a specified error stack entry.
DbiGetErrorInfo	Provides descriptive error information about the last error that occurred.
DbiGetErrorString	Returns the message associated with a given error code.
DbiGetExactRecordCount	Retrieves the current exact number of records associated with the cursor.
DbiGetField	Retrieves the data contents of the requested field from the record buffer.
DbiGetFieldDescs	Retrieves a list of descriptors for all the fields in the table associated with the cursor.
DbiGetFieldTypeDesc	Retrieves a description of the specified field type.
DbiGetFilterInfo	Retrieves information about a specified filter.
DbiGetIndexDesc	Retrieves the properties of the given index associated with the cursor.
DbiGetIndexDescs	Retrieves index properties.
DbiGetIndexForField	Returns the description of any useful index on the specified field.
DbiGetIndexSeqNo	Retrieves the ordinal number of the index in the index list of the specified cursor.
DbiGetIndexTypeDesc	Retrieves a description of the index type.
DbiGetLdName	Retrieves the name of the language driver associated with the specified object name (table name).
DbiGetLdObj	Retrieves the language driver object associated with the given cursor.
DbiGetLinkStatus	Returns the link status of the cursor.
DbiGetNetUserName	Retrieves the user's network login name. User names should be available for all networks supported by Microsoft Windows.
DbiGetNextRecord	Retrieves the next record in the table associated with the cursor.
DbiGetNumberFormat	Gets the number format for the current session.
DbiGetObjFromName	Returns an object handle of the specified type or with the given name, if any.
DbiGetObjFromObj	Returns an object of the specified object type associated with or derived from a given object.

DbiGetPriorRecord	Retrieves the previous record in the table associated with the given cursor.
DbiGetProp	Returns a property of an object.
DbiGetRecord	Retrieves the current record, if any, in the table associated with the cursor.
DbiGetRecordCount	Retrieves the current number of records associated with the cursor.
DbiGetRecordForKey	Finds and retrieves a record matching a key and positions the cursor on that record.
DbiGetRelativeRecord	Positions the cursor on a record in the table relative to the current position of the cursor.
DbiGetRintDesc	Retrieves the referential integrity descriptor identified by the referential integrity sequence number and the cursor.
DbiGetSeqNo	Retrieves the sequence number of the current record in the table associated with the cursor.
DbiGetSesInfo	Retrieves the environment settings for the current session.
DbiGetSysConfig	Retrieves BDE system configuration information.
DbiGetSysInfo	Retrieves system status and information.
DbiGetSysVersion	Retrieves the system version information, including the BDE version number, date, and time, and the client interface version number.
DbiGetTableOpenCount	Returns the total number of cursors that are open on the specified table.
DbiGetTableTypeDesc	Returns a description of the capabilities of the table type for the driver type.
DbiGetTimeFormat	Gets the time format for the current session.
DbiGetTranInfo	Retrieves the transaction state.
DbiGetVchkDesc	Retrieves the validity check descriptor identified by the validity check sequence number and the cursor.
DbiInit	Initializes the BDE environment.
DbiInitRecord	Initializes the record buffer to a blank record according to the data types of the fields.
DbiInsertRecord	Inserts a new record into the table associated with the given cursor.
DbiIsRecordLocked	Checks if current record is locked.
DbiIsTableLocked	Returns the number of locks of a specified type acquired on the table associated with the given session.
DbiIsTableShared	Determines whether the table is physically shared or not.
DbiLinkDetail	Establishes a link between two tables such that the detail table has its record set limited to the set of records matching the linking key values of the master table cursor.
DbiLinkDetailToExp	Links the detail cursor to the master cursor using an expression.
DbiLoadDriver	Loads a given driver.
DbiMakePermanent	Changes a temporary table created by DbiCreateTempTable into a permanent table.

DbiModifyRecord	Modifies the current record of table associated with the cursor with the data supplied.
DbiNativeToAnsi	Translates a string in the native language driver to an ANSI string.
DbiOpenBlob	Prepares the cursor's record buffer to access a BLOB field.
DbiOpenCfgInfoList	Returns a handle to an in-memory table listing all the nodes in the configuration file accessible by the specified path.
DbiOpenDatabase	Opens a database in the current session and returns a database handle.
DbiOpenDatabaseList	Creates an in-memory table containing a list of accessible databases and their descriptions.
DbiOpenDriverList	Creates an in-memory table containing a list of driver names available to the client application.
DbiOpenFamilyList	Creates an in-memory table listing the family members associated with a specified table.
DbiOpenFieldList	Creates an in-memory table listing the fields in a specified table and their descriptions.
DbiOpenFieldTypesList	Creates an in-memory table containing a list of field types supported by the table type for the driver type.
DbiOpenFieldXlt	Builds a field translation object that can be used to translate a logical or physical field type into any other compatible logical or physical field type.
DbiOpenFileList	Opens a cursor on the virtual table containing all the tables accessible by the client application and their descriptions.
DbiOpenFunctionArgList	Returns a list of arguments to a data source function.
DbiOpenFunctionList	Returns a description of a data source function.
DbiOpenIndex	Opens the index for the table associated with the cursor.
DbiOpenIndexList	Opens a cursor on an in-memory table listing the indexes on a specified table, along with their descriptions.
DbiOpenIndexTypesList	Creates an in-memory table containing a list of all supported index types for the driver type.
DbiOpenLockList	Creates an in-memory table containing a list of locks acquired on the table associated with the cursor.
DbiOpenRintList	Creates an in-memory table listing the referential integrity links for a specified table, along with their descriptions.
DbiOpenSecurityList	Creates an in-memory table listing record-level security information about a specified table.
DbiOpenSPList	Creates a table containing information about the stored procedures associated with the database.
DbiOpenSPParamList	Creates a table listing the parameters associated with a specified stored procedure.
DbiOpenTable	Opens the given table for access and associates a cursor handle with the opened table.
DbiOpenTableList	Creates an in-memory table with information about all the tables accessible to the client application.
DbiOpenTableTypesList	Creates an in-memory table listing table type names for the

	given driver.
DbiOpenUserList	Creates an in-memory table containing a list of users sharing the same network file.
DbiOpenVchkList	Creates an in-memory table containing records with information about validity checks for fields within the specified table.
DbiPackTable	Optimizes table space by rebuilding the table associated with the cursor and releasing any free space.
DbiPutBlob	Writes data into an open BLOB field.
DbiPutField	Writes the field value to the correct location in the supplied record buffer.
DbiQAlloc	Allocates a new statement handle for a prepared query.
DbiQExec	Executes the previously prepared query identified by the supplied statement handle and returns a cursor to the result set, if one is generated.
DbiQExecDirect	Executes a SQL or QBE query and returns a cursor to the result set, if one is generated.
DbiQExecProcDirect	Executes a stored procedure and returns a cursor to the result set, if one is generated.
DbiQFree	Frees the resources associated with a previously prepared query identified by the supplied statement handle.
DbiQGetBaseDescs	Returns the original database, table, and field names of the fields that make up the result set of a query.
DbiQInstantiateAnswer	Creates a permanent table from a cursor handle.
DbiQPrepare	Prepares a SQL or QBE query for execution, and returns a handle to a statement containing the prepared query.
DbiQPrepareProc	Prepares and optionally binds parameters for a stored procedure.
DbiQSetParams	Associates data with parameter markers embedded within a prepared query.
DbiQSetProcParams	Binds parameters for a stored procedure prepared with DbiQPrepareProc.
DbiReadBlock	Reads a specified number of records (starting from the next position of the cursor) into a buffer.
DbiRegenIndex	Regenerates an index to make sure that it is up-to-date (all records currently in the table are included in the index and are in the index order).
DbiRegenIndexes	Regenerates all out-of-date indexes on a given table.
DbiRegisterCallBack	Registers a callback function for the client application.
DbiRelPersistTableLock	Releases the persistent table lock on the specified table.
DbiRelRecordLock	Releases the record lock on either the current record of the cursor or only the locks acquired in the current session.
DbiRelTableLock	Releases table locks of the specified type associated with the current session (the session in which the cursor was created).
DbiRenameTable	Renames the table and all of its resources to the new name specified.

DbiResetRange	Removes the specified table's limited range previously established by the function DbiSetRange.
DbiSaveChanges	Forces all updated records associated with the table to be written to disk.
DbiSetCurrSession	Sets the current session of the client application to the session associated with *hSes*.
DbiSetDateFormat	Sets the date format for the current session.
DbiSetDirectory	Sets the current directory for a standard database.
DbiSetFieldMap	Sets a field map of the table associated with the given cursor.
DbiSetLockRetry	Sets the table and record lock retry time for the current session.
DbiSetNumberFormat	Sets the number format for the current session.
DbiSetPrivateDir	Sets the private directory for the current session.
DbiSetProp	Sets the specified property of an object to a given value.
DbiSetRange	Sets a range on the result set associated with the cursor.
DbiSetTimeFormat	Sets the time format for the current session.
DbiSetToBegin	Positions the cursor to BOF (just before the first record).
DbiSetToBookMark	Positions the cursor to the location saved in the specified bookmark.
DbiSetToCursor	Sets the position of one cursor (the destination cursor) to that of another (the source cursor).
DbiSetToEnd	Positions the cursor to EOF (just after the last record).
DbiSetToKey	Positions an index-based cursor on a key value.
DbiSetToRecordNo	Positions the cursor of a dBASE or FoxPro table to the given physical record number.
DbiSetToSeqNo	Positions the cursor to the specified sequence number of a Paradox table.
DbiSortTable	Sorts an opened or closed table, either into itself or into a destination table. There are options to remove duplicates, to enable case-insensitive sorts and special sort functions, and to control the number of records sorted.
DbiStartSession	Starts a new session for the client application.
DbiSwitchToIndex	Allows the user to change the active index order of the given cursor.
DbiTimeDecode	Decodes time into separate components (hours, minutes, milliseconds).
DbiTimeEncode	Encodes separate time components into time for use by DbiPutField and other functions.
DbiTimeStampDecode	Extracts separate encoded date and time components from the timestamp.
DbiTimeStampEncode	Encodes the encoded date and encoded time into a timestamp.
DbiTranslateField	Translates a logical or physical field value to any compatible logical or physical field value.
DbiTranslateRecordStructure	Translates the source driver's physical or logical fields to equivalent physical or logical fields of the destination driver.

DbiTruncateBlob	Shortens the size of the contents of a BLOB field, or deletes the contents of a BLOB field from the record, by shortening it to zero.
DbiUndeleteRecord	Undeletes a dBASE or FoxPro record that has been marked for deletion (a "soft" delete).
DbiUnlinkDetail	Removes a link between two cursors.
DbiValidateProp	Validates a property.
DbiVerifyField	Verifies that the data specified is a valid data type for the field specified, and that all validity checks in place for the field are satisfied. It can also be used to check if a field is blank.
DbiWriteBlock	Writes a block of records to the table associated with the cursor.

DbiAcqPersistTableLock {button C Examples,JI(`>example',`exdbiacqpersisttablelock')} {button Delphi Examples,JI(`>example',`dexdbiacqpersisttablelock')}

C syntax

```
DBIResult DBIFN DbiAcqPersistTableLock (hDb, pszTableName, [pszDriverType]);
```

Delphi syntax

```
function DbiAcqPersistTableLock (hDb: hDBIDb; pszTableName: PChar;
  pszDriverType: PChar): DBIResult stdcall;
```

Description

DbiAcqPersistTableLock acquires an exclusive persistent lock on the table that prevents other users from using the table or creating a table of the same name.

Parameters

hDb Type: hDBIDb (Input)
Specifies the database handle.

pszTableName Type: pCHAR (Input)
Specifies the pointer to table name. For Paradox , if *pszTableName* is a fully qualified name of a table, the *pszDriverType* parameter need not be specified. If the path is not included, the path name is taken from the current directory of the database associated with *hDb*.

For SQL databases, this parameter can be a fully qualified name that includes the owner name.

pszDriverType Type: pCHAR (Input)
Specifies the pointer to the driver type. Optional. For Paradox, dBASE, and FoxPro tables, this parameter is required if *pszTableName* has no extension. This parameter is ignored if the database associated with *hDb* is a SQL database.

For Paradox tables, *pszDriverType* is required if the client application wants to overwrite the default file extension, including the situation where *pszTableName* is terminated with a period(.) *pszDriverType* must be szPARADOX.

If *pszTableName* does not supply the default extension, and *pszTableType* is NULL, DbiOpenTable tries to open the table with the default file extension of all file-based drivers listed in the configuration file in the order that the drivers are listed.

Usage

This function can be used to acquire an exclusive lock on a non-existent table as a way to reserve the table name. The function fails if the table is already in use. You cannot use DbiSetLockRetry to retry persistent locks.

dBASE and FoxPro: This function is not supported for dBASE and FoxPro tables.

Access: This function is not supported for Access tables.

SQL: This function depends on the capabilities of the server. Some servers provide non-blocking table locks; others provide blocking table locks only; others don't provide table locking. In no case is table locking truly persistent, however. If table locking is supported for the server but locks are not held across transactions, the lock is automatically reacquired after transaction commit. If the application requires a commit, it is responsible for insuring that the window of exposure between lock release and reacquisition has not impacted its consistency requirements. This function is provided to enable a degree of consistency with other drivers. It is recommended that transactions or transactions combined with explicit locking be used for SQL.

Prerequisites

The client application must have exclusive access to the table; if another user is accessing the table, the attempt to lock the table fails.

Completion state

The acquired persistent lock must be explicitly released by the client application. To release the lock, the client application that placed the lock must call DbiRelPersistTableLock.

DbiResult return values

DBIERR_NONE	The persistent lock was acquired successfully.
DBIERR_INVALIDHNDL	The specified database handle is invalid or NULL.
DBIERR_INVALIDPARAM	Either *pszTableName* or **pszTableName* is NULL.
DBIERR_INVALIDFILENAME	An invalid file name was specified by *pszTableName*.
DBIERR_NOSUCHTABLE	*pszTableName* is invalid.
DBIERR_UNKNOWNTBLTYPE	The driver type specified by *pszTableType* is invalid.
DBIERR_LOCKED	The table is already opened by another user, or another session.
DBIERR_NOTSUPPORTED	This function is not supported for dBASE and FoxPro tables.

See also

DbiOpenLockList

C Examples: DbiAcqPersistTableLock

Place a lock on a non-existent table:

```
DBIResult fDbiAcqPersistTableLock(hDBIDb hDb, char *TableName, char *Driver)
{
  DBIResult rslt;
  rslt = Chk(DbiAcqPersistTableLock(hDb, TableName, Driver));
  return rslt;
}
```

Delphi Examples: DbiAcqPersistTableLock

Place a lock on a non-existent table:

This example places and releases persistent lock on the TTable T. This example uses the following input:

```
  AcqAndRelPersistTableLock(Table1);
```

```
procedure AcqAndRelPersistTableLock(T: TTable);
var
  Drv: PChar;
begin
  with T do begin
    if (TableType = ttParadox) then
      Drv := StrNew(szParadox)
    else if (TableType = ttdBASE) then
      Drv := StrNew(szdBASE)
    else Drv := nil;
    try
      Check(DbiAcqPersistTableLock(DBHandle, PChar(TableName), Drv));
      Check(DbiRelPersistTableLock(DBHandle, PChar(TableName), Drv));
    finally
      if Assigned(Drv) then StrDispose(Drv);
    end;
  end;
end;
```

DbiAcqTableLock {button C Examples,JI(`>example',`exdbiacqtablelock')} {button Delphi Examples,JI(`>example',`dexdbiacqtablelock')}

C syntax

```
DBIResult DBIFN DbiAcqTableLock (hCursor, eLockType);
```

Delphi syntax

```
function DbiAcqTableLock (hCursor: hDBICur; eLockType: DBILockType):
  DBIResult stdcall;
```

Description

DbiAcqTableLock acquires a table-level lock on the table associated with the given cursor.

Parameters

hCursor Type: hDBICur (Input)
Specifies the cursor handle.

eLockType Type: DBILockType (Input)
Specifies the table lock type.

Usage

This function is used to prevent other users from updating a table. It can be used to ensure that the data read by the client application is the same data that is stored in the table at that specific moment.

This function is used to acquire a lock of higher precedence than the lock acquired when the cursor was opened. Locks acquired are owned by the session, not the cursor. If a lock cannot be obtained, an error is returned.

Redundant locks can be acquired on the table. For each lock acquired, a separate call to DbiRelTableLock is required to release it.

dBASE and FoxPro: If a READ lock is attempted, it is automatically upgraded to a WRITE lock.

Paradox: Both READ locks and WRITE locks can be acquired.

Access: Access tables are locked exclusively.

SQL: This function depends on the capabilities of the server. Some servers provide non-blocking table locks; others provide blocking table locks only; others don't provide table locking. If table locking is supported for the server but locks are not held across transactions, the lock is automatically reacquired after transaction commit. If the application requires a commit, it is responsible for insuring that the window of exposure between lock release and reacquisition has not impacted its consistency requirements. This function is provided to enable a degree of consistency with other drivers. It is recommended that transactions or transactions combined with explicit locking be used for SQL.

Completion state

Any cursor opened on a table can release locks placed by any cursor opened on that table within the same session. When the last cursor on the table is closed, the locks on the table are automatically released.

DbiResult return values

DBIERR_NONE The lock was acquired successfully.

DBIERR_INVALIDHNDL The specified cursor handle is invalid or NULL.

DBIERR_LOCKED The requested lock is not available.

DBIERR_TBLLOCKLIMIT	The lock limit has been reached.

See also

DbiRelTableLock, DbiIsTableLocked, DbiOpenLockList, DbiAcqPersistTableLock, DbiOpenTable

eLockType

eLockType can be one of the following values:

Lock Type	Description
dbiWRITELOCK	When a write lock is placed, it prevents other sessions from placing any locks. For SQL tables, a write lock is the same as a read lock; behavior varies according to the server.
dbiREADLOCK	When a read lock is placed, it prevents other users from placing a write lock. For dBASE and FoxPro tables, a read lock is automatically upgraded to a write lock. For SQL tables, a write lock is the same as a read lock; behavior varies according to the server.

Note: Exclusive locks and NO locks are not considered acquired table locks. They are achieved with the DbiOpenTable function, and are owned by the cursor, rather than the session.

Note: Persistent locks are acquired table locks for Paradox and SQL tables only; acquired by the DbiAcqPersistTableLock function.

C Examples: DbiAcqTableLock

Place a write lock on an existing table:

```
DBIResult fDbiAcqTableLock(hDBICur hTmpCur)
{
  DBIResult rslt;
  rslt = Chk(DbiAcqTableLock(hTmpCur, dbiWRITELOCK));
  return rslt;
}
```

Delphi Examples: DbiAcqTableLock

Place a write lock on an existing table:

Delphi users should use the method TTable.LockTable rather than directly calling DbiAcqTableLock. This method is defined as:

```
  procedure TTable.LockTable(LockType: TLockType);
```

The following code places a write lock on a TTable object called Table1:

```
  Table1.LockTable(ltWriteLock);
```

Place a write lock on the specified cursor's table:

Delphi users should use TTable.LockTable: This example uses the following input:

```
  fDbiAcqTableLock(Table1.Handle);
```

```
procedure fDbiAcqTableLock(hTmpCur: hDBICur);
begin
  Check(DbiAcqTableLock(hTmpCur, dbiWRITELOCK));
end;
```

DbiActivateFilter {button C Examples,JI(`>example',`exdbiactivatefilter')} {button Delphi Examples,JI(`>example',`dexdbiaddfilter')}

C syntax
```
DBIResult DBIFN DbiActivateFilter (hCursor, [hFilter]);
```

Delphi syntax
```
function DbiActivateFilter (hCursor: hDBICur; hFilter: hDBIFilter):
  DBIResult stdcall;
```

Description
DbiActivateFilter activates a filter.

Parameters
hCursor Type: hDBICur (Input)
Specifies the cursor handle of the cursor for which the filter is to be activated.

hFilter Type: hDBIFilter (Input)
Specifies the filter handle of the filter to be activated.

Usage
A single cursor can have many filters associated with it. If the filter handle is NULL, all filters for this cursor are activated. See DbiAddFilter for a detailed explanation of filters.

Prerequisites
The filter must have been successfully added with DbiAddFilter, which returns the filter handle.

Completion state
Once the filter is activated, the filter controls the record set and all operations for that cursor are affected. Only those records which meet the criteria defined by the filter will be retrieved. For example, moving to the next record moves the cursor to the next record that passes the filter criteria, not to the next sequential record. The filter provides a restricted view of live data.

DbiResult return values
DBIERR_NONE	The filter was activated successfully.
DBIERR_INVALIDHNDL	The specified cursor handle is invalid or NULL.
DBIERR_NOSUCHFILTER	The specified filter handle is invalid.

See also
DbiAddFilter, DbiDeactivateFilter, DbiDropFilter

C Examples: DbiActivateFilter

Limiting the records which are available in the table using a filter.

This example limits records to those with a CUST_NO field greater than 1500.

```
static const char szTblName[] = "cust";      // Name of the table
  static const char szTblType[] = szDBASE;    // Type of table
  static const char szField[]   = "CUST_NO"; // Name of the field for the
                                              //   third node of the tree.
  static const DFLOAT fConst    = 1500.0;     // Value of the constant for
                                              //   the second node of the
  tree.

  void
  Filter (void)
  {
      hDBIDb          hDb = 0;            // Handle to the database.
      hDBICur         hCur = 0;           // Handle to the table.
      DBIResult       rslt;               // Return value from IDAPI
  functions.
      pBYTE           pcanExpr;           // Structure containing filter
  info.
      hDBIFilter      hFilter;            // Filter handle.
      UINT16          uSizeNodes;         // Size of the nodes in the tree.
      UINT16          uSizeCanExpr;       // Size of the header information.
      UINT32          uSizeLiterals;      // Size of the literals.
      UINT32          uTotalSize;         // Total size of the filter
  expression.
      UINT32          uNumRecs = 10;      // Number of records to display.
      CANExpr         canExp;             // Contains the header information.
      struct {
          CANBinary BinaryNode;
          CANField  FieldNode;
          CANConst  ConstantNode;
      }
      Nodes = {                           // Nodes of the filter tree.
      {
          // Offset 0
          nodeBINARY,                     // canBinary.nodeClass
          canGT,                          // canBinary.canOp
          sizeof(Nodes.BinaryNode),       // canBinary.iOperand1
          sizeof(Nodes.BinaryNode) + sizeof(Nodes.FieldNode),
                                          // canBinary.iOperand2
                                          //   Offsets in the Nodes array
      },
      {
          // Offset sizeof(Nodes.BinaryNode)
          nodeFIELD,                      // canField.nodeClass
          canFIELD,                       // canField.canOp
          1,                              // canField.iFieldNum
          0,                              // canField.iNameOffset: szField is
  the
                                          //   literal at offset 0
      },
      {
          // Offset sizeof(Nodes.BinaryNode) + sizeof(Nodes.FieldNode)
          nodeCONST,                      // canConst.nodeClass
```

```
        canCONST,                       // canConst.canOp
        fldFLOAT,                       // canConst.iType
        sizeof(fConst),                 // canConst.iSize
        8,                              // canConst.iOffset: fconst is the
                                        // literal at offset
strlen(szField) + 1
    }};

    Screen("*** Filter Example ***\r\n");

    BREAK_IN_DEBUGGER();

    Screen("    Initializing IDAPI...");

    if (InitAndConnect(&hDb) != DBIERR_NONE)
    {
        Screen("\r\n*** End of Example ***");
        return;
    }

    Screen("    Setting the database directory...");
    rslt = DbiSetDirectory(hDb, (pCHAR)szTblDirectory);
    ChkRslt(rslt, "SetDirectory");

    Screen("    Open the %s table...", szTblName);
    rslt = DbiOpenTable(hDb, (pCHAR)szTblName, (pCHAR)szTblType, NULL,
NULL, 0,
                        dbiREADWRITE, dbiOPENSHARED, xltFIELD, FALSE, NULL,
                        &hCur);
    if (ChkRslt(rslt, "OpenTable") != DBIERR_NONE)
    {
        CloseDbAndExit(&hDb);
        Screen("\r\n*** End of Example ***");
        return;
    }

    // Go to the beginning of the table
    rslt = DbiSetToBegin(hCur);
    ChkRslt(rslt, "SetToBegin");

    Screen("\r\n    Display the %s table...", szTblName);
    DisplayTable(hCur, uNumRecs);

    uSizeNodes      = sizeof(Nodes);    // Size of the nodes.
    uSizeLiterals   = (UINT16)(strlen((pCHAR)szField) + 1 +
sizeof(fConst));
                                        // Size of the literals.
    uSizeCanExpr    = sizeof(CANExpr);  // Size of the header information.
    uTotalSize      = (UINT16)(uSizeCanExpr + uSizeNodes + uSizeLiterals);
                                        // Total size of the filter.

    // Initialize the header information
    canExp.iVer = 1;                    // Version.
    canExp.iTotalSize = (UINT16)uTotalSize; // Total size of the filter.
```

```
    canExp.iNodes = 3;                      // Number of nodes.
    canExp.iNodeStart = uSizeCanExpr;   // The offset in the buffer where
the
                                        //   expression nodes start.
    canExp.iLiteralStart = (UINT16)(uSizeCanExpr + uSizeNodes);
                                        // The offset in the buffer where
the
                                        //   literals start.

    // Allocate contiguous memory space to hold
    // 1) Header information  i.e. the CANExpr structure
    // 2) Binary, field and constant nodes  i.e. the Nodes structure
    // 3) Literal and constant pool  i.e. field names and constant values
    pcanExpr = (pBYTE)malloc(uTotalSize * sizeof(BYTE));
    if (pcanExpr == NULL)
    {
        Screen("    Could not allocate memory...");
        DbiCloseCursor(&hCur);
        CloseDbAndExit(&hDb);
        Screen("\r\n*** End of Example ***");
        return;
    }

    // Initialize the filter expression by placing header, nodes and
    // pool into pcanexpr

    // Move header information into pcanexpr. pcanExpr will now look as
follows:
    // **canExp**|                 |                         |
    // | CANExpr |    All Nodes    | Literal, Constant Pool |
    // |-------------------------------------------------|
    // 0
sizeof(uTotalSize)
    memmove(pcanExpr, &canExp, uSizeCanExpr);

    // Move node structure into pcanexpr. pcanExpr will now look as
follows:
    // |**canExp*|**Node Structure*|                         |
    // | CANExpr |    All Nodes    | Literal, Constant Pool |
    // |-------------------------------------------------|
    // 0
sizeof(uTotalSize)
    memmove(&pcanExpr[uSizeCanExpr], &Nodes, uSizeNodes);

    // Move the literal into pcanexpr. pcanExpr will now look as follows:
    // |**canExp*|**Node Structure*|***szField              |
    // | CANExpr |    All Nodes    | Literal, Constant Pool |
    // |-------------------------------------------------|
    // 0
sizeof(uTotalSize)
    memmove(&pcanExpr[uSizeCanExpr + uSizeNodes],
            szField, strlen(szField) + 1);  // First literal "CUST_NO"

    // Move the literal into pcanexpr. pcanExpr will now look as follows:
    // |**canExp*|**Node Structure*|***szField*****fConst***|
```

```
    // | CANExpr |    All Nodes    | Literal, Constant Pool |
    // |-------------------------------------------------|
    // 0
sizeof(uTotalSize)
    memmove(&pcanExpr[uSizeCanExpr + uSizeNodes + strlen(szField) + 1],
            &fConst, sizeof(fConst));       // Second literal 1500.00

    rslt = DbiSetToBegin(hCur);
    ChkRslt(rslt, "SetToBegin");

    Screen("\r\n    Add a filter to the %s table which will limit"
           " the records\r\n        which are displayed to those whose"
           " %s field is greater than %.1lf...", szTblName, szField,
fConst);
    rslt = DbiAddFilter(hCur, 0L, 1, FALSE, (pCANExpr)pcanExpr, NULL,
                        &hFilter);
    if (ChkRslt(rslt, "AddFilter") != DBIERR_NONE)
    {
        rslt = DbiCloseCursor(&hCur);
        ChkRslt(rslt, "CloseCursor");
        free(pcanExpr);
        CloseDbAndExit(&hDb);
        Screen("\r\n*** End of Example ***");
        return;
    }

    // Activate the filter.
    Screen("    Activate the filter on the %s table...", szTblName);
    rslt = DbiActivateFilter(hCur, hFilter);
    ChkRslt(rslt, "ActivateFilter");

    rslt = DbiSetToBegin(hCur);
    ChkRslt(rslt, "SetToBegin");

    Screen("\r\n    Display the %s table with the filter set...",
szTblName);
    DisplayTable(hCur, uNumRecs);

    Screen("\r\n    Deactivate the filter...");
    rslt = DbiDeactivateFilter(hCur, hFilter);
    ChkRslt(rslt, "DeactivateFilter");

    Screen("\r\n    Drop the filter...");
    rslt = DbiDropFilter(hCur, hFilter);
    ChkRslt(rslt, "DropFilter");

    rslt = DbiCloseCursor(&hCur);
    ChkRslt(rslt, "CloseCursor");

    free(pcanExpr);

    Screen("    Close the database and exit IDAPI...");
    CloseDbAndExit(&hDb);
```

```
    Screen("\r\n*** End of Example ***");
}
```

Limiting the records which are available in the table using a comparison filter.

This example shows how to use filters to limit the result set of a table. This example does a comparison of the first N digits of a character field.

```
static const char szTblName[] = "cust";     // Name of the table
  static const char szTblType[] = szDBASE;    // Type of table
  static const char szField[]   = "CUST_NO"; // Name of the field for the
                                              //   third node of the tree.
  static const DFLOAT fConst    = 1500.0;     // Value of the constant for
                                              //   the second node of the
  tree.

  void
  Filter (void)
  {
      hDBIDb          hDb = 0;            // Handle to the database.
      hDBICur         hCur = 0;           // Handle to the table.
      DBIResult       rslt;               // Return value from IDAPI
  functions.
      pBYTE           pcanExpr;           // Structure containing filter
  info.
      hDBIFilter      hFilter;            // Filter handle.
      UINT16          uSizeNodes;         // Size of the nodes in the tree.
      UINT16          uSizeCanExpr;       // Size of the header information.
      UINT32          uSizeLiterals;      // Size of the literals.
      UINT32          uTotalSize;         // Total size of the filter
  expression.
      UINT32          uNumRecs = 10;      // Number of records to display.
      CANExpr         canExp;             // Contains the header information.
      struct {
          CANBinary BinaryNode;
          CANField  FieldNode;
          CANConst  ConstantNode;
      }
      Nodes = {                           // Nodes of the filter tree.
      {
          // Offset 0
          nodeBINARY,                     // canBinary.nodeClass
          canGT,                          // canBinary.canOp
          sizeof(Nodes.BinaryNode),       // canBinary.iOperand1
          sizeof(Nodes.BinaryNode) + sizeof(Nodes.FieldNode),
                                          // canBinary.iOperand2
                                          //   Offsets in the Nodes array
      },
      {
          // Offset sizeof(Nodes.BinaryNode)
          nodeFIELD,                      // canField.nodeClass
          canFIELD,                       // canField.canOp
          1,                              // canField.iFieldNum
          0,                              // canField.iNameOffset: szField is
  the
                                          //   literal at offset 0
      },
      {
```

```
        // Offset sizeof(Nodes.BinaryNode) + sizeof(Nodes.FieldNode)
        nodeCONST,                      // canConst.nodeClass
        canCONST,                       // canConst.canOp
        fldFLOAT,                       // canConst.iType
        sizeof(fConst),                 // canConst.iSize
        8,                              // canConst.iOffset: fconst is the
                                        // literal at offset
strlen(szField) + 1
    }};

    Screen("*** Filter Example ***\r\n");

    BREAK_IN_DEBUGGER();

    Screen("    Initializing IDAPI...");

    if (InitAndConnect(&hDb) != DBIERR_NONE)
    {
        Screen("\r\n*** End of Example ***");
        return;
    }

    Screen("    Setting the database directory...");
    rslt = DbiSetDirectory(hDb, (pCHAR)szTblDirectory);
    ChkRslt(rslt, "SetDirectory");

    Screen("    Open the %s table...", szTblName);
    rslt = DbiOpenTable(hDb, (pCHAR)szTblName, (pCHAR)szTblType, NULL,
NULL, 0,
                        dbiREADWRITE, dbiOPENSHARED, xltFIELD, FALSE, NULL,
                        &hCur);
    if (ChkRslt(rslt, "OpenTable") != DBIERR_NONE)
    {
        CloseDbAndExit(&hDb);
        Screen("\r\n*** End of Example ***");
        return;
    }

    // Go to the beginning of the table
    rslt = DbiSetToBegin(hCur);
    ChkRslt(rslt, "SetToBegin");

    Screen("\r\n    Display the %s table...", szTblName);
    DisplayTable(hCur, uNumRecs);

    uSizeNodes      = sizeof(Nodes);    // Size of the nodes.
    uSizeLiterals   = (UINT16)(strlen((pCHAR)szField) + 1 +
sizeof(fConst));
                                        // Size of the literals.
    uSizeCanExpr    = sizeof(CANExpr);  // Size of the header information.
    uTotalSize      = (UINT16)(uSizeCanExpr + uSizeNodes + uSizeLiterals);
                                        // Total size of the filter.

    // Initialize the header information
```

```
    canExp.iVer = 1;                    // Version.
    canExp.iTotalSize = (UINT16)uTotalSize; // Total size of the filter.
    canExp.iNodes = 3;                      // Number of nodes.
    canExp.iNodeStart = uSizeCanExpr;   // The offset in the buffer where
the
                                        //   expression nodes start.
    canExp.iLiteralStart = (UINT16)(uSizeCanExpr + uSizeNodes);
                                        // The offset in the buffer where
the
                                        //   literals start.

    // Allocate contiguous memory space to hold
    // 1) Header information  i.e. the CANExpr structure
    // 2) Binary, field and constant nodes  i.e. the Nodes structure
    // 3) Literal and constant pool  i.e. field names and constant values
    pcanExpr = (pBYTE)malloc(uTotalSize * sizeof(BYTE));
    if (pcanExpr == NULL)
    {
        Screen("    Could not allocate memory...");
        DbiCloseCursor(&hCur);
        CloseDbAndExit(&hDb);
        Screen("\r\n*** End of Example ***");
        return;
    }

    // Initialize the filter expression by placing header, nodes and
    // pool into pcanexpr

    // Move header information into pcanexpr. pcanExpr will now look as
follows:
    // **canExp**|                |                         |
    // | CANExpr |    All Nodes    | Literal, Constant Pool |
    // |-------------------------------------------------|
    // 0
sizeof(uTotalSize)
    memmove(pcanExpr, &canExp, uSizeCanExpr);

    // Move node structure into pcanexpr. pcanExpr will now look as
follows:
    // |**canExp*|**Node Structure*|                       |
    // | CANExpr |    All Nodes    | Literal, Constant Pool |
    // |-------------------------------------------------|
    // 0
sizeof(uTotalSize)
    memmove(&pcanExpr[uSizeCanExpr], &Nodes, uSizeNodes);

    // Move the literal into pcanexpr. pcanExpr will now look as follows:
    // |**canExp*|**Node Structure*|***szField             |
    // | CANExpr |    All Nodes    | Literal, Constant Pool |
    // |-------------------------------------------------|
    // 0
sizeof(uTotalSize)
    memmove(&pcanExpr[uSizeCanExpr + uSizeNodes],
            szField, strlen(szField) + 1);  // First literal "CUST_NO"
```

```
    // Move the literal into pcanexpr. pcanExpr will now look as follows:
    // |**canExp*|**Node Structure*|***szField*****fConst***|
    // | CANExpr |    All Nodes    | Literal, Constant Pool |
    // |-------------------------------------------------|
    // 0
sizeof(uTotalSize)
    memmove(&pcanExpr[uSizeCanExpr + uSizeNodes + strlen(szField) + 1],
            &fConst, sizeof(fConst));       // Second literal 1500.00

    rslt = DbiSetToBegin(hCur);
    ChkRslt(rslt, "SetToBegin");

    Screen("\r\n    Add a filter to the %s table which will limit"
           " the records\r\n        which are displayed to those whose"
           " %s field is greater than %.1lf...", szTblName, szField,
fConst);
    rslt = DbiAddFilter(hCur, 0L, 1, FALSE, (pCANExpr)pcanExpr, NULL,
                        &hFilter);
    if (ChkRslt(rslt, "AddFilter") != DBIERR_NONE)
    {
        rslt = DbiCloseCursor(&hCur);
        ChkRslt(rslt, "CloseCursor");
        free(pcanExpr);
        CloseDbAndExit(&hDb);
        Screen("\r\n*** End of Example ***");
        return;
    }

    // Activate the filter.
    Screen("    Activate the filter on the %s table...", szTblName);
    rslt = DbiActivateFilter(hCur, hFilter);
    ChkRslt(rslt, "ActivateFilter");

    rslt = DbiSetToBegin(hCur);
    ChkRslt(rslt, "SetToBegin");

    Screen("\r\n    Display the %s table with the filter set...",
szTblName);
    DisplayTable(hCur, uNumRecs);

    Screen("\r\n    Deactivate the filter...");
    rslt = DbiDeactivateFilter(hCur, hFilter);
    ChkRslt(rslt, "DeactivateFilter");

    Screen("\r\n    Drop the filter...");
    rslt = DbiDropFilter(hCur, hFilter);
    ChkRslt(rslt, "DropFilter");

    rslt = DbiCloseCursor(&hCur);
    ChkRslt(rslt, "CloseCursor");

    free(pcanExpr);

    Screen("    Close the database and exit IDAPI...");
```

```
    CloseDbAndExit(&hDb);

    Screen("\r\n*** End of Example ***");
}
```

Limiting the records which are available in the table using a continue filter.

This example shows how to use filters to limit the result set of a table. This example shows the use of a canContinue node. A Continue node is used to stop evaluating when a certain condition is false for the first time. This filter will limit the result set to those customers living in Hawaii, you are listed in the table before the customer with ID 1624.

```
static const char szTblName[] = "customer";
  static const char szTblType[] = szPARADOX;
  static const CHAR szField1[] = "Customer No";
  static const CHAR szField2[] = "State/Prov";
  static const CHAR szConst[]  = "HI";
  static const DFLOAT fConst = 1624.0;

  void
  FiltCont (void)
  {
      hDBIDb          hDb = 0;        // Handle to the database.
      hDBICur         hCur = 0;       // Handle to the table.
      DBIResult       rslt;           // Return value from IDAPI functions.
      pBYTE           pcanExpr;       // Structure containing
                                      //   filter info.
      hDBIFilter      hFilter;        // Filter handle.
      UINT16          uSizeNodes;     // Size of the nodes in the
                                      //   tree.
      UINT16          uSizeCanExpr;   // Size of the header
                                      //   information.
      UINT32          uSizeLiterals;  // Size of the literals.
      UINT32          uTotalSize;     // Total size of the filter
                                      //   expression.
      UINT32          uNumRecs  = 10; // Number of records to
                                      //   display.
      CANExpr         canExp;         // Contains the header
                                      //   information.

      struct {
         CANBinary BinaryNode1;
         CANBinary  BinaryNode2;
         CANField  FieldNode1;
         CANConst  ConstantNode1;
         CANUnary UnaryNode1;
         CANBinary  BinaryNode3;
         CANField  FieldNode2;
         CANConst  ConstantNode2;
      } Nodes = {                     // Nodes of the filter tree.
      {
          // Offset 0. Node 1. sizeof(Nodes.BinaryNode1)
          nodeBINARY,                 // canBinary.nodeClass
          canAND,                     // canBinary.canOp
          sizeof(Nodes.BinaryNode1),  // canBinary.iOperand1 - node 2
```

```
    sizeof(Nodes.BinaryNode1) + sizeof(Nodes.BinaryNode2) +
    sizeof(Nodes.FieldNode1) + sizeof(Nodes.ConstantNode1),
                                // canBinary.iOperand2 - node 5
},
{
    // Offset 8. Node 2. sizeof(Nodes.BinaryNode2)
    nodeBINARY,                 // canBinary.nodeClass
    canEQ ,                     // canBinary.canOp
    sizeof(Nodes.BinaryNode1) + sizeof(Nodes.BinaryNode2),
                                // canBinary.iOperand1 - node 3

    sizeof(Nodes.BinaryNode1) + sizeof(Nodes.BinaryNode2) +
    sizeof(Nodes.FieldNode1),   // canBinary.iOperand2 - node 4
},
{
    // Offset 16. Node 3. sizeof(Nodes.FieldNode1)
    nodeFIELD,                  // canField.nodeClass
    canFIELD,                   // canField.canOp
    5,                          // canField.iFieldNum
    12,                         // canField.iNameOffset: szField2
},                              //   is the literal at offset
{                               //   strlen(szField1) + 1

    // Offset 24. Node 4. sizeof(Nodes.ConstantNode1)
    nodeCONST,                  // canConst.nodeClass
    canCONST,                   // canConst.canOp
    fldZSTRING,                 // canConst.iType
    3,                          // canConst.iSize
    31,                         // canConst.iOffset: fconst is
},                              //   the literal at offset
{                               //   strlen(szField1) + 1 +
                                //   sizeof(fConst) +
                                //   strlen(szField2) + 1

    // Offset 34. Node 5. sizeof(Nodes.UnaryNode1)
    nodeUNARY,      // canBinary.nodeClass
    canCONTINUE,    // canBinary.canOp
    sizeof(Nodes.BinaryNode1) + sizeof(Nodes.BinaryNode2) +

    sizeof(Nodes.FieldNode1) + sizeof(Nodes.ConstantNode1) +
    sizeof(Nodes.UnaryNode1),   // canBinary.iOperand1 - node 6
},                              //   Offsets in the Nodes array
{
    // Offset 40. Node 6. sizeof(Nodes.BinaryNode3)
    nodeBINARY,     // canBinary.nodeClass
    canNE ,         // canBinary.canOp
    sizeof(Nodes.BinaryNode1) + sizeof(Nodes.BinaryNode2) +
    sizeof(Nodes.FieldNode1) + sizeof(Nodes.ConstantNode1) +
    sizeof(Nodes.UnaryNode1) + sizeof(Nodes.BinaryNode3),
                                // canBinary.iOperand1 - node 7

    sizeof(Nodes.BinaryNode1) + sizeof(Nodes.BinaryNode2) +
    sizeof(Nodes.FieldNode1) + sizeof(Nodes.ConstantNode1) +
    sizeof(Nodes.UnaryNode1) + sizeof(Nodes.BinaryNode3) +
    sizeof(Nodes.FieldNode2),   // canBinary.iOperand2 - node 8
```

```
    },
    {
        // Offset 48. Node 7. sizeof(Nodes.FieldNode2)
        nodeFIELD,                  // canField.nodeClass
        canFIELD,                   // canField.canOp
        1,                          // canField.iFieldNum
        0,                          // canField.iNameOffset:
    },                              //   szField1 is the literal at
    {                               //   offset 0.

        // Offset 56. Node 8. Size 14 Bytes
        nodeCONST,                  // canConst.nodeClass
        canCONST,                   // canConst.canOp
        fldFLOAT,                   // canConst.iType
        sizeof(fConst),             // canConst.iSize
        23,                         // canConst.iOffset: fconst is
    }};                             //   the literal at offset
                                    //   strlen(szField1) + 1 +
                                    //   strlen(szField2) + 1

    Screen("*** Continue Filter Example ***\r\n");

    BREAK_IN_DEBUGGER();

    Screen("    Initializing IDAPI...");
    if (InitAndConnect(&hDb) != DBIERR_NONE)
    {
        Screen("\r\n*** End of Example ***");
        return;
    }

    Screen("    Setting the database directory...");
    rslt = DbiSetDirectory(hDb, (pCHAR)szTblDirectory);
    ChkRslt(rslt, "SetDirectory");

    Screen("    Open the %s table...", szTblName);
    rslt = DbiOpenTable(hDb, (pCHAR)szTblName, (pCHAR)szTblType, NULL,
NULL, 0,
                        dbiREADWRITE, dbiOPENSHARED, xltFIELD, FALSE, NULL,
                        &hCur);
    if (ChkRslt(rslt, "OpenTable") != DBIERR_NONE)
    {
        CloseDbAndExit(&hDb);
        Screen("\r\n*** End of Example ***");
        return;
    }

    // Go to the beginning of the table
    rslt = DbiSetToBegin(hCur);
    ChkRslt(rslt, "SetToBegin");

    Screen("\r\n    Display the %s table...", szTblName);
    DisplayTable(hCur, uNumRecs);
```

```
    // Size of the nodes.
    uSizeNodes      = sizeof(Nodes);
    // Size of the literals.
    uSizeLiterals   = strlen(szField1) + 1 + sizeof(fConst) +
                      strlen(szField2) + 1 + strlen(szConst)
                      + 1;
    // Size of the header information.
    uSizeCanExpr    = sizeof(CANExpr);
    // Total size of the filter.
    uTotalSize      = uSizeCanExpr + uSizeNodes + uSizeLiterals;
    // Initialize the header information
    canExp.iVer = 1; // Version.
    canExp.iTotalSize = (UINT16)uTotalSize; // Total size of the filter.
    canExp.iNodes = 8; // Number of nodes.
    canExp.iNodeStart = uSizeCanExpr; // The offset in the
                                      //   buffer where the
                                      //   expression nodes
                                      //   start.
    // The offset in the buffer where the literals start.
    canExp.iLiteralStart = (UINT16)(uSizeCanExpr + uSizeNodes);

    // Allocate contiguous memory space to hold
    // 1) Header information  i.e. the CANExpr structure
    // 2) Compare, field and constant nodes  i.e. the Nodes structure
    // 3) Literal and constant pool  i.e. field names and constant values
    pcanExpr = (pBYTE)malloc(uTotalSize * sizeof(BYTE));
    if (pcanExpr == NULL)
    {
        Screen("    Could not allocate memory...");
        DbiCloseCursor(&hCur);
        CloseDbAndExit(&hDb);
        Screen("\r\n*** End of Example ***");
        return;
    }

    // Initialize the filter expression by placing header, nodes and
    // pool into pcanexpr

    // Move the literal into pcanexpr. pcanExpr will now look as follows:
    // |**canExp*|                |                                  |
    // | CANExpr |    All Nodes    |      Literal, Constant Pool      |
    // |-----------------------------------------------------------|
    // 0
sizeof(uTotalSize)
    memmove(pcanExpr, &canExp, uSizeCanExpr); // Insert Header Information

    // Move the literal into pcanexpr. pcanExpr will now look as follows:
    // |**canExp*|**Node Structure*|                               |
    // | CANExpr |    All Nodes    |      Literal, Constant Pool      |
    // |-----------------------------------------------------------|
    // 0
sizeof(uTotalSize)
    memmove(&pcanExpr[uSizeCanExpr], &Nodes, uSizeNodes); // Insert Nodes

    // Move the literal into pcanexpr. pcanExpr will now look as follows:
```

```
    // |**canExp*|**Node Structure*|*szField1*                        |
    // | CANExpr |    All Nodes    |      Literal, Constant Pool      |
    // |-----------------------------------------------------------|
    // 0
sizeof(uTotalSize)
    memmove(&pcanExpr[uSizeCanExpr + uSizeNodes],
            szField1, strlen(szField1) + 1); // First litteral

    // Move the literal into pcanexpr. pcanExpr will now look as follows:
    // |**canExp*|**Node Structure*|*szField1*szField2*               |
    // | CANExpr |    All Nodes    |      Literal, Constant Pool      |
    // |-----------------------------------------------------------|
    // 0
sizeof(uTotalSize)
    memmove(&pcanExpr[(uSizeCanExpr + uSizeNodes +
                       strlen(szField1) + 1)],
            szField2, strlen(szField2) + 1); // Second litteral

    // Move the literal into pcanexpr. pcanExpr will now look as follows:
    // |**canExp*|**Node Structure*|*szField1*szField2*fConst*        |
    // | CANExpr |    All Nodes    |      Literal, Constant Pool      |
    // |-----------------------------------------------------------|
    // 0
sizeof(uTotalSize)
    memmove(&pcanExpr[(uSizeCanExpr + uSizeNodes +
                       strlen(szField1) + 1 +
                       strlen(szField2) + 1)],
            &fConst, sizeof(fConst)); // First Constant

    // Move the literal into pcanexpr. pcanExpr will now look as follows:
    // |**canExp*|**Node Structure*|*szField1*szField2*fConst*szConst*|
    // | CANExpr |    All Nodes    |      Literal, Constant Pool      |
    // |-----------------------------------------------------------|
    // 0
sizeof(uTotalSize)
    memmove(&pcanExpr[(uSizeCanExpr + uSizeNodes +
                       strlen(szField1) + 1 +
                       strlen(szField2) + 1 + sizeof(fConst))],
            szConst, strlen(szConst) + 1); // Second Constant

    rslt = DbiSetToBegin(hCur);
    ChkRslt(rslt, "SetToBegin");

    Screen("\r\n    Add a filter to the %s table which will"
           " limit the records\r\n        in the result set"
           " to those whose %s field is equal to '%s', until\r\n"
           "        the first record where the %s field is equal to
%.1lf...",
           szTblName, szField2, szConst, szField1, fConst);
    rslt = DbiAddFilter(hCur, 0L, 1, FALSE, (pCANExpr)pcanExpr,
                        NULL, &hFilter);
    if (ChkRslt(rslt, "AddFilter") != DBIERR_NONE)
    {
        rslt = DbiCloseCursor(&hCur);
        ChkRslt(rslt, "CloseCursor");
```

```
        free(pcanExpr);
        CloseDbAndExit(&hDb);
        Screen("\r\n*** End of Example ***");
        return;
    }

    // Activate the filter.
    Screen("    Activate the filter on the %s table...",
           szTblName);
    rslt = DbiActivateFilter(hCur, hFilter);
    ChkRslt(rslt, "ActivateFilter");

    rslt = DbiSetToBegin(hCur);
    ChkRslt(rslt, "SetToBegin");

    Screen("\r\n    Display the %s table with the filter"
           " set...", szTblName);
    DisplayTable(hCur, uNumRecs);

    Screen("\r\n    Deactivate the filter...");
    rslt = DbiDeactivateFilter(hCur, hFilter);
    ChkRslt(rslt, "DeactivateFilter");

    Screen("\r\n    Drop the filter...");
    rslt = DbiDropFilter(hCur, hFilter);
    ChkRslt(rslt, "DropFilter");

    rslt = DbiCloseCursor(&hCur);
    ChkRslt(rslt, "CloseCursor");

    free(pcanExpr);

    Screen("    Close the database and exit IDAPI...");
    CloseDbAndExit(&hDb);

    Screen("\r\n*** End of Example ***");
}
```

Delphi Examples: DbiActivateFilter

An example for this function is under development and will be provided in an upcoming Help release.

DbiAddAlias {button C Examples,JI(`>example',`exdbiaddalias')} {button Delphi Examples,JI(`>example',`dexdbiaddalias')}

C syntax

```
DBIResult DbiAddAlias( [hCfg], pszAliasName, pszDriverType, pszParams,
  bPersistent );
```

Delphi syntax

```
function DbiAddAlias (hCfg: hDBICfg; pszAliasName: PChar; pszDriverType:
  PChar; pszParams: PChar; bPersist: Bool): DBIResult stdcall;
```

Description

Adds an alias to the configuration file specified by the parameter *hCfg*.

Parameters

hCfg Type: hDBICfg (Input)
Specifies the configuration file to be used. This parameter is required to be NULL, indicating that the new alias is added to the configuration file for the current session.

pszAliasName Type: pCHAR (Input)
Pointer to the alias name. This is the name of the new alias that is to be added.

pszDriverType Type: pCHAR (Input)
Pointer to the driver type. This is the driver type for the new alias that is to be added. If this parameter is NULL, the alias will be for the STANDARD database. If szPARADOX, szDBASE, or szASCII are passed, this will add an entry in the STANDARD database alias generated to indicate that this will be the preferred driver type. If a driver name is an ODBC driver not previously added to the configuration file being modified, the BDE adds it automatically.

pszParams Type: pCHAR (Input)
Pointer to a list of optional parameters. This is a list defined as follows:

```
"AliasOption: Option Data[;AliasOption: Option Data][;...]"
```

AliasOption must correspond to a value retrieved by DbiOpenCfgInfoList. For a STANDARD database alias, the only valid parameter is PATH, all others will be ignored (no errors). If the parameter setting contains a semicolon (;) , enclose the entire setting in quotes ("Option Data";).

bPersistent Type: BOOL (Input)
This determines the scope of the new alias:

TRUE	Stored in the configuration file for future sessions.
FALSE	For use only in this session.

Examples

To set the path for a STANDARD database use:

```
"PATH:c:\mydata"
```

To set the server name and user name for a SQL driver use:

```
"SERVER NAME: server:/path/database;USER NAME: myname"
```

Usage

The alias added by this function will have whatever default values are associated with the driver specified unless they are specifically mentioned in the *pszParams* parameter. For a standard database alias, all entries in *pszParams* except PATH will be ignored. You can use DbiOpenCfgInfoList to modify the default values after DbiAddAlias has been called.

ODBC: DbiAddAlias automatically adds ODBC drivers and data sources as BDE aliases to the active session when they aren't currently stored in the configuration file. The BDE also supports ODBC 3 drivers.

Note: All changes you make to the current session are also applied to any sessions with

sesCFGUPDATE set to ON.

Prerequisites

DbiInit must be called prior to calling DbiAddAlias.

DbiResult return values

DBIERR_INVALIDPARAM	Null or invalid alias name. Invalid characters include a colon (:) and backslash (\ ). szASCII, szDBASE, and szPARADOX are entered as a STANDARD alias with the respective default driver.
DBIERR_NONE	The alias was added successfully.
DBIERR_NAMENOTUNIQUE	Another alias with the same name already exists (applicable only when *bPersistent* is TRUE).
DBIERR_OBJNOTFOUND	One (or more) of the optional parameters passed in through *pszParams* was not found as a valid type in the driver section of the configuration file.
DBIERR_UNKNOWNDRIVER	No driver name found in configuration file matching *pszDriverType*.

See Also

DbiInit, DbiOpenCfgInfoList, DbiAddDriver

C Examples: DbiAddAlias

Example 1: Add a STANDARD database alias to the configuration file.

This example uses the following input:

fDbiAddAlias1("TestAlias", "PATH:C:\\BDE32\\EXAMPLES\\TABLES");

```
DBIResult fDbiAddAlias1(char *AliasName, char *AliasPath)
{
   DBIResult   rslt;
   rslt = Chk(DbiAddAlias(NULL, AliasName, NULL, AliasPath, TRUE));
   return rslt;
}
```

Example 2: Add an InterBase database alias to the configuration file.

This example uses the following input:

fDbiAddAlias2("RemoteAlias",
"PATH:frobosrv:d:/interbas;"
"SERVER NAME:frobosrv:d:/interbas/slim.gdb;"
"USER NAME:test;"
"SQLQRYMODE:SERVER;"
"SQLPASSTHRU MODE:SHARED NOAUTOCOMMIT");

Note: The last parameter in the string does not have a semicolon (;) at the end.

```
DBIResult fDbiAddAlias2(char *AliasName, char *AliasPath)
{
   DBIResult   rslt;
   rslt = Chk(DbiAddAlias(NULL, AliasName, "INTRBASE", AliasPath, TRUE));
   return rslt;
}
```

Delphi Examples: DbiAddAlias

Example 1: Add a STANDARD database alias to the configuration file.
This example uses the following input:

```
  fDbiAddAlias1('TestAlias', 'PATH:C:\BDE32\EXAMPLES\TABLES');

procedure fDbiAddAlias1(AliasName, AliasPath: string);
begin
  Check(DbiAddAlias(nil, PChar(AliasName), nil, PChar(AliasPath), True));
end;
```

Example 2: Add an InterBase database alias to the configuration file.
This example uses the following input:

```
fDbiAddAlias2('RemoteAlias', 'PATH:frobosrv:d:/interbas;' +
            'SERVER NAME:frobosrv:d:/interbas/slim.gdb;' +
            'USER NAME:test;' +
            'SQLQRYMODE:SERVER;' +
            'SQLPASSTHRU MODE:SHARED NOAUTOCOMMIT');
```

Note: The last parameter in the string does not have a semicolon (;) at the end.

```
procedure fDbiAddAlias2(AliasName, AliasPath: string);
begin
  Check(DbiAddAlias(nil, PChar(AliasName), 'INTRBASE', PChar(AliasPath),
  True));
end;
```

DbiAddDriver {button C Examples,JI(`>example',`exdbiadddriver')} {button Delphi Examples,JI(`>example',`dexdbiadddriver')}

C syntax

```
DBIResult DbiAddDriver( [hCfg], pszDriverName, pszParams, bPersistent );
```

Delphi syntax

```
function DbiAddDriver (hCfg: hDBICfg; pszDriverName: PChar; pszParams:
  PChar; bPersist: Bool): DBIResult stdcall;
```

Description

Adds a driver to the configuration file specified by the parameter *hCfg*.

Parameters

hCfg Type: hDBICfg (Input)
Specifies the configuration file to be used. This parameter is required to be NULL, indicating that the new alias is added to the configuration file for the current session.

pszDriverName Type: pCHAR (Input)
Pointer to the driver name. This is the new driver that is to be added.

pszParams Type: pCHAR (Input)
Pointer to a list of optional parameters. This is a list defined as follows:

```
"AliasOption: Option Data[;AliasOption: Option Data][;...]"
```

AliasOption must correspond to a value retrieved by DbiOpenCfgInfoList. If the parameter setting contains a semicolon (;) , enclose the entire setting in quotes ("Option Data";).

bPersistent Type: BOOL (Input)
This determines the scope of the new alias:

TRUE	Stored in the configuration file for future sessions.
FALSE	For use only in this session.

Usage

The driver added by this function will have whatever default values are associated with the driver unless they are specifically mentioned in the *pszParams* parameter.

Note: All changes you make to the current session are also applied to any sessions with sesCFGUPDATE set to ON.

Prerequisites

DbiInit must be called prior to calling DbiAddDriver.

DbiResult return values

DBIERR_INVALIDPARAM	Null or invalid driver name. Invalid characters include a colon (:) and backslash (\ ).
DBIERR_NONE	The driver was added successfully.
DBIERR_NAMENOTUNIQUE	Another driver with the same name already exists (applicable only when *bPersistent* is TRUE).
DBIERR_OBJNOTFOUND	One (or more) of the optional parameters passed in through *pszParams* was not found as a valid type in the driver section of the configuration file.

See Also

DbiInit, DbiOpenCfgInfoList, DbiDeleteDriver, DbiAddAlias

C Examples: DbiAddDriver

An example for this function is under development and will be provided in an upcoming Help release.

Delphi Examples: DbiAddDriver

An example for this function is under development and will be provided in an upcoming Help release.

DbiAddFilter {button C Examples,JI(`>example',`exdbiactivatefilter')} {button Delphi Examples,JI(`>example',`dexdbiaddfilter')}

C syntax

```
DBIResult DBIFN DbiAddFilter (hCursor, [iClientData], [iPriority],
  [bCanAbort], pcanExpr, [pfFilter], phFilter);
```

Delphi syntax

```
function DbiAddFilter (hCursor: hDBICur; iClientData: Longint; iPriority:
  Word; bCanAbort: Bool; pcanExpr: pCANExpr; pfFilter: pfGENFilter; var
  hFilter: hDBIFilter): DBIResult stdcall;
```

Description

DbiAddFilter adds a filter to a table. When activated with DbiActivateFilter, only those records in the table that satisfy the filter condition are seen.

Parameters

hCursor Type: hDBICur (Input)
Specifies the cursor handle of the table to which the filter is being applied.

iClientData Type: UINT32 (Input)
Not currently used. Must be 0.

iPriority Type: UINT16 (Input)
Not currently used. Must be 1.

bCanAbort Type: BOOL (Input)
Not currently used. Must be FALSE.

pcanExpr Type: pCANExpr (Input)
Pointer to the CANExpr structure, which describes the filter condition as a Boolean expression in prefix format.

pfFilter Type: pfGENFilter (Input)
Not currently used. Must be NULL.

phFilter Type: phDBIFilter (Output)
Pointer to the filter handle.

Usage

Filters subset result sets. They are similar to a SQL statement's WHERE clause, but are expressed in prefix format. The filter must be specified by the client as a filter expression returning TRUE or FALSE. Multiple filters are allowed per table, and if more than one filter is active, records that violate any active filter condition are not included in the result set. Filters can be switched on and off when needed (using DbiActivateFilter and DbiDeactivateFilter), and are automatically dropped when the table is closed.

DbiGetSeqNo is not influenced by filters; the sequence number returned is that of the record in the original table. DbiGetRecordCount does not guarantee to return an exact count of all records in the filter set. Use DbiGetExactRecordCount to return the exact count of all records in the filter set. Drivers can return the count of all records (including those not satisfying the filter condition) or can return an estimate.

Note: Passthrough SQL query cursors do not support this function currently.

Oracle8: Not supported for object types (ADT, REF, nested table, and VARARRAY).

DbiResult return values

DBIERR_NONE The filter has been successfully added.

DBIERR_INVALIDHNDL The specified cursor handle is invalid or NULL.

DBIERR_NA	The filter condition described by the filter expression could not be handled by the driver.

See also

DbiActivateFilter, DbiDeactivateFilter, DbiDropFilter

C Examples: DbiAddFilter

An example for this function is under development and will be provided in an upcoming Help release.

Delphi Examples: DbiAddFilter

Add a filter that only shows records that the specified field is not NULL. This example uses the following input:

fDbiAddFilter(Table1, Table1.FieldByName('Addr1'), hFilter);

Note: When done with filter, call DbiDeactivateFilter and DbiDropFilter. If table is connected to data aware cnotrols, call TTable.Refresh.

```
procedure fDbiAddFilter(Table: TTable; Field: TField; var hF: hDBIFilter);
type
  // Setup the node structure
  TNodes = record
    UNode: CANUnary;
    FNode: CANField;
  end;

var
  Nodes: TNodes;
  Expression: CANExpr;
  pCan: pByte;

begin
  // Uninary Node - set the operator to NOT BLANK or (NOT NULL)
  with Nodes.UNode do begin
    nodeClass := nodeUNARY;
    canOp := canNOTBLANK;
    iOperand1 := 12;
  end;
  // Field Node - set the field number and literal pool offset
  with Nodes.FNode do begin
    nodeClass := nodeFIELD;
    canOp := canFIELD2;
    iFieldNum := Field.Index + 1;
    iNameOffset := 0;
  end;
  // Expression - set the expression size, nodes and start positions
  with Expression do begin
    iVer := 1;
    iTotalSize := sizeof(CANExpr) + sizeof(Nodes) + Length(Field.FieldName)
  + 1;
    iNodes := 2;
    iNodeStart := sizeof(CANExpr);
    iLiteralStart := sizeof(CANExpr) + sizeof(Nodes);
  end;

  GetMem(pCan, Expression.iTotalSize * sizeof(BYTE));
  try
    // Move expression, nodes and literal pool into a contiguous memory
  space
    Move(Expression, pCan^, sizeof(CANExpr));
    Inc(pCan, sizeof(CANExpr));
    Move(Nodes, pCan^, sizeof(Nodes));
    Inc(pCan, sizeof(Nodes));
    Move(Field.FieldName, pCan^, Length(Field.FieldName) + 1);
    Dec(pCan, sizeof(Nodes) + sizeof(CANExpr));
    // Add and activate the filter
    Check(DbiAddFilter(Table.Handle, 0, 1, False, pCANExpr(pCan), nil, hF));
```

```
    Check(DbiActivateFilter(Table.Handle, hF));
  finally
    FreeMem(pCan, Expression.iTotalSize * sizeof(BYTE));
  end;
end;
```

DbiAddIndex {button C Examples,JI(`>example',`exdbiaddindex')} {button Delphi Examples,JI(`>example',`dexdbiaddindex')}

C syntax

```
DBIResult DBIFN DbiAddIndex (hDb, hCursor, pszTableName, [pszDriverType],
  pIdxDesc, [pszKeyviolName]);
```

Delphi syntax

```
function DbiAddIndex (hDb: hDBIDb; hCursor: hDBICur; pszTableName: PChar;
  pszDriverType: PChar; var IdxDesc: IDXDesc; pszKeyviolName: PChar):
  DBIResult stdcall;
```

Description

DbiAddIndex creates an index on an existing table specified by *pszTableName* or associated with the cursor handle specified by *hCursor*.

Parameters

hDb Type: hDBIDb (Input)
Specifies the database handle.

hCursor Type: hDBICur (Input)
Specifies the cursor on the table. Optional. If *hCursor* is specified, the operation is performed on the table associated with the cursor. If *hCursor* is NULL, *pszTableName* and *pszTableType* determine the table to be used. This option is not supported with Access tables.

pszTableName Type: pCHAR (Input)
Pointer to the table name. Optional. If *hCursor* is NULL, *pszTableName* and *pszDriverType* determine the table to be used. (If both *pszTableName* and *hCursor* are specified, *pszTableName* is ignored.)

For Paradox, dBASE, and FoxPro, if *pszTableName* is a fully qualified name of a table, the *pszDriverType* parameter need not be specified. If the path is not included, the path name is taken from the current directory of the database associated with *hDb*.

For SQL databases, this parameter can be a fully qualified name that includes the owner name.

pszDriverType Type: pCHAR (Input)
Pointer to the driver type. Optional. For Paradox, dBASE, and FoxPro tables, this parameter is required if *pszTableName* has no extension. This parameter is ignored if the database associated with *hDb* is a SQL database. *pszDriverType* can be one of the following values: szDBASE, szMSACCESS, or szPARADOX.

pIdxDesc Type: pIDXDesc (Input)
Pointer to the index descriptor structure (IDXDesc). The IDXDesc elements required vary by database driver.

pszKeyviolName Type: pCHAR (Input/Output)
Optional. Pointer to a key violation table name (a buffer of DBIMAXPATHLEN+1 characters). You can specify a table name or use this parameter to retrieve the name generated by the BDE. This parameter is not supported for Access tables.

Usage

If a cursor handle is supplied, the function generally does not affect the order or the position of the cursor. However, adding Paradox primary indexes sets the cursor position to the beginning of the file.

Index descriptors vary by driver. For details, see IDXDesc and IDXType

dBASE: The client application must have permission to lock the table exclusively.

FoxPro: The client application must have permission to lock the table exclusively.

SQL: The client application must have the appropriate privileges to add indexes. Also, if an index is added to any SQL table, then any cursors open on that table must be closed and reopened, to allow for possible changes in the buffer size.

Paradox: The client application must have permission to lock the table exclusively. If adding a non-maintained Paradox index, only a read lock is required.

Access: You can't add indexes to an open cursor, and Access does not support key violation tables.

Oracle8: Not supported for object types (ADT, REF, nested table, and VARARRAY).

Prerequisites

If the table name or cursor handle is used to specify the table, the cursor must be opened exclusively on behalf of the client application, and is closed after the index has been created. If the index is a maintained or primary Paradox index, the cursor also must be opened exclusively.

Completion state

Before the cursor is reordered to reflect the newly added index, the application must use or switch to the index.

DbiResult return values

DBIERR_NONE The index was successfully added.

DBIERR_INVALIDHNDL	The specified database handle or the cursor handle (if specified) is invalid or NULL.
DBIERR_INVALIDPARAM	Neither hCursor nor pszTableName was specified.
DBIERR_UNKNOWNTBLTYPE	The parameter, pszDriverType is invalid.
DBIERR_PRIMARYKEYREDEFINE	The primary index already exists; illegal to define another.
DBIERR_INVALIDINDEXTYPE	The index descriptor is invalid.
DBIERR_INVALIDIDXDESC	The index descriptor is invalid.
DBIERR_INVALIDFLDTYPE	Attempting to index an invalid field type (that is, BLOB field)
DBIERR_INVALIDINDEXNAME	The index name or tag name is invalid (usually for dBASE or FoxPro tables)
DBIERR_NAMEREQUIRED	Index name is required.
DBIERR_NAMENOTUNIQUE	Index name was not unique.
DBIERR_MUSTUSBASEORDER	The default order must be used when adding an index.
DBIERR_NEEDEXCLACCESS	Table is opened in share mode when creating a maintained or primary index.

See also

DbiOpenIndexList, DbiGetIndexDesc, DbiSetToKey, DbiRegenIndex, DbiRegenIndexes, DbiDeleteIndex, DbiOpenIndex, DbiCloseIndex, DbiSwitchToIndex, DbiCreateTable, DbiDoRestructure

C Examples: DbiAddIndex

Example 1: Add an index to a Paradox 4.0 or 5.0 version table:
Note: This is a primary index.

```
DBIResult fDbiAddIndex1(hDBIDb hTmpDb, hDBICur hTmpCur)
{
   DBIResult   rslt;
   IDXDesc     NewIndex;
   DBIKEY      aiKeys = { 1 };   // Field to put index on

   NewIndex.iIndexId = 0;
   NewIndex.bPrimary = TRUE;
   NewIndex.bUnique = TRUE;
   NewIndex.bDescending = FALSE;
   NewIndex.bMaintained = TRUE;
   NewIndex.bSubset = FALSE;
   NewIndex.bExpIdx = FALSE;
   NewIndex.iFldsInKey = 1;
   memcpy(NewIndex.aiKeyFld, aiKeys, sizeof(DBIKEY));
   NewIndex.bCaseInsensitive = FALSE;
   rslt = Chk(DbiAddIndex(hTmpDb, hTmpCur, NULL, NULL, &NewIndex, NULL));

   return rslt;
}
```

Example 2: Add an index to a Paradox 4.0 or 5.0 version table:
Note: This is a case-insensitive, secondary, maintained index:

```
DBIResult fDbiAddIndex2(hDBIDb hTmpDb, hDBICur hTmpCur)
{
   DBIResult   rslt;
   IDXDesc     NewIndex;
   DBIKEY      aiKeys = { 2 };   // Field to put index on

   strcpy(NewIndex.szName, "TempIndex");
   NewIndex.bPrimary = FALSE;
   NewIndex.bUnique = FALSE;
   NewIndex.bDescending = FALSE;
   NewIndex.bMaintained = TRUE;
   NewIndex.bSubset = FALSE;
   NewIndex.bExpIdx = FALSE;
   NewIndex.iFldsInKey = 1;
   memcpy(NewIndex.aiKeyFld, aiKeys, sizeof(DBIKEY));
   NewIndex.bCaseInsensitive = TRUE;
   rslt = Chk(DbiAddIndex(hTmpDb, hTmpCur, NULL, NULL, &NewIndex, NULL));

   return rslt;
}
```

Example 3: Add an index to a dBASE for Windows version table:

```
DBIResult fDbiAddIndex3(hDBIDb hTmpDb, hDBICur hTmpCur)
{
   DBIResult   rslt;
   IDXDesc     NewIndex;
   DBIKEY      aiKeys = { 2 };   // Field to put index on

   strcpy(NewIndex.szTagName, "TestIndex");
   NewIndex.bPrimary = FALSE;
   NewIndex.bUnique = FALSE;
   NewIndex.bDescending = FALSE;
   NewIndex.bMaintained = TRUE;
   NewIndex.bSubset = FALSE;
   NewIndex.bExpIdx = FALSE;
   NewIndex.iFldsInKey = 1;
   memcpy(NewIndex.aiKeyFld, aiKeys, sizeof(DBIKEY));
   strcpy(NewIndex.szKeyExp, "");   // Although this is not an Expression
  index,
   strcpy(NewIndex.szKeyCond, "");  // szKeyExp and szKeyCond must be set
  blank
   NewIndex.bCaseInsensitive = FALSE;
   NewIndex.iBlockSize = 0;

   rslt = Chk(DbiAddIndex(hTmpDb, hTmpCur, NULL, NULL, &NewIndex, NULL));
   return rslt;
}
```

Example 4: Add an expression index to a dBASE for Windows version table:

```
DBIResult fDbiAddIndex4(hDBIDb hTmpDb, hDBICur hTmpCur, char *Expression)
{
   DBIResult   rslt;
   IDXDesc     NewIndex;
   DBIKEY      aiKeys = { 0 };   // Field to put index on

   strcpy(NewIndex.szTagName, "ExpIndex");
   NewIndex.bPrimary = FALSE;
   NewIndex.bUnique = FALSE;
   NewIndex.bDescending = FALSE;
   NewIndex.bMaintained = TRUE;
   NewIndex.bSubset = FALSE;
   NewIndex.bExpIdx = TRUE;
   NewIndex.iFldsInKey = 0;
   memcpy(NewIndex.aiKeyFld, aiKeys, sizeof(DBIKEY));
   strcpy(NewIndex.szKeyExp, Expression);
   strcpy(NewIndex.szKeyCond, "");
   NewIndex.bCaseInsensitive = FALSE;
   NewIndex.iBlockSize = 0;

   rslt = Chk(DbiAddIndex(hTmpDb, hTmpCur, NULL, NULL, &NewIndex, NULL));
   return rslt;
}
```

Delphi Examples: DbiAddIndex

Also see TTable.AddIndex in the Delphi online help. TTable.AddIndex will usually handle most of your indexing needs.

Example 1: Add an index to a Paradox 4.0 or 5.0 version table:
This is a primary index. This example uses the following input:

```
  fDbiAddIndex1(Table1);


procedure fDbiAddIndex1(Tbl: TTable);
var
  NewIndex: IDXDesc;
begin
  if not Tbl.Exclusive then
    raise EDatabaseError.Create('TTable.Exclusive must be set to ' +
      'true in order to add an index to the table');
  with NewIndex do begin
    iIndexId:= 0;
    bPrimary:= True;
    bUnique:= True;
    bDescending:= False;
    bMaintained:= True;
    bSubset:= False;
    bExpIdx:= False;
    iFldsInKey:= 1;
    aiKeyFld[0]:= 1;
    bCaseInsensitive:= False;
  end;
  Check(DbiAddIndex(Tbl.dbhandle, Tbl.handle, PChar(Tbl.TableName),
          szParadox, NewIndex, nil));
end;
```

Example 2: Add an index to a Paradox 4.0 or 5.0 version table.
This is a case insensitive, secondary, maintained index. This example uses the following input:

```
   fDbiAddIndex2(Table1);
```

The procedure is defined as:

```
procedure fDbiAddIndex2(Tbl: TTable);
var
  NewIndex: IDXDesc;
  Buffer: pchar;
begin
  if not Tbl.Exclusive then
    raise EDatabaseError.Create
    ('TTable.Exclusive must be set to true in order to ' +
      'add an index to the table');
  with NewIndex do begin
    szName:= 'NewIndex';
    iIndexId:= 0;
    bPrimary:= False;
    bUnique:= False;
    bDescending:= False;
    bMaintained:= True;
    bSubset:= False;
```

```
    bExpIdx:= False;
    iFldsInKey:= 1;
    aiKeyFld[0]:= 2;
    bCaseInsensitive:= True;
  end;
  Check(DbiAddIndex(Tbl.dbhandle, Tbl.handle, PChar(Tbl.TableName),
    szParadox, NewIndex, nil));
end;
```

Example 3: Add an index to a Paradox 7.0 version table.
This is a secondary unique / descending index. This example uses the following input:

```
  fDbiAddIndex3(Table1);
```

The procedure is defined as:

```
procedure fDbiAddIndex3(Tbl: TTable);
var
  NewIndex: IDXDesc;
begin
  if not Tbl.Exclusive then
    raise EDatabaseError.Create
    ('TTable.Exclusive must be set to true in order to ' +
      'add an index to the table');
  NewIndex.szName := 'NewIndex';
  NewIndex.iIndexId := 0;
  NewIndex.bPrimary := False;
  NewIndex.bUnique := TRUE;
  NewIndex.bDescending := True;
  NewIndex.bMaintained := True;
  NewIndex.bSubset := False;
  NewIndex.bExpIdx := False;
  NewIndex.iFldsInKey := 1;
  NewIndex.aiKeyFld[0]:= 2;
  NewIndex.bCaseInsensitive := True;
  Check(DbiAddIndex(Tbl.dbhandle, Tbl.handle, PChar(Tbl.TableName),
          szParadox, NewIndex, nil));
end;
```

Example 4: Add an index to a dBASE for Windows version table.
This example uses the following input:

```
  fDbiAddIndex4(Table1);
```

The procedure is defined as:

```
procedure fDbiAddIndex4(Tbl: TTable);
var
  NewIndex: IDXDesc;
begin
  NewIndex.szTagName := 'NewIndex1';
  NewIndex.bPrimary := False;
  NewIndex.bUnique := False;
  NewIndex.bDescending := False;
  NewIndex.bMaintained := True;
  NewIndex.bSubset := False;
  NewIndex.bExpIdx := False;
  NewIndex.iFldsInKey := 1;
  NewIndex.aiKeyFld[0] := 2;
```

```
  NewIndex.szKeyExp := '';   // Although this is not an Expression index,
  NewIndex.szKeyCond := '';  // szKeyExp and szKeyCond must be set blank
  NewIndex.bCaseInsensitive := False;
  NewIndex.iBlockSize := 0;
  Check(DbiAddIndex(Tbl.dbhandle, Tbl.handle, PChar(Tbl.TableName),
    szParadox, NewIndex, nil));
end;
```

Example 5: Add an expression index to a dBASE for Windows version table.
This example uses the following input:

```
  fDbiAddIndex5(Table1);
```

The procedure is defined as:

```
procedure fDbiAddIndex5(Tbl: TTable);
var
  NewIndex: IDXDesc;
begin
  NewIndex.szTagName := 'EXPINDEX';
  NewIndex.bPrimary := False;
  NewIndex.bUnique := False;
  NewIndex.bDescending := False;
  NewIndex.bMaintained := True;
  NewIndex.bSubset := False;
  NewIndex.bExpIdx := True;
  NewIndex.iFldsInKey := 1;
  NewIndex.aiKeyFld[0] := 2;
  NewIndex.szKeyExp := 'UPPER(FIELD1) + UPPER(FIELD2)';
  NewIndex.szKeyCond := '';
  NewIndex.bCaseInsensitive := False;
  NewIndex.iBlockSize := 0;
  Check(DbiAddIndex(Tbl.dbhandle, Tbl.handle, PChar(Tbl.TableName),
    szDBASE, NewIndex, nil));
end;
```

DbiAddPassword {button C Examples,JI(`>example',`exdbiaddpassword')} {button Delphi Examples,JI(`>example',`dexdbiaddpassword')}

C syntax
```
DBIResult DBIFN DbiAddPassword (pszPassword);
```

Delphi syntax
```
function DbiAddPassword (pszPassword: PChar): DBIResult stdcall;
```

Description
DbiAddPassword adds a password to the current session. This function is supported for Paradox tables only.

Parameters
pszPassword Type: pCHAR (Input)
Pointer to the password to be added.

Usage
DbiAddPassword provides users with access to a previously encrypted table (adding a password does not encrypt the table). Examples of operations on an encrypted table include: opening the table, record and field access on the table, and batch functions (copy, delete, empty, or restructure). DbiCreateTable and DbiDoRestructure can be used to place or remove table encryption.

Paradox: Table and field level security is supported for the Paradox driver only.

SQL: This function is not supported with SQL tables. Access rights for SQL drivers are controlled when the database is opened.

DbiResult return values
DBIERR_NONE The password was successfully added.

DBIERR_PASSWORDLIMIT Maximum number of passwords have already been added.

DBIERR_INVALIDPASSWORD The specified password is invalid (for example, it is too long or contains invalid characters).

See also
DbiDropPassword, DbiCreateTable, DbiDoRestructure

C Examples: DbiAddPassword

Create a table with a password and add that password to the current session.

If phCur is not NULL, return an open cursor. This example uses the following input:

```
fDbiAddPassword(hDb, &hTmpCur, "BORLAND C");
```

```
DBIResult fDbiAddPassword(hDBIDb hDb, phDBICur phCur, pCHAR pPWD)
{
   DBIResult   rslt;
   CRTblDesc   TblDesc;
   FLDDesc     fldDesc[] = {{ // Field 1 - ALPHA
               1, "MyAlpha", fldZSTRING, fldUNKNOWN, 10, 0,
               0, 0, 0, fldvNOCHECKS, fldrREADWRITE
               }};

   memset((void *) &TblDesc, 0, sizeof(CRTblDesc));
   lstrcpy(TblDesc.szTblName, "COBRA.DB");
   TblDesc.iFldCount = 1;
   TblDesc.pfldDesc = fldDesc;
   TblDesc.bProtected = TRUE;
   lstrcpy(TblDesc.szPassword, pPWD);

   rslt = Chk(DbiAddPassword(pPWD));
   if (rslt != DBIERR_NONE)
      return rslt;

   rslt = Chk(DbiCreateTable(hDb, TRUE, &TblDesc));
   if (rslt != DBIERR_NONE)
      return rslt;

   if (phCur != NULL)
     rslt = Chk(DbiOpenTable(hDb, TblDesc.szTblName, NULL, NULL, NULL, NULL,
                  dbiREADWRITE, dbiOPENSHARED, xltFIELD, FALSE, NULL,
  phCur));

   return rslt;
}
```

Delphi Examples: DbiAddPassword

Add a password to the current session.

Delphi users should use TSession.AddPassword rather than directly calling dbiAddPassword. The method TSession.AddPassword is defined as:

```
  procedure AddPassword(const Password: string);
```

The following code adds the password "Hip Hop" to TSession Session:

```
  Session.AddPassword('Hip Hop');
```

Add a password to the specified handle's session:

Delphi users should use TSession.AddPassword. See DbiGetCurrSession to get the current session's handle. This example uses the following input:

```
  fDbiAddPassword(Session3.Handle, 'SPRINT');
```

The procedure is defined as:

```
procedure fDbiAddPassword(hSes: hDBISes; Pswd: string);
begin
  Check(DbiSetCurrSession(hSes));
  Check(DbiAddPassword(PChar(Pswd)));
end;
```

DbiAnsiToNative {button C Examples,JI(`>example',`exdbiansitonative')} {button Delphi Examples,JI(`>example',`dexdbiansitonative')}

C syntax

```
DBIResult DBIFN DbiAnsiToNative (pLdObj, pOemStr, pAnsiStr, iLen,
  pbDataLoss);
```

Delphi syntax

```
function DbiAnsiToNative (LdObj: Pointer; pNativeStr: PChar; pAnsiStr:
  PChar; iLen: Word; var bDataLoss: Bool): DBIResult stdcall;
```

Description

DbiAnsiToNative translates strings from ANSI to the language driver's native character set. If the native character set is ANSI, no translation takes place.

Parameters

pLdObj Type: pVOID (Input)
Pointer to the language driver object returned from DbiGetLdObj.

pOemStr Type: pCHAR (Output)
Pointer to the client buffer where the translation string is placed. If *pOemStr* equals *pAnsiStr*, conversion occurs in place.

pAnsiStr Type: pCHAR (Input)
Pointer to the client buffer containing the ANSI data.

iLen Type: UINT16 (Input)
If *iLen* equals 0, assumes null-terminated string; otherwise *iLen* specifies the length of the buffer to convert.

pbDataLoss Type: pBOOL (Output)
Pointer to a client variable. If set to TRUE, the ANSI string cannot map to a character in the native character set.

Usage

Works on drivers with both ANSI and OEM native character sets. Does not handle multi-byte character sets, such as Japanese ShiftJIS. If the native character set is ANSI, no translation takes place. See International Compatibility

DBIResult return values

DBIERR_NONE The translation completed successfully.

DBIERR_INVALIDPARAM	A parameter passed to the function is invalid.
DBIERR_NOTSUPPORTED	A function is not supported by the driver.

See also

DbiNativeToAnsi, DbiGetLdObj

C Examples: DbiAnsiToNative

Translate the ANSI string into the specified tables's language driver native character set.

This example uses the following input:

fDbiAnsiToNative(hPXCur, Buffer, ANSIStr);

```
DBIResult fDbiAnsiToNative(hDBICur hTmpCur, pCHAR OemStr, pCHAR AnsiStr)
{
   DBIResult   rslt;
   pVOID       LD;
   BOOL        Loss;

   rslt = Chk(DbiGetLdObj(hTmpCur, &LD));
   if (rslt != DBIERR_NONE)
      return rslt;

   rslt = Chk(DbiAnsiToNative(LD, OemStr, AnsiStr, 0, &Loss));
   if (rslt != DBIERR_NONE)
      return rslt;
   if (Loss == TRUE)
      return DBIERR_INVALIDLANGDRV;

   return rslt;
}
```

Delphi Examples: DbiAnsiToNative

Translate a string from the ANSI character set to the language driver's native character set. This example uses the following input:

```
  fDbiAnsiToNative(CustTbl, string);
```

The function is defined as:

```
function fDbiAnsiToNative(Table: TTable; AnsiStr: string): string;
var
  pDesc: pLDDesc;
  Len: Word;
  Done: Boolean;
begin
  Len := Length(AnsiStr);
  SetLength(Result, Len);
  Check(DbiGetLDObj(Table.Handle, pointer(pDesc)));
  Check(DbiAnsiToNative(pointer(pDesc), PChar(Result), PChar(AnsiStr), Len,
  Done));
end;
```

DbiAppendRecord {button C Examples,JI(`>example',`exdbiappendrecord')}{button Delphi Examples,JI(`>example',`dexdbiinsertrecord')}

C syntax

```
DBIResult DBIFN DbiAppendRecord (hCursor, pRecBuf);
```

Delphi syntax

```
function DbiAppendRecord (hCursor: hDBICur; pRecBuff: Pointer): DBIResult
  stdcall;
```

Description

DbiAppendRecord appends a record to the end of the table associated with the cursor.

Parameters

hCursor Type: hDBICur (Input)
Specifies the cursor handle to the table to which the record is being appended.

pRecBuf Type: pBYTE (Input)
Specifies the pointer to the record buffer.

Usage

The contents of the current record buffer are appended. This function is equivalent to calling DbiSetToEnd followed by DbiInsertRecord.

dBASE, FoxPro, and Access: This function behaves the same as DbiInsertRecord.

Paradox: For tables with a primary index, where physical reordering of records is forced, DbiAppendRecord is equivalent to DbiInsertRecord. If referential integrity or validity checks are applied to the Paradox table, the data is verified prior to appending the record. If any of the checks fail, an error is returned and the operation is not completed.

SQL: This function behaves the same as DbiInsertRecord.

Prerequisites

A valid cursor handle must be obtained. Other users cannot have a write lock on the table. The record buffer should be initialized with DbiInitRecord, and data filled in using DbiPutField.

Completion state

This function leaves the cursor positioned on the inserted record. If there is an active range and the inserted record falls outside the range, the cursor might be positioned at the beginning or end of the file.

DbiResult return values

DBIERR_NONE	The data was successfully appended.
DBIERR_INVALIDHNDL	The specified cursor is invalid or NULL.
DBIERR_INVALIDPARAM	The record buffer is NULL.
DBIERR_KEYVIOL	The table has a unique index and the inserted key value conflicts with an existing record's key value.
DBIERR_FOREIGNKEYERR	A linking field value does not exist in the corresponding master table (Paradox only).
DBIERR_MINVALERR	The specified data is less than the required minimum value.
DBIERR_MAXVALERR	The specified data is greater than the required maximum value.

DBIERR_LOOKUPTABLEERR	One or more of the fields in the record buffer have failed an existing validity check (Paradox only).
DBIERR_REQDERR	A required field in the record buffer was left blank (not applicable to dBASE or FoxPro).
DBIERR_TABLEREADONLY	Table access denied; the cursor does not have write access to the table.
DBIERR_NOTSUFFTABLERIGHTS	Insufficient table rights to append a record (Paradox only).
DBIERR_NOTSUFFSQLRIGHTS	Insufficient SQL rights for operation.
DBIERR_NODISKSPACE	The record cannot be appended because there is insufficient disk space.

See also

DbiGetNextRecord, DbiGetPriorRecord, DbiGetRecord, DbiGetCursorProps, DbiGetRelativeRecord, DbiOpenTable, DbiInitRecord, DbiPutBlob, DbiPutField, DbiVerifyField

For SQL-related restrictions, see DbiInsertRecord.

C Examples: DbiAppendRecord

An example for this function is under development and will be provided in an upcoming Help release.

Delphi Examples: DbiAppendRecord

Append a record to the end of the table associated with the cursor.

Delphi users should use TTable.AppendRecord rather than directly calling dbiAppendRecord. The TTable.AppendRecord method is defined as:

```
  procedure AppendRecord(const Values: array of const);
```

This statement appends a record to a TTable called Customer. Note that NULL values (using the Pascal nil) are entered for some of the values, but are not required for missing values at the end of the array argument, for example, after the Discount field.

```
Customer.AppendRecord([CustNoEdit.Text, CoNameEdit.Text, AddrEdit.Text,
  nil, nil, nil, nil, nil, nil, DiscountEdit.Text]);
```

DbiApplyDelayedUpdates {button C Examples,JI(`>example',`exdbiapplydelayedupdates')} {button Delphi Examples,JI(`>example',`dexdbiapplydelayedupdates')}

C syntax

```
DBIResult DBIFN EXPORT DbiApplyDelayedUpdates (hCursor,eUpdCmd);

typedef enum           // Op types for cached updates cursor
  {
     dbiDelayedUpdCommit        = 0  // Commit the updates
     dbiDelayedUpdCancel        = 1  // Rollback the updates
     dbiDelayedUpdCancelCurrent = 2  // Cancel current record change
     dbiDelayedUPDPREPARE       = 3  // Phase 1 of two-phase commit
  } DBIDelayedUpdCmd;
```

Delphi syntax

```
function DbiApplyDelayedUpdates (hCursor: hDBICur; eUpdCmd:
  DBIDelayedUpdCmd): DBIResult stdcall;
```

Description

When the cached updates mode is active, DbiApplyDelayedUpdates writes any changes made to cached data to the underlying database, or rolls back all modifications made to the cached data. DbiApplyDelayedUpdates sends to the database a batch of all inserts, deletes, and modifications made since the last DbiApplyDelayedUpdates function call.

Parameters

hCursor Type: hDBICur (Input)
Specifies the cached updates cursor handle.

eUpdCmd Type: DBIDelayedUpdCmd (Input)
Specifies the operation to be performed on the cached updates cursor.

Usage

After making the changes to table data cached by the cached updates mode, call DbiApplyDelayedUpdates either to commit (write to the actual table) or rollback the changes. The rollback operation quickly discards the update information from the cache.

You may continue modifying data in the cached updates mode after calling DbiApplyDelayedUpdates. When you are ready to write the modified data permanently, call DbiApplyDelayedUpdates to commit changes to the actual database. When finished, DbiEndDelayedUpdates closes the cached updates mode.

Use of the cached updates mode is a two-phase process involving the use of the operation types for the cached updates cursor:

Phase 1: The operation DbiDelayedUpdPrepare causes all changes in the cache to be applied to the underlying data. Unless being used in a single-user environment, this operation should always be used within the context of a transaction to allow for error-recovery in the event of an error during the update. Any errors encountered during this phase should be handled through callback functions.

Phase 2: The operation dbiDelayedUpdateCommit performs the second phase. After successfully calling DbiDelayedUpdPrepare directly, follow it with the dbiDelayedUpdateCommit operation. The internal cache is updated to reflect the fact that the updates were successfully applied to the underlying database (that is, the successfully applied records are removed from the cache).

There are two ways to cancel changes made while cached updates are enabled:

The operation dbiDelayedUpdCancel clears the cache and restores the dataset to the state it was in when:

- the table was opened,
- cached updates were enabled, or
- updates were last successfully applied.

The operation dbiDelayedUpdCancelCurrent restores the current record in the dataset to an unmodified state. If the record was not modified this call has no effect. This operation is similar to the dbiDelayedUpdCancel operation but operates only on the current record.

Standard: Every non-blob field is used in determining the record modifications

Prerequisites

A call to DbiBeginDelayedUpdates must have been made.

DbiResult return values

DBIERR_NONE	The update information in the temporary cache was successfully written to the database.

See also

DbiBeginDelayedUpdates, DbiEndDelayedUpdates, Cached updates

C Examples: DbiApplyDelayedUpdates

An example for this function is under development and will be provided in an upcoming Help release.

Delphi Examples: DbiApplyDelayedUpdates

An example for this function is under development and will be provided in an upcoming Help release.

DbiBatchMove {button C Examples,JI(`>example',`exdbibatchmove')} {button Delphi Examples,JI(`>example',`dexdbibatchmove')}

C syntax

```
DBIResult DBIFN DbiBatchMove (pSrcTblDesc, hSrcCur, pDstTblDesc, hDstCur,
  ebatMode, iFldCount, pSrcFldMap, pszIndexName, pszIndexTagName, iIndexId,
  [pszKeyviolName], [pszProblemsName], [pszChangedName], p1ProbRecs,
  p1KeyvRecs, p1ChangedRecs, bAbortOnFirstProb, bAbortOnFirstKeyviol,
  p1RecsToMove, bTransliterate);
```

Delphi syntax

```
function DbiBatchMove (pSrcTblDesc: pBATTblDesc; hSrcCur: hDBICur;
  pDstTblDesc: pBATTblDesc; hDstCur: hDBICur; ebatMode: eBATMode; iFldCount:
  Word; pSrcFldMap: PWord; pszIndexName: PChar; pszIndexTagName: PChar;
  iIndexId: Word; pszKeyviolName: PChar; pszProblemsName: PChar;
  pszChangedName: PChar; lProbRecs: PLongint; lKeyvRecs: PLongint;
  lChangedRecs: PLongint; bAbortOnFirstProb: Bool; bAbortOnFirstKeyviol:
  Bool; var lRecsToMove: Longint; bTransliterate: Bool): DBIResult stdcall;
```

Description

DbiBatchMove is used to append, update, or subtract records from a source table to a destination table. It can also be used to copy an entire table to a table of a different driver type.

Parameters

pSrcTblDesc Type: pBATTblDesc (Input)
Optional. Pointer to the source table descriptor (BATTblDesc). If NULL, then *hSrcCur* is used to identify the source table. If not NULL, the specified table is opened, and the entire table is processed.

hSrcCur Type: hDBICur (Input)
Optional. Specifies the cursor handle of the source table; *hSrcCur* is used only if *psrcTab* is NULL. The source table is processed from the current position of the cursor.

pDstTblDesc Type: pBATTblDesc (Input)
Optional. Pointer to the destination table descriptor (BATTblDesc). If NULL, then *hDstCur* is used to identify the destination table. If not NULL, the specified table is opened, and the entire table is processed. Must be specified if mode is batCOPY.

hDstCur Type: hDBICur (Input)
Optional. Specifies the cursor handle of the destination table; *hDstCur* is used only if pdstTab is NULL. The destination table is processed from the current position of the cursor.

ebatMode Type: eBATMode (Input)
Specifies the mode; valid modes are batAPPEND, batUPDATE, batAPPENDUPDATE, batSUBTRACT, or batCOPY. The mode determines how the append operation is used. See the Usage section for details.

iFldCount Type: UINT16 (Input)
Specifies the number of fields in *pSrcFldMap*. Optional. Normally set to 0.

pSrcFldMap Type: pUINT16 (Input)
Pointer to an array of field numbers in the source table to be copied; the number of fields in the array must be equal to *iFldCount*. Optional. If set to NULL, the fields in the source are matched from left to right with the fields in the destination. This array is indexed by the destination field position (0 to n-1) and contains either the source field number (1 to n) to be matched with the destination or zero to leave the destination field blank or unmodified.

pszIndexName Type: pCHAR (Input)

Pointer to the index name. Optional. This parameter is used only when *ebatMode* is batUPDATE, batAPPENDUPDATE, or batSUBTRACT to specify the index used by the destination table to define matching records.

pszIndexTagName Type: pCHAR (Input)
Pointer to the index tag name. Optional. This parameter is used only when *ebatMode* is batUPDATE, batAPPENDUPDATE, or batSUBTRACT to specify the index used by the destination table to define matching records.

iIndexId Type: UINT16 (Input)
Specifies the index identification number. Optional. This parameter is used only when *ebatMode* is batUPDATE, batAPPENDUPDATE, or batSUBTRACT to specify the index used by the destination table to define matching records.

pszKeyviolName Type: pCHAR (Input)
Optional. Pointer to the Key Violation table name. All records that cause an integrity violation when inserted or updated into the destination table can be placed here. If NULL, no Key Violation table is created. If the user supplies a table name, that name is used. If not NULL and a pointer to a NULL character is specified, BDE generates a name for the auxiliary table and copies the name back to the location specified by the pointer; therefore, this area must be at least DBIMAXPATHLEN+1 bytes. If no auxiliary table is created, this area is set to all NULLs.

pszProblemsName Type: pCHAR (Input)
Optional. Pointer to the Problems table name. Unless the user has overridden the default behavior with a callback, records are placed in a Problems table if they cannot be placed into the destination table without trimming data.

If NULL, no Problems table is created. If the user supplies a table name, that name is used. If not NULL and a pointer to a NULL character is specified, BDE generates a name for the auxiliary table and copies the name back to the location specified by the pointer; therefore, this area must be at least DBIMAXPATHLEN+1 bytes. If no auxiliary table is created, this area is set to all NULLs.

pszChangedName Type: pCHAR (Input)
Optional. Pointer to the Changed table name. A change table is created when DbiBatchMove is called with ebatMode being either batUPDATE or batAPPENDUPDATE. If NULL, no Changed table is created. If the user supplies a table name, that name is used. If not NULL and a pointer to a NULL character is specified, BDE generates a name for the auxiliary table and copies the name back to the location specified by the pointer; therefore, this area must be at least DBIMAXPATHLEN+1 bytes. If no auxiliary table is created, this area is set to all NULLs.

p1ProbRecs Type: pUINT32 (Output)
Pointer to the client variable that receives the number of records that were added, or would have been added to the Problems table. (When *pszProblemsName* is NULL, the Problems table is not actually created. In that case, *p1ProbRecs* reports the number of records that would have been added to the Problems table.) Optional. If *p1ProbRecs* is NULL, the number of records is not returned.

p1KeyvRecs Type: pUINT32 (Output)
Pointer to the client variable that receives the number of records that were added, or would have been added to the Key Violations table. (If *pszKeyViolName* is NULL, the Key Violations table is not actually created. In that case, p1KeyvRecs reports the number of records that would have been added to the Key Violations table.) Optional. If *p1KeyvRecs* is NULL, the number of records is not returned.

p1ChangedRecs Type: pUINT32 (Output)
Pointer to the client variable that receives the number of records that were added, or would have been added to the Changed table. (If *pszChangedName* is NULL, the Changed table is not actually created. In that case, *p1ChangedRecs* reports the number of records that

would have been added to the Changed table.) Optional. If *p1ChangedRecs* is NULL, the number of records is not returned.

bAbortOnFirstProb Type: BOOL (Input)
Specifies whether to cancel as soon as a record is encountered that would be written to the Problems table. If TRUE, the operation is canceled and DBIERR_NONE is returned.

bAbortOnFirstKeyviol Type: BOOL (Input)
Specifies whether to cancel as soon as a record is encountered that would be written to the Key Violations table. If TRUE, the operation is canceled and DBIERR_NONE is returned.

p1RecsToMove Type: pUINT32 (Input/Output)
On input, *p1RecsToMove* specifies the number of records to be read from the source table. On output, pointer to the client variable that receives the actual number of records read from the source table. If *p1RecsToMove* contains 0 or p1RecsToMove is NULL, all of the records in the table are processed.

bTransliterate Type: BOOL (Input)
Specifies whether to transliterate character data from one character set to another, when the source and destination character sets differ. TRUE causes all data in character fields of the source table to be transliterated into the character set of the destination table.

Usage

Depending on the mode specified in *ebatMode*, DbiBatchMove can be used in the following ways:

Mode	Use
batAPPEND	Adds records from the source table to the destination table.
batUPDATE	Overwrites matching records in the destination table, which must have a unique index. (Records from the source table that don't match are not added.)
batAPPENDUPDATE	Adds non-matching records to the destination table, which must have a unique index, and overwrites matching records.
batSUBTRACT	Deletes matching records from the destination table, which must have a unique index.
batCOPY	Copies a table to a new table of a different driver type. This creates the destination table with a record structure that minimizes potential data loss. (See the following section for a description of the method by which field types are translated.)

Important: For *batAPPEND* and *batCOPY* no index is required on the destination table. For the other three mode options an index is required.

Where an index is required on the destination table, the index is used to find matching records.

When the source and destination record structures differ in the field size or type, data from the source table is converted to the size or type of the destination table. If the conversion is not allowed, an error is returned and no data is transferred.

As each destination record is constructed, the default behavior is to trim any data that does not fit, possibly producing a NULL value in the destination. To override this default behavior, the client must register a callback of type cbBATCHRESULT with a client-allocated callback buffer CBRESTCbDesc (the same structure as is used for DbiDoRestructure). Before data transfer begins a callback is made for each pair of source and destination fields that could result in data loss. During this callback, RESTCBDesc.iErrCode is set to DBIERR_OBJMAYBETRUNCATED, RESTCBDesc.eRestrObjType is set to restrNEWFLD, RESTCBDesc.iObjNum is set to the field number of the destination field, and RESTCBDesc.uObjDesc.fldDesc contains the destination FLDDesc. If the client returns

cbrYES from the callback, this field is trimmed. If cbrNO is returned, then any records that would be trimmed are written to the problems table instead of the destination. If any one field is marked for no trimming and the data must be trimmed, the entire record is written to the Problems table.

You can adjust the size of a batch to accommodate server transaction logs that are not big enough to handle the whole batch. Set the database property dbBATCHCOUNT for the number of records you want to include in a batch before an auto-commit occurs. The value you set will override the default and take effect for all subsequent calls of DbiBatchMove. You can reset the default (32 Kb) by using the BDE Administrator to change the Database Configuration option BATCH COUNT.

Note: When using a Sybase or Microsoft backend, you cannot do a DbiBatchMove using *eBATMode* set to batCOPY to copy a table containing a memo field larger than 32K. This is a server limitation. You should create the new table, add a unique index, and then use DbiBatchMove with *eBATMode* set to batAPPEND instead.

Oracle8: Not supported for object types (ADT, REF, nested table, and VARARRAY).

Prerequisites

If cursors are not passed in, this call acquires a read lock on the source and a write lock on the destination. If cursors are passed in, the client is responsible for controlling locking behavior.

Completion state

If the function is called within the context of a transaction on the destination database handle, it does not modify the transaction.

DbiResult return values

DBIERR_NONE	The operation was performed successfully.
DBIERR_INVALIDPARAM	Either the source or the destination table identification is invalid.
DBIERR_INVALIDFILENAME	The source table name provided is an empty string.

See also

DbiOpenTable, DbiCreateTable, DbiRegisterCallBack, DbiDoRestructure

C Examples: DbiBatchMove

Create and copy table information from a dBASE or FoxPro table to a Paradox table.

Source table must be a dBASE or FoxPro table. This example uses the following input:
fDbiBatchMove(hDb, &hPXCur, hdBASECur);

```
DBIResult fDbiBatchMove(hDBIDb hTmpDb, phDBICur phDestCur, hDBICur hSrcCur)
{
   DBIResult    rslt;
   CURProps     CurProps;
   BATTblDesc   TblDesc;

   // Get source cursor properties
   rslt = Chk(DbiGetCursorProps(hSrcCur, &CurProps));
   if (rslt != DBIERR_NONE)
      return rslt;
   TblDesc.hDb = hTmpDb;
   strcpy(TblDesc.szTblName, CurProps.szName);
   strcpy(TblDesc.szTblType, szPARADOX);

   // Delete table if it exists
   rslt = DbiDeleteTable(hTmpDb, CurProps.szName, szPARADOX);
   if ((rslt != DBIERR_NOSUCHFILE) && (rslt != DBIERR_NOSUCHTABLE))
      Chk(rslt);


   // Copy the information from the dBASE table to the Paradox table
   rslt = Chk(DbiBatchMove(NULL, hSrcCur, &TblDesc, NULL, batCOPY,
                       0, 0, NULL, NULL, NULL, "KEYVIOL", "PROBLEMS",
  "CHANGED",
                       NULL, NULL, NULL, FALSE, FALSE, NULL, FALSE));
   if (rslt != DBIERR_NONE)
      return rslt;


   // Open the newly created Paradox table
   rslt = Chk(DbiOpenTable(hTmpDb, CurProps.szName, szPARADOX, NULL, NULL,
  0,
                        dbiREADWRITE, dbiOPENSHARED, xltFIELD, FALSE, NULL,
  phDestCur));

   return rslt;
}
```

Delphi Examples: DbiBatchMove

Create and copy table information from one table to another.

Because the DBIBatchMove function is a complex function that accepts over 20 parameters, Delphi users are often better served by using the VCL component TBatchMove. This component is found on the Data Access page of the Component Palette.

To use the TBatchMove component in your application, set the component's Mode, Mapping, Source, and Destination properties. You can then execute the batch move at design time by right clicking on the component and choosing Execute from the Speed Menu. Or you can execute the function at runtime by calling its Execute method. The following example illustrates using the component at runtime by appending TTables Table1 to Table2:

```
procedure TForm1.Button1Click(Sender : TObject);
begin
  with BatchMove1 do begin
    Mode := batAppend;
    Source := Table1;
    Destination := Table2;
    Execute;
  end;
end;
```

Append the source handle's table records to the destination handle's table:

BatchCount is the amount of records to append before committing the transaction. Most Delphi users should use the TBatchMove component or the TTable.BatchMove method. This example uses the following input:

```
  fDbiBatchMove(VendorTbl.Handle, VendorIBTbl.Handle, 500);
```

The procedure is defined as:

```
procedure fDbiBatchMove(hSrcCur, hDstCur: hDBICur; BatchCount: Longint);
var
  hDb: hDBIDb;
  Count: Longint;
  Length: Word;
  DBType: string;
begin
  Count := 0;
  // Get the database handle from the destination cursor
  Check(DbiGetObjFromObj(hDBIObj(hDstCur), objDATABASE, hDBIObj(hDb)));
  SetLength(DBType, DBIMAXNAMELEN);
  // Get the database type from the database handle
  Check(DbiGetProp(hDBIObj(hDb), dbDATABASETYPE, PChar(DBType),
  DBIMAXNAMELEN,
    Length));
  SetLength(DBType, StrLen(PChar(DbType)));
  if DBType <> 'STANDARD' then
    // If the database is SQL based, set the batch count
    Check(DbiSetProp(hDBIObj(hDb), dbBATCHCOUNT, BatchCount));
  // Append the records
  Check(DbiBatchMove(nil, hSrcCur, nil, hDstCur, batchAPPEND, 0, nil, nil,
    nil, 0, nil, nil, nil, nil, nil, nil, True, True, Count, False));
end;
```

dbBATCHCOUNT

The database property, dbBATCHCOUNT, lets you control the number of records to include in a batch before an auto-commit occurs. In this way you can adjust the size of a batch to accommodate server transaction logs that are not big enough to handle the whole batch.

```
dbBATCHCOUNT //rw UINT16, batch mod count (records) before auto-commit
```

The value of the batch count property is obtained from the BDE DB OPEN section of the Windows Registry:

```
BATCH COUNT  // UINT16, batch mod count (records) before auto-commit
```

If this option is not present or is zero then the default is the number of records that will fit in 32k bytes. You can override this value by setting dbBATCHCOUNT and the new value will take effect for all subsequent DbiBatchMove calls. When DbiBatchMove is called in an explicit client transaction (inside a DbiBeginTran/DbiEndTran block), the value of batch count is ignored.

DbiBcdFromFloat {button C Examples,JI(`>example',`exdbibcdfromfloat')} {button Delphi Examples,JI(`>example',`dexdbibcdfromfloat')}

C syntax

```
DBIResult DBIFN DbiBcdFromFloat (piVal, iPrecision, iPlaces, pBcd);
```

Delphi syntax

```
function DbiBcdFromFloat (var iVal: Double; iPrecision: Word; iPlaces: Word;
  var Bcd: FMTBcd): DBIResult stdcall;
```

Description

DbBcdFromFloat converts a number in the BDE logical FLOAT format into the BDE logical binary coded decimal (BCD) format.

Parameters

piVal Type: pDFLOAT (Input)
Specifies the FLOAT data to convert.

iPrecision Type: UINT16 (Input)
Specifies the precision of the BCD number. This number must be 32 for Paradox fields and usually the *iUnits1* value for other drivers.

iPlaces Type: UINT16 (Input)
Specifies the number of decimals of the BCD number.

pBcd Type: pFMTBcd (Output)
Pointer to the client buffer that receives the BCD number (FMTBcd). The BDE logical BCD format has a length which equals (*iPrecision* 2).

Usage

Due to a lack of high precision support for floating point values on the Intel platform, using the functions DbiBcdFromFloat and DbiBcdToFloat to convert values with more than 14 significant digits is not recommended. To convert high precision values on the Intel platform, develop a manual conversion routine that uses the information in the FMTBcd structure.

See Also

DbiBcdToFloat

C Examples: DbiBcdFromFloat

An example for this function is under development and will be provided in an upcoming Help release.

Delphi Examples: DbiBcdFromFloat

An example for this function is under development and will be provided in an upcoming Help release.

DbiBcdToFloat {button C Examples,JI(`>example',`exdbibcdtofloat')} {button Delphi Examples,JI(`>example',`dexdbibcdtofloat')}

C syntax

```
DBIResult DBIFN DbiBcdToFloat (pBcd, piVal);
```

Delphi syntax

```
function DbiBcdToFloat (var Bcd: FMTBcd; var iVal: Double): DBIResult
  stdcall;
```

Description

DbiBcdToFloat converts a number in the BDE logical binary coded decimal (BCD) format to the BDE FLOAT format.

Parameters

pBcd Type: pFMTBcd (Input)
Pointer to a FMTBcd structure that specifies the binary coded decimal (BCD) data to convert.

piVal Type: pDFLOAT (Output)
Pointer to the client buffer that receives the FLOAT number.

Usage

Due to a lack of high precision support for floating point values on the Intel platform, using the functions DbiBcdFromFloat and DbiBcdToFloat to convert values with more than 14 significant digits is not recommended. To convert high precision values on the Intel platform, develop a manual conversion routine that uses the information in the FMTBcd structure.

See Also

DbiBcdFromFloat

C Examples: DbiBcdToFloat

An example for this function is under development and will be provided in an upcoming Help release.

Delphi Examples: DbiBcdToFloat

An example for this function is under development and will be provided in an upcoming Help release.

DbiBeginDelayedUpdates {button C Examples,JI(`>example',`exdbibegindelayedupdates')} {button Delphi Examples,JI(`>example',`dexdbibegindelayedupdates')}

C syntax
```
DBIResult DBIFN DbiBeginDelayedUpdates (phCursor);
```

Delphi syntax
```
function DbiBeginDelayedUpdates (var hCursor: hDBICur): DBIResult stdcall;
```

Description
DbiBeginDelayedUpdates converts a cursor to a cached updates cursor. Given an open cursor, prepares for cached updates. Returns a new cursor; the old cursor is no longer valid.

Parameters
phCursor Type: phDBICur (Input/Output)
On input, specifies the original cursor. On output, returns the new cursor; the old cursor is no longer valid.

Usage
Use DbiBeginDelayedUpdates to activate the cached updates mode. This feature lets you retrieve data from a table and make changes to that temporarily cached data without immediately writing to the actual underlying table. You can make changes over a prolonged period with a minimum amount of resource locking at the actual database. After modifying the data, call DbiApplyDelayedUpdates to save changes in the actual table.

Prerequisites
None

Completion state
The record buffer (RecBuffsize) and the bookmark (BookMarksize) increase in size upon completion of DbiBeginDelayedUpdates. You should call DbiGetCursorProps and reallocate memory properties for bookmark and record buffers accordingly.

DbiResult return values
DBIERR_NONE The cached updates mode was initiated and the new cached updates cursor handle was successfully created.

See also
DbiEndDelayedUpdates, DbiApplyDelayedUpdates, Cached updates

C Examples: DbiBeginDelayedUpdates

An example for this function is under development and will be provided in an upcoming Help release.

Delphi Examples: DbiBeginDelayedUpdates

An example for this function is under development and will be provided in an upcoming Help release.

DbiBeginLinkMode {button C Examples,JI(`>example',`exdbibeginlinkmode')} {button Delphi Examples,JI(`>example',`dexdbibeginlinkmode')}

C syntax

```
DBIResult DBIFN DbiBeginLinkMode (phCursor);
```

Delphi syntax

```
function DbiBeginLinkMode (var hCursor: hDBICur): DBIResult stdcall;
```

Description

DbiBeginLinkMode converts a cursor to a link cursor. Given an open cursor, prepares for linked access. Returns a new cursor; the old cursor is no longer valid.

Parameters

phCursor Type: phDBICur (Input/Output)
On input, specifies the original cursor. On output, returns the new cursor; the old cursor is no longer valid.

Usage

Enables linking between tables using DbiLinkDetail. Both master and detail cursors must be link-enabled before calling DbiLinkDetail. DbiEndLinkMode must be called to end Link mode before the cursor is closed.

Warning: Using the original cursor (supplied as input) will result in an error when used with any BDE calls.

DbiResult return values

DBIERR_NONE The cursor was successfully converted to a linked cursor.

See also

DbiEndLinkMode, DbiLinkDetail, DbiLinkDetailToExp, DbiUnlinkDetail, DbiGetLinkStatus

C Examples: DbiBeginLinkMode

An example for this function is under development and will be provided in an upcoming Help release.

Delphi Examples: DbiBeginLinkMode

Create a master/detail link between two tables. The two hDBICur cusrors return linked handles to both tables. Tables must be open on the indexes to link on. This example only supports a 'full index' link; if the link is on a composite index, both tables must link on the complete index. This example uses the following input:

```
  fDbiBeginLinkMode(CustomerTbl, OrdersTbl, hMas, hDet);
```

The procedure is defined as:

```
procedure fDbiBeginLinkMode(MasTbl, DetTbl: TTable; var hMasCur,
  hDetCur: hDBICur);
var
  MasIdxDesc, DetIdxDesc: IDXDesc;
begin
  Check(DbiCloneCursor(MasTbl.Handle, False, False, hMasCur));
  Check(DbiCloneCursor(DetTbl.Handle, False, False, hDetCur));
  Check(DbiGetIndexDesc(hMasCur, 0, MasIdxDesc));
  Check(DbiGetIndexDesc(hDetCur, 0, DetIdxDesc));
  Check(DbiSetToBegin(hMasCur));
  Check(DbiBeginLinkMode(hMasCur));
  Check(DbiBeginLinkMode(hDetCur));
  Check(DbiLinkDetail(hMasCur, hDetCur, MasIdxDesc.iFldsInKey,
    @MasIdxDesc.aiKeyFld, @DetIdxDesc.aiKeyFld));
end;
```

DbiBeginTran {button C Examples,JI(`>example',`exdbibegintran')} {button Delphi Examples,JI(`>example',`dexdbibegintran')}

C syntax

```
DBIResult DBIFN DbiBeginTran (hDb, eXIL, phXact);
```

Delphi syntax

```
function DbiBeginTran (hDb: hDBIDb; eXIL: eXILType; var hXact: hDBIXact):
  DBIResult stdcall;
```

Description

DbiBeginTran begins a transaction on SQL server tables or local (Paradox, dBASE, FoxPro, Access) tables.

Parameters

hDb Type: hDBIDb (Input)
Specifies the database handle.

eXIL Type: eXILType (Input)
Specifies the transaction isolation level.

phXact Type: phDBIXact (Output)
Pointer to the transaction handle.

Usage

This function begins a transaction on the given database. Within a transaction, operations are not committed automatically, giving the client control over transaction behavior. The transaction remains active until a call to DbiEndTran is made to end the transaction.

Some servers do not allow Data Definition Language (DDL) statements within a transaction, or implicitly commit the transaction when a DDL statement is issued. For such servers, DDL operations are not allowed within a transaction. If table lock release requests cause implicit commits, a request for a table lock release is held until the transaction is ended.

Servers vary in the availability and behavior of isolation and read repeatability capabilities. Some SQL drivers support only the server default isolation level. To check the isolation level actually used, call DbiGetTranInfo after a successful call to DbiBeginTran.

Nested transactions are not supported. If a previously requested transaction is still active, this function returns an error.

For local transactions, only xlDIRTYREAD is supported.

Prerequisites

A valid database handle must be obtained from a server.

DbiResult return values

DBIERR_NONE The transaction has begun successfully.

DBIERR_ACTIVETRAN There is already an active transaction.

C Examples: DbiBeginTran

Start a transaction on the specified database.

This example uses the following input:

fDbiBeginTran(hDb, xilDIRTYREAD, &xTran);

```
DBIResult fDbiBeginTran(hDBIDb hTmpDb, eXILType xType, phDBIXact phXact)
{
   DBIResult   rslt;
   rslt = Chk(DbiBeginTran(hTmpDb, xType, phXact));
   return rslt;
}
```

Delphi Examples: DbiBeginTran

Start a transaction on the specified database.

Delphi users should use the TDataBase.StartTransaction method rather than directly calling DbiBeginTran. This method is defined as:

```
procedure TDataBase.StartTransaction;
```

The following code begins a transaction on a TDataBase object called DataBase1 at the isolation level specified by the TransIsolation property:

```
DataBase1.StartTransaction;
```

(Note: If a transaction is currently active, Delphi will raise an exception.)

Start a transaction on a database.

If the database is local, set the transaction isolation level to 'Dirty Read' (the only supported local isolation level). Most Delphi users should use TDatabase.StartTransaction. This example uses the following input:

```
fDbiBeginTran(Database1.Handle, xilREADCOMMITTED, hTran);
```

The procedure is defined as:

```
procedure fDbiBeginTran(hTmpDb: hDBIDb; Mode: eXILType; var hXact:
  hDBIXact);
var
  DBType: string;
  W: Word;
begin
  SetLength(DBType, DBIMAXNAMELEN);
  Check(DbiGetProp(hDBIObj(hTmpDb), dbDATABASETYPE, PChar(DBType),
    DBIMAXNAMELEN, W));
  SetLength(DBType, StrLen(PChar(DBType)));
  // If the transaction is on a local table, make sure it is set to Dirty
  Read
  if (DBType = 'STANDARD') then
    Mode := xilDIRTYREAD;
  Check(DbiBeginTran(hTmpDb, Mode, hXact));
end;
```

DbiCheckRefresh {button C Examples,JI(`>example',`exdbicheckrefresh')} {button Delphi Examples,JI(`>example',`dexdbicheckrefresh')}

C syntax

```
DBIResult DBIFN DbiCheckRefresh (VOID);
```

Delphi syntax

```
function DbiCheckRefresh: DBIResult stdcall;
```

Description

DbiCheckRefresh checks for remote updates to tables for all cursors in the current session, and refreshes the cursors if changed.

Usage

DbiCheckRefresh is useful for implementing an auto-refresh function that periodically refreshes client data. It can be called when a specified time period for the client process auto-refresh timer has elapsed. To receive a notification on the cursors that were actually refreshed, install a callback of the type cbTABLECHANGED.

SQL: This function is not operational with SQL drivers.

DbiResult return values

DBIERR_NONE All cursors in the current session have been successfully refreshed.

See also

DbiForceReread, DbiRegisterCallBack

C Examples: DbiCheckRefresh

Refresh all cursors in the current session.

This example uses the following input:

fDbiCheckRefresh();

```
DBIResult fDbiCheckRefresh(VOID)
{
   DBIResult   rslt;
   rslt = Chk(DbiCheckRefresh());
   return rslt;
}
```

Delphi Examples: DbiCheckRefresh

Refresh all sessions within the application, if needed. This is a good function to place in a TTimer to refresh data. This example uses the following input:

```
  fDbiCheckRefresh;
```

The procedure is defined as:

```
procedure fDbiCheckRefresh;
var
  OldSession: TSession;
  B: Byte;
begin
  OldSession := Sessions.CurrentSession;
  for B := 0 to (Sessions.Count - 1) do begin
    Sessions.CurrentSession := Sessions.Sessions[B];
    Check(DbiCheckRefresh);
  end;
  Sessions.CurrentSession := OldSession;
end;
```

DbiCloneCursor {button C Examples,JI(`>example',`exdbiclonecursor')} {button Delphi Examples,JI(`>example',`dexdbiclonecursor')}

C syntax

```
DBIResult DBIFN DbiCloneCursor (hCurSrc, bReadOnly, bUniDirectional,
  phCurNew);
```

Delphi syntax

```
function DbiCloneCursor (hCurSrc: hDBICur; bReadOnly: Bool; bUniDirectional:
  Bool; var hCurNew: hDBICur): DBIResult stdcall;
```

Description

DbiCloneCursor creates a new cursor (cloned cursor) that is similar to the given cursor (source cursor).

Parameters

hCurSrc Type: hDBICur (Input)
Specifies the cursor handle of the source cursor.

bReadOnly Type: BOOL (Input)
Specifies whether the cloned cursor access mode is to be read-only or read-write. TRUE specifies read-only and FALSE specifies read-write.

The client is able to choose the access mode of the cloned cursor only if the access mode of the source cursor is dbiREADWRITE. If the access mode of the source cursor is dbiREADONLY, then the access mode of the cloned cursor must be read-only.

bUnidirectional Type: BOOL (Input)

Specifies whether the cloned cursor movement is unidirectional or bidirectional (applies to SQL tables only). TRUE specifies unidirectional; FALSE specifies bidirectional.

Generally, bidirectional movement is preferable. However, if the client application knows that the cloned cursor is to access data solely from beginning to end, unidirectional movement might deliver better performance.

The client is able to choose the type of cursor movement for the cloned cursor only if the source cursor's *bUnidirectional* parameter is FALSE (bidirectional). If the source cursor's *bUnidirectional*parameter is TRUE (unidirectional), the cloned cursor can be only unidirectional.

phCurNew Type: phDBICur (Output)
Pointer to the cursor handle for the cloned cursor.

Usage

DbiCloneCursor provides the client a relatively quick way to get a cursor for a table that is already opened. The source cursor can be opened on a table or a query. The cloned cursor can then be used as a regular cursor, inheriting certain properties from the source cursor, but remaining completely independent in terms of position and ordering.

The cloned cursor inherits the following properties from the source cursor:

- Current index
- Range
- Translate mode
- Share mode
- Position
- Field maps
- Filters

Putting a field map or a filter on a cloned cursor does not affect the source cursor. The filters of a cloned cursor do not have the same filter handles as the original cursor, however, the filter ID (obtained with DbiGetFilterInfo) is invariant to the clone. This can be used to obtain the new filter handle for a given filter.

Positional commands (for example, DbiGetNextRecord) performed on the source cursor have no effect on the cloned cursor and vice versa.

dBASE and FoxPro: All indexes open on the source cursor are open on the clone.

Access: A cursor that references a table that is opened exclusively cannot be cloned.

Completion state

The returned cursor inherits certain properties from the source cursor but is completely independent in terms of position and ordering. The cloned cursor must be closed separately.

DbiResult return values

DBIERR_NONE	The cloned cursor was created successfully.
DBIERR_CURSORLIMIT	The maximum number of cursors has been exceeded.
DBIERR_INVALIDHNDL	The specified source cursor handle is invalid or NULL, or the pointer to the new cursor handle is NULL.

See also

DbiOpenTable

bReadOnly

The following table illustrates the effect that the access mode of the source cursor has on the cloned cursor access mode:

Source cursor	***bReadOnly***	**Cloned cursor**
Read-only	TRUE	Read-only
Read-only	FALSE	Read-only
Read-write	TRUE	Read-only
Read-write	FALSE	Read-write

bUnidirectional

The following table lists the effect of the source cursor's direction on the cloned cursor's direction:

Source direction	bUniDirectional	Cloned direction
Unidirectional	TRUE	Unidirectional
Unidirectional	FALSE	Unidirectional
Bidirectional	TRUE	Unidirectional
Bidirectional	FALSE	Bidirectional

C Examples: DbiCloneCursor

Clone (copy) a cursor and set the specified index. This example uses the following input:
ffDbiCloneCursor(hOrderCur, &hNewCur, "Customer No");

```
DBIResult fDbiCloneCursor(hDBICur hTmpCurSrc, phDBICur phTmpCurNew, pCHAR
  IndexName)
{
   DBIResult rslt;

   rslt = Chk(DbiCloneCursor(hTmpCurSrc, FALSE, FALSE, phTmpCurNew));
   if (rslt != DBIERR_NONE)
      return rslt;

   rslt = Chk(DbiSwitchToIndex(phTmpCurNew, IndexName, NULL, 0, FALSE));

   return rslt;
}
```

Delphi Examples: DbiCloneCursor

Return a new cursor positioned at the first record.

This example uses the following input:

```
  fDbiCloneCursor(Table1.Handle, MyNewCursor);
```

The procedure is defined as:

```
procedure fDbiCloneCursor(hTmpCur: hDBICur; var hNewCur: hDBICur);
begin
  Check(DbiCloneCursor(hTmpCur, False, False, hNewCur));
  Check(DbiSetToBegin(hNewCur));
  Check(DbiGetNextRecord(hNewCur, dbiNOLOCK, nil, nil));
end;
```

DbiCloseCursor{button C Examples,JI(`>example',`exdbiclosecursor')} {button Delphi Examples,JI(`>example',`dexdbiclosecursor')}

C syntax
```
DBIResult DBIFN DbiCloseCursor (phCursor);
```

Delphi syntax
```
function DbiCloseCursor (var hCursor: hDBICur): DBIResult stdcall;
```

Description
DbiCloseCursor closes a cursor.

Parameters
phCursor Type: phDBICur (Input)
Pointer to the cursor handle to be closed.

Usage
This function can be used to close all types of cursors. For temporary tables, DbiCloseCursor removes the table from memory.

If the cursor closed is the last remaining cursor for the table in the current session, then all locks acquired with DbiAcqTableLock are released.

If the given cursor is valid, the cursor is closed even if an error message is returned. Any error returned is to inform the client of a potential problem (for example, a network problem).

Completion state
All resources associated with the cursor are released, including record locks, filters, and all indexes that have been opened by DbiOpenIndex for that particular cursor. The cursor handle is invalid after DbiCloseCursor is called (even if an error, such as a network problem, occurs).

DbiResult return values

DBIERR_NONE	The table cursor was successfully closed.
DBIERR_INVALIDHNDL	The specified cursor handle is invalid or NULL.
DBIERR_NODISKSPACE	Table could not be saved to disk due to lack of space.

See also
DbiOpenTable, DbiCreateTempTable, DbiCreateInMemTable, DbiQExec, DbiQExecDirect, DbiOpenTableList, DbiOpenFileList, DbiOpenIndexList, DbiOpenFieldList, DbiOpenVchkList, DbiOpenRintList, DbiOpenSecurityList, DbiOpenFamilyList, DbiCloneCursor, DbiCloseDatabase

C Examples: DbiCloseCursor

Close the specified table cursor:

If the cursor is not an open cursor, the function exits. This example uses the following input:

fDbiCloseCursor(&hCur);

```
DBIResult fDbiCloseCursor (phDBICur phTmpCur)
{
   DBIResult        rslt = DBIERR_NONE;
   if (*phTmpCur != 0)
      rslt = Chk(DbiCloseCursor(phTmpCur));
   return rslt;
}
```

Delphi Examples: DbiCloseCursor

Close the valid cursor passed in:

If you have opened a cursor with a dbi call, then this example applies. Otherwise, use the Close method of a Delphi TDataSet descendent component.

This example uses the following input:

```
  fDbiCloseCursor(hCursor);
```

The procedure is defined as:

```
procedure fDbiCloseCursor(phTmpCur: phDBICur);
begin
  check(DbiCloseCursor(phTmpCur));
end;
```

DbiCloseDatabase {button C Examples,JI(`>example',`exdbiclosedatabase')} {button Delphi Examples,JI(`>example',`dexdbiclosedatabase')}

C syntax

```
DBIResult DBIFN DbiCloseDatabase (phDb);
```

Delphi syntax

```
function DbiCloseDatabase (var hDb: hDBIDb): DBIResult stdcall;
```

Description

DbiCloseDatabase closes a database and all cursors associated with the database handle.

Parameters

phDb Type: phDBIDb (Input)
Pointer to the database handle returned by DbiOpenDatabase.

Usage

DbiCloseDatabase releases the provided database handle and any associated cursors.

When closing the standard database handle with DbiCloseDatabase, all dBASE, FoxPro, Access, Paradox, and Text tables previously opened within this database are closed and the associated resources released.

SQL: Each database represents one or more connections to a specific SQL server. Closing the database closes those connections as well as releases other client database resources that have been acquired.

Prerequisites

DbiInit and DbiOpenDatabase must be called before a valid database handle is available.

Completion state

The client handle, *phDb*, is set to NULL.

DbiResult return values

DBIERR_NONE The database specified by phDb was closed successfully.

DBIERR_INVALIDHNDL The specified database handle is invalid or NULL.

See also

DbiOpenDatabase, DbiExit, DbiCloseCursor

C Examples: DbiCloseDatabase

Close the database associated with the valid handle passed in:

This example uses the following input:

fDbiCloseDatabase(&hDb);

```
DBIResult fDbiCloseDatabase(phDBIDb phTmpDb)
{
   DBIResult        rslt = DBIERR_NONE;
   if (*phTmpDb != 0)
      rslt = Chk(DbiCloseDatabase(phTmpDb));
   return rslt;
}
```

Delphi Examples: DbiCloseDatabase

Close the database associated with the valid handle passed in:

Delphi users should call TDatabase.Close rather than directly calling DbiCloseDatabase. This method is defined as:

```
procedure TDatabase.Close;
```

The following code closes a TDatabase component called MyDatabase:

```
MyDatabase.Close;
```

Close the database associated with the handle:

Most Delphi users should use TDatabase.Close. This example uses the following input:

```
fDbiCloseDatabase(hDb);
```

```
procedure fDbiCloseDatabase(var hTmpDb: hDBIDb);
begin
  Check(DbiCloseDatabase(hTmpDb));
end;
```

DbiCloseFieldXlt {button C Examples,JI(`>example',`exdbiclosefieldxlt')} {button Delphi Examples,JI(`>example',`dexdbiclosefieldxlt')}

C syntax
```
DBIResult DBIFN DbiCloseFieldXlt (hXlt);
```

Delphi syntax
```
function DbiCloseFieldXlt (hXlt: hDBIXlt): DBIResult stdcall;
```

Description
DbiCloseFieldXlt closes a field translation object.

Parameters
hXlt Type: hDBIXlt (Input)
Specifies the field translation handle.

DbiResult return values
DBIERR_NONE The translation object was closed successfully.

DBIERR_INVALIDHNDL The specified translation handle is invalid.

See also
DbiOpenFieldXlt, DbiTranslateField

C Examples: DbiCloseFieldXlt

An example for this function is under development and will be provided in an upcoming Help release.

Delphi Examples: DbiCloseFieldXlt

An example for this function is under development and will be provided in an upcoming Help release.

DbiCloseIndex {button C Examples,JI(`>example',`exdbicloseindex')} {button Delphi Examples,JI(`>example',`dexdbicloseindex')}

C syntax

```
DBIResult DBIFN DbiCloseIndex (hCursor, pszIndexName, iIndexId);
```

Delphi syntax

```
function DbiCloseIndex (hCursor: hDBICur; pszIndexName: PChar; iIndexId:
  Word): DBIResult stdcall;
```

Description

DbiCloseIndex closes the specified index for this cursor.

Parameters

hCursor Type: hDBICur (Input)
Specifies the cursor handle.

pszIndexName Type: pCHAR (Input)
Specifies the pointer to the index name. *pszIndexName* cannot be the name of the current active index of the cursor or a production index.

iIndexId Type: UINT16 (Input)
Currently not used.

Usage

DbiCloseIndex is applicable only with dBASE and FoxPro tables. It is used primarily to manipulate non-production indexes. DbiCloseIndex cannot close a current index, or a production index. To close a current index, DbiSwitchToIndex must be called first, to make another index (or no index) current.

This function does not affect the order of the records or the current position of the cursor.

Prerequisites

The index must be open.

Completion state

Once a production index is closed, it is no longer maintained.

DbiResult return values

DBIERR_NONE	The index was successfully closed.
DBIERR_NA	Operation is not applicable.
DBIERR_INVALIDHNDL	The specified cursor handle is invalid or NULL.
DBIERR_CANNOTCLOSE	The given index is a production index and must stay open.
DBIERR_ACTIVEINDEX	The given index is currently used by the cursor to order the result set.
DBIERR_NOSUCHINDEX	The given index is either not opened or no such index exists for the table.

See also

DbiSwitchToIndex, DbiOpenTable, DbiOpenIndex

C Examples: DbiCloseIndex

An example for this function is under development and will be provided in an upcoming Help release.

Delphi Examples: DbiCloseIndex

Close the specified index for this cursor.

This example uses the following input:

```
  fDbiCloseIndex(Table2.Handle, 'SYMBOL');
```

```
procedure fDbiCloseIndex(hTmpCur: hDBICur; IndexName: string);
begin
  Check(DbiCloseIndex(hTmpCur, PChar(IndexName), 0));
end;
```

DbiCloseSession {button C Examples,JI(`>example',`exdbiclosesession')} {button Delphi Examples,JI(`>example',`dexdbiclosesession')}

C syntax
```
DBIResult DBIFN DbiCloseSession (hSes);
```

Delphi syntax
```
function DbiCloseSession (hSes: hDBISes): DbiResult stdcall;
```

Description
DbiCloseSession closes the session associated with the given session handle.

Parameters
hSes Type: hDBISes (Input)
Specifies the session handle.

Completion state
When a session is closed, all resources (database handles, cursors, table level locks, and record level locks) attached to the given session are released. Any buffers that BDE has allocated that are specific to the session are also released. If *hSes* is the session handle of the current session, the client application is set to the default session after DbiCloseSession is completed. The client application cannot close the default session without exiting the client.

DbiResult return values
DBIERR_NONE The session specified by *hSes* was closed successfully.

DBIERR_INVALIDSESHANDL The specified session handle is invalid or NULL, or the session has already been closed.

See also
DbiGetCurrSession, DbiSetCurrSession, DbiStartSession, DbiGetSysInfo, DbiGetSesInfo

C Examples: DbiCloseSession

Close the current session.

DbiCloseSession releases all session resources. This example uses the following input:
fDbiCloseSession();

```
DBIResult fDbiCloseSession(VOID)
{
   DBIResult   rslt;
   hDBISes     hSes;

   rslt = Chk(DbiGetCurrSession(&hSes));
   if (rslt != DBIERR_NONE)
      return rslt;

   rslt = Chk(DbiCloseSession(hSes));

   return rslt;
}
```

Delphi Examples: DbiCloseSession

Close the current session.

Delphi users should use TSession.Close rather than directly calling dbiCloseSession. The method TSession.Close is defined as:

```
  procedure TSession.Close;
```

The following code closes TSession Session:

```
  Session.Close;
```

Close the session associated with the handle.

Most Delphi users should use Session.Close or the TSession component. This example uses the following input:

```
  DbiCloseSession(hSes);
```

```
procedure fDbiCloseSession(hTmpSes: hDBISes);
begin
  Check(DbiCloseSession(hTmpSes));
end;
```

DbiCompareBookMarks {button C Examples,JI(`>example',`exdbicomparebookmarks')} {button Delphi Examples,JI(`>example',`dexdbicomparebookmarks')}

C syntax

```
DBIResult DBIFN DbiCompareBookMarks (hCur, pBookMark1, pBookMark2,
  pCmpBkmkResult);
```

Delphi syntax

```
function DbiCompareBookMarks (hCur: hDBICur; pBookMark1: Pointer;
  pBookMark2: Pointer; var CmpBkmkResult: Word): DBIResult stdcall;
```

Description

DbiCompareBookMarks compares the relative positions of two bookmarks associated with the cursor.

Parameters

hCur Type: hDBICur (Input)
Specifies the cursor handle.

pBookMark1 Type: pBYTE (Input)
Specifies the pointer to the first bookmark.

pBookMark2 Type: pBYTE (Input)
Specifies the pointer to the second bookmark.

pCmpBkmkResult Type: pCMPBkMkRslt (Output)
Pointer to the client variable that receives the comparison result.

Usage

Both bookmarks must be placed on cursors opened on the same table with the same order.

Note: Comparing bookmarks from cursors with different orders or that are unstable can lead to unpredictable results.

Prerequisites

Valid bookmarks must have been obtained with DbiGetBookMark.

DbiResult return values

DBIERR_NONE	Bookmarks were compared successfully.
DBIERR_INVALIDHNDL	The specified cursor is invalid or NULL.
DBIERR_INVALIDPARAM	At least one of the following parameters is NULL: *pBookMark1*, *pBookMark2*.
DBIERR_INVALIDBOOKMARK	Bookmarks are incompatible or corrupt.

See also

DbiGetCursorProps, DbiGetBookMark, DbiSetToBookMark

pCmpBkmkResult
Comparison results can be:

Result	Description
CMPLess	Bookmark1 is before Bookmark2 in the result set.
CMPEql	Bookmark1 is the same as Bookmark2.
CMPGtr	Bookmark1 is after Bookmark2 in the result set.
CMPKeyEql	Bookmark1 and Bookmark2 have the same key value. Used in cases involving non-unique keys when it is uncertain if two bookmarks represent the same record.

C Examples: DbiCompareBookMarks

Compare the relative locations of two different bookmarks in a table.

This example uses the following input:

fDbiCompareBookMarks(hPXCurEx, BookMark1, BookMark2);

```
DBIResult fDbiCompareBookMarks(hDBICur hCur, pBYTE pBk1, pBYTE pBk2)
{
   DBIResult   rslt;
   CMPBkMkRslt CmpRslt;
   Chk(DbiCompareBookMarks(hCur, pBk1, pBk2, &CmpRslt));
   return rslt;
}
```

Delphi Examples: DbiCompareBookMarks

Compare the relative positions of two bookmarks associated with the cursor.
See also the method GetBookmark associated with a TTable, TQuery, and TStoredProc. This example uses the following input:

```
  Compare := fdbiCompareBookMarks(Table1, BookMark25, BookMark50);
```

```
function fdBICompareBookMarks(DataSet: TDataSet; Bookmark1, Bookmark2:
  TBookmark): Integer;
var
  Compare: Integer;
begin
  Check(DbiCompareBookMarks(DataSet.Handle, Bookmark1, Bookmark2, Compare));
  Result:= Compare;
end;
```

DbiCompareKeys {button C Examples,JI(`>example',`exdbicomparekeys')} {button Delphi Examples,JI(`>example',`dexdbicomparekeys')}

C syntax

```
DBIResult DBIFN DbiCompareKeys (hCursor, pKey1, [pKey2], iFields, iLen,
  piResult);
```

Delphi syntax

```
function DbiCompareKeys (hCursor: hDBICur; pKey1: Pointer; pKey2: Pointer;
  iFields: Word; iLen: Word; var iResult: SmallInt): DBIResult stdcall;
```

Description

DbiCompareKeys compares two key values based on the current index of the cursor.

Parameters

hCursor Type: hDBICur (Input)
Specifies the cursor handle.

pKey1 Type: pBYTE (Input)
Pointer to the first key value. The key is assumed to be in physical format.

pKey2 Type: pBYTE (Input)
Pointer to the second key value. Optional. If *pKey2* is NULL, the key value is extracted from the current record. If the key is specified, it is assumed to be in physical format.

iFields Type: UINT16 (Input)
Specifies the number of fields to be used for composite keys. *iFields* and *iLen* together indicate how much of the key is to be used for matching. If both are 0, the entire key is used. If a match is required on a given field of the key, all the key fields preceding it in the composite key must also be supplied for a match. Only character fields can be matched for a partial key; all other field types must be fully matched.

For partial key matches, *iFields* must be equal to the number of key fields preceding (if any) the field being partially matched.

iLen Type: UINT16 (Input)
Specifies a partial length in the last field to be used for composite keys; works in conjunction with *iFields*. The last field of the composite key must be a character type if *iLen* not equal to 0.

piResult Type: pINT16 (Output)
Pointer to the client variable that receives the compared result.

Usage

This function is used to compare two key values. Keys can be obtained by using DbiExtractKey.

Prerequisites

There must be an active index.

DbiResult return values

DBIERR_NONE The key fields were compared successfully.

DBIERR_NOCURREC pKey2 is NULL and the current record is invalid.

See also

DbiExtractKey

piResult
The result can be one of the following values:

Result	Description
-1	pKey1 < pKey2
0	pKey1 = pKey2
1	pKey1 > pKey2

C Examples: DbiCompareKeys

An example for this function is under development and will be provided in an upcoming Help release.

Delphi Examples: DbiCompareKeys

An example for this function is under development and will be provided in an upcoming Help release.

DbiCopyTable {button C Examples,JI(`>example',`exdbicopytable')} {button Delphi Examples,JI(`>example',`dexdbicopytable')}

C syntax

```
DBIResult DBIFN DbiCopyTable (hDb, bOverwrite, pszSrcTableName,
  pszSrcDriverType, pszDestName);
```

Delphi syntax

```
function DbiCopyTable (hDb: hDBIDb; bOverWrite: Bool; pszSrcTableName:
  PChar; pszSrcDriverType: PChar; pszDestTableName : PChar): DBIResult
  stdcall;
```

Description

DbiCopyTable duplicates the source table, to a destination table.

Parameters

hDb Type: hDBIDb (Input)
Specifies the database handle.

bOverwrite Type: BOOL (Input)
Specifies whether to overwrite an existing destination table or not. If TRUE, the table is overwritten; if FALSE, an error is returned if the destination table already exists.

pszSrcTableName Type: pCHAR (Input)
Pointer to the name of the table to be copied. *pszSrcTblName* can include a file extension, in which case *pszSrcDriverType* is ignored.

pszSrcDriverType Type: pCHAR (Input)
Pointer to the driver type, when *pszTblName* specifies a table name without a file extension. Required with Paradox, dBASE, FoxPro, and Access tables if no table extension is specified in *pszSrcTableName*.

pszDestName Type: pCHAR (Input)
Pointer to the name of the destination table.

Usage

This function is used to copy tables of the same driver type. It cannot copy a table across databases or driver types. To transfer data from one database type to another, see DbiBatchMove.

Driver-specific rules must be followed in defining family members:

Access: All Access objects in the .MDB file that are associated with the table are copied.

dBASE and FoxPro: For dBASE and FoxPro tables, default family members include

- The table (usually ends with a .DBF extension)
- BLOB file (usually <*tablename*>.DBT or <*tablename*>.FPT)
- Production index (usually <*tablename*>.MDX or <*tablename*>.CDX)

Non-production indexes are not included in the default family.

Paradox: For Paradox tables, default family members include

- The table (<*tablename*>. DB)
- The BLOB file (<*tablename*>.MB)
- All indexes
- Any <tablename>. VAL file

If the table is encrypted and the master password is not available, the copy fails. See DbiAddPassword.

SQL: The DbiCopyTable function copies only the table itself. The indexes are not copied.

Oracle8: Not supported for object types (ADT, REF, nested table, and VARARRAY).

Prerequisites

A read lock is required on source dBASE, FoxPro, Access, and Paradox tables. For SQL tables, at least a READ (SELECT) privilege is required on the source table.

Completion state

The source table is copied to the destination table.

DbiResult return values

DBIERR_NONE	The table was successfully copied.
DBIERR_INVALIDHNDL	The specified database handle is invalid or NULL.
DBIERR_INVALIDPARAM	The source or destination table name was not specified.
DBIERR_INVALIDFILENAME	An empty string or invalid filename was specified for the source or destination table name.
DBIERR_FILEEXISTS	The table already exists, and bOverwrite specifies not to overwrite it.
DBIERR_FAMFILEINVALID	The family file is corrupt.
DBIERR_NOSUCHTABLE	The source table does not exist.
DBIERR_NOTSUFFTABLERIGHTS	The user does not have permission to delete the existing destination table (Paradox only).
DBIERR_NOTSUFFFAMILYRIGHTS	The user does not have rights to family members (Paradox only).
DBIERR_LOCKED	The table is locked by another user.

See also

DbiBatchMove

C Examples: DbiCopyTable

Copy a source table into a destination table.

If *pNewCur* is not NULL, DbiCopyTable returns a cursor handle from the newly created table. This example uses the following input:

fDbiCopyTable(hDb, "STOCK.DB", "NEWSTOCK.DB", &hNewCur);

```
DBIResult fDbiCopyTable(hDBIDb hTmpDb, pCHAR SourceTbl, pCHAR DestTbl,
  phDBICur pNewCur)
{
   DBIResult        rslt;
   rslt = Chk(DbiCopyTable(hTmpDb, TRUE, SourceTbl, NULL, DestTbl));
   if (rslt != DBIERR_NONE)
      return rslt;
   if (pNewCur != NULL)
      rslt = Chk(DbiOpenTable(hTmpDb, DestTbl, NULL, NULL, NULL, 0,
  dbiREADWRITE,
                  dbiOPENSHARED, xltFIELD, FALSE, NULL, pNewCur));
   return rslt;
}
```

Delphi Examples: DbiCopyTable

Copy a source table into a destination table.

If the destination table exists, it is overwritten.If the tables are Paradox, dBASE, FoxPro, or Access, you must supply a file extension. This example uses the following input:

```
  fDbiCopyTable(Table1.dbhandle, 'CUSTOMER.DB', 'CUSTOMER2.DB');
```

```
procedure fDbiCopyTable(hTmpDb: hDbiDb; SrcTableName: string; DestName:
  string);
begin
  Check(DbiCopyTable(hTmpDb, True, PChar(SrcTableName), nil,
  PChar(DestName)));
end;
```

DbiCreateInMemTable {button C Examples,JI(`>example',`exdbicreateinmemtable')} {button Delphi Examples,JI(`>example',`dexdbicreateinmemtable')}

C syntax

```
DBIResult DBIFN DbiCreateInMemTable (hDb, pszName, iFields, pfldDesc,
  phCursor);
```

Delphi syntax

```
function DbiCreateInMemTable (hDb: hDBIDb; pszName: PChar; iFields: Word;
  pfldDesc: pFLDDesc; var hCursor: hDBICur): DBIResult stdcall;
```

Description

DbiCreateInMemTable creates a temporary in-memory table.

Parameters

hDb Type: hDBIDb (Input)
Specifies the database handle.

pszName Type: pCHAR (Input)
Pointer to the table name.

iFields Type: UINT16 (Input)
Specifies the number of fields in the table.

pfldDesc Type: pFLDDesc (Input)
Pointer to an array of field descriptor (FLDDesc) structures.

phCursor Type: phDBICur (Output)
Pointer to the cursor handle.

Usage

Only logical BDE field types are supported by the in-memory table. Physical field types are not supported. The table is kept in memory if possible, but it could be swapped to disk if the table becomes too big. Maximum table size is 512M with a maximum record size of 16K with a maximum of 1024 fields. Logical Autoincrement and BLOB fields are not supported. Indexes are not supported. Records cannot be deleted(DbiDeleteRecord). DbiGetExactRecordCount (Delphi's TTable.RecordCount) is not supported; instead use DbiGetRecordCount.

Completion state

This function returns a cursor on the temporary table in *phCursor*. The table will be deleted when the cursor is closed.

DbiResult return values

DBIERR_NONE The table was created successfully.

DBIERR_NODISKSPACE The table could not be saved to disk due to lack of space.

See also

DbiCreateTempTable, DbiCreateTable

C Examples: DbiCreateInMemTable

Create an in-memory table with two fields, one alpha, and one numeric.

This example uses the following input:

fDbiCreateInMemTable(hDb, &hInMemCur);

```
DBIResult fDbiCreateInMemTable(hDBIDb hTmpDb, phDBICur phTmpCur)
{
   DBIResult    rslt;
   UINT16       NumFields;

   FLDDesc fldDesc[] = {
         { // Field 1 - ALPHA
            1, "MyAlpha", fldZSTRING, fldUNKNOWN, 10, 0,
            0, 0, 0, fldvNOCHECKS, fldrREADWRITE
         },
         {  // FIELD 2 - NUMERIC
            2, "MyNumber", fldFLOAT, fldUNKNOWN, 0, 0, 0,
            0, 0, fldvNOCHECKS, fldrREADWRITE
         }
   };
   NumFields = sizeof(fldDesc) / sizeof(FLDDesc);
   rslt = Chk(DbiCreateInMemTable(hTmpDb, "InMemTbl", NumFields, fldDesc,
                                  phTmpCur));
   return rslt;
}
```

Delphi Examples: DbiCreateInMemTable

Example 1: Create an in-memory table using a custom field descriptor.

```
procedure MakeInMemTable1;
const
  fldDes: array[0..2] of FLDDesc = (
    // Field 1 - ALPHA
    (iFldNum:      1;
     szName:       'ALPHA';
     iFldType:     fldZSTRING;
     iSubType:     fldUNKNOWN;
     iUnits1:      10;
     iUnits2:      0;
     iOffset:      0;
     iLen:         0;
     iNullOffset:  0;
     efldvVchk:    fldvNOCHECKS;
     efldrRights:  fldrREADWRITE),
    // Field 2 - NUMERIC
    (iFldNum:      2;
     szName:       'NUMERIC';
     iFldType:     fldFLOAT;
     iSubType:     fldUNKNOWN;
     iUnits1:      0;
     iUnits2:      0;
     iOffset:      0;
     iLen:         0;
     iNullOffset:  0;
     efldvVchk:    fldvNOCHECKS;
     efldrRights:  fldrREADWRITE),
    // Field 3 - SHORT
    (iFldNum:      3;
     szName:       'SHORT';
     iFldType:     fldINT16;
     iSubType:     fldUNKNOWN;
     iUnits1:      0;
     iUnits2:      0;
     iOffset:      0;
     iLen:         0;
     iNullOffset:  0;
     efldvVchk:    fldvNOCHECKS;
     efldrRights:  fldrREADWRITE));
var
  hIMcur   : hDBICur;
  hNilDB   : hDBIDb;
begin
  Check(dbiOpenDatabase(nil, nil, dbiREADWRITE, dbiOPENSHARED, nil, 0, nil,
    nil, hNilDB));
  Check(dbiCreateInMemTable(hNilDB, 'InMemTbl', 3, @fldDes, hIMCur));
end;
```

Example 2: Create an in-memory table by borrowing the field descriptor from an

existing Paradox table.
The existing table is passed in the Tbl parameter.

```
procedure MakeInMemTable2(Tbl: TTable);
var
  TblProps : CurProps;
  PFDesc   : pFldDesc;
  hIMcur   : hDBICur;
  MemSize  : Integer;
begin
  Check(dbiGetCursorProps(Tbl.Handle, TblProps));
  MemSize := SizeOf(FldDesc) * (TblProps.iFields);
  PFDesc := AllocMem(MemSize);
  try
    Check(dbiGetFieldDescs(Tbl.Handle, PFDesc));
    Check(dbiCreateInMemTable(Tbl.DBHandle, 'InMemTbl', 3, PFDesc, hIMCur));
  finally
    FreeMem(PFDesc, MemSize);
  end;
end;
```

DbiCreateTable {button C Examples,JI(`>example',`exdbicreatetable')} {button Delphi Examples,JI(`>example',`dexdbicreatetable')}

C syntax

```
DBIResult DBIFN DbiCreateTable (hDb, bOverWrite, pcrTblDsc);
```

Delphi syntax

```
function DbiCreateTable (hDb: hDBIDb; bOverWrite: Bool; var crTblDsc:
  CRTblDesc): DBIResult stdcall;
```

Description

DbiCreateTable creates a table in the database associated with the given database handle.

Parameters

hDb Type: hDBIDb (Input)
Specifies the database handle.

bOverWrite Type: BOOL (Input)
Specifies whether to overwrite an existing table or not. If TRUE is specified, and there is an existing table, it will be overwritten. If FALSE is specified, and there is an existing table, an error is returned.

pcrTblDsc Type: pCRTblDesc (Input)
Pointer to the table descriptor structure (CRTblDesc). Refer to DbiGetFieldTypeDesc and DbiGetIndexTypeDesc for more information on the legal values for these structures for each Borland Database Engine driver.

Optional parameters

The optional parameter fields *iOptParams*, *pfldOptParams*, and *pOptData* are used to set other driver-specific attributes of the table. These parameters are used to describe a single record that is constructed by the client and contains the null-terminated ASCII strings that specify the values for these driver-specific attributes.

iOptParams is the number of optional parameters. *pfldOptParams* contains a pointer to an array of FLDDesc of *iOptParams* size. Each of these field descriptors is given a field name equal to the name of the optional parameter (for example, MDXBLOCKSIZE) and has *iLen* and *iOffset* set to the length (including the NULL terminator) and position in the *pOptData* record buffer of the ASCII string containing the value of this parameter (for example, 512). All other elements of the FLDDesc are ignored. The *pOptData* record buffer need only be large enough to hold all the null-terminated strings for each optional parameter value. This style of setting optional parameters is also used by DbiOpenDatabase. The names of the optional parameters can be obtained using DbiOpenCfgInfoList with a configuration path of DRIVERS\DRIVERNAME\TABLECREATE.

Usage

The required descriptors are specified in CRTblDesc; different drivers might require different descriptors.

Text: DbiCreateTable can be used to create a text file to export the data to it. For text file creation, only *szTblName* and *szTblType* values in the CRTblDesc are used and the rest of the values are ignored (*szTblType* is specified as ASCIIDRV). A text file is created with the given name; no field descriptions are necessary.

Paradox: Referential integrity can be created only when creating or restructuring the detail table. The master table must already exist and must be in the same directory as the table being created. A lookup table may exist in any accessible directory, but must exist at the time this table is created.

SQL: All indexes are maintained; there are no non-maintained indexes.

Oracle8: Not supported for object types (ADT, REF, nested table, VARARRAY).

Integrity Constraints: When creating a table by using DbiCreateTable, you can use integrity constraints to ensure that references in the key fields of secondary tables (in the same database) or foreign tables (in another database) are maintained to key fields in a primary table. For example, if several tables have keys referencing primary key Customer ID in the Customer table, then this dependecy must be checked so that referenced customer IDs cannot be deleted, thereby orphaning records in secondary or foreign tables.

Prerequisites

If the client chooses to overwrite an existing table; the existing table must be closed. DbiCreateTable supports up to 255 val-checks.

Completion state

All files associated with the table are created.

DbiResult return values

DBIERR_NONE The table was created successfully.

DBIERR_INVALIDFILEEXTN	The driver type or file extension is invalid.
DBIERR_INVALIDOPTION	The index description is invalid.
DBIERR_INVALIDINDEXSTRUCT	Invalid index structure. For SQL servers, all indexes are maintained; verify that *bMaintained* in pidxDesc specifies TRUE.
DBIERR_FILEEXISTS	The table already exists (returned when *bOverWrite* is FALSE).
DBIERR_INVALIDHNDL	The specified database handle is invalid or NULL.
DBIERR_UNKNOWNTBLTYPE	The specified driver type is invalid.
DBIERR_MULTILEVELCASCADE	An illegal attempt was made to create a referential integrity link that is already in use as a link to a higher level cascade update (Paradox only).
DBIERR_FLDLIMIT	*iFldCount* exceeds maximum number of fields.
DBIERR_INVALIDFIELDNAME	An invalid field name was specified.
DBIERR_NAMENOTUNIQUE	The specified field name or index name is not unique.
DBIERR_INVALIDFLDTYPE	The specified field type is unknown or not allowed.
DBIERR_RECTOOBIG	The record size exceeds the maximum allowed.
DBIERR_INVALIDINDEXNAME	The specified index name is invalid.
DBIERR_INVALIDINDEXTYPE	The specified index type is invalid.
DBIERR_INDEXNAMEREQUIRED	No index name was specified.
DBIERR_LOOKUPTBLOPENERR	The specified lookup table could not be opened.

See also

DbiCopyTable, DbiSortTable, DbiDoRestructure

C Examples: DbiCreateTable

Example 1: Create a Paradox table:

This example uses the following input:
fDbiCreateTable1(hDb);

```
DBIResult fDbiCreateTable1(hDBIDb hTmpDb)
{
   CHAR szTblType[] = szPARADOX;
   CHAR szTblName[] = "PX_Table";
   CRTblDesc   TblDesc;    // Create Table Descriptor
   DBIResult rslt;
   UINT16 NumFields  = 2;
   FLDDesc fldDesc[] = {
            { // Field 1 - TIMESTAMP
               1, "MyAlpha", fldPDXDATETIME, fldUNKNOWN, 0, 0,
               0, 0, 0, fldvNOCHECKS, fldrREADWRITE
            },
            {  // FIELD 2 - AUTOINCREMENT
               2, "MyNumber", fldPDXAUTOINC, fldUNKNOWN, 0, 0, 0,
               0, 0, fldvNOCHECKS, fldrREADWRITE
            }
   };
   memset((void *) &TblDesc, 0, sizeof(CRTblDesc));
   lstrcpy(TblDesc.szTblName, szTblName);
   lstrcpy(TblDesc.szTblType, szTblType);
   TblDesc.iFldCount = NumFields;
   TblDesc.pfldDesc = fldDesc;
   //Could add indexes, validity checks, and security descriptors here.
   rslt = Chk(DbiCreateTable(hTmpDb, TRUE, &TblDesc));
   return rslt;
}
```

Example 2: Create a dBASE table:

This example uses the following input:
fDbiCreateTable2(hDb);

```
DBIResult fDbiCreateTable2(hDBIDb hTmpDb)
{
   CHAR szTblType[] = szDBASE;
   CHAR szTblName[] = "dBASE_TBL";
   CRTblDesc   TblDesc;    // Create Table Descriptor
   DBIResult rslt;
   UINT16 NumFields  = 2;
   FLDDesc fldDesc[] = {
            { // Field 1 - MEMO
               1, "MyAlpha", fldDBMEMO, fldUNKNOWN, 0, 0,
               0, 0, 0, fldvNOCHECKS, fldrREADWRITE
            },
            {  // FIELD 2 - BOOLEAN
               2, "MyNumber", fldDBBOOL, fldUNKNOWN, 0, 0, 0,
               0, 0, fldvNOCHECKS, fldrREADWRITE
            }
   };
   memset((void *) &TblDesc, 0, sizeof(CRTblDesc));
   lstrcpy(TblDesc.szTblName, szTblName);
   lstrcpy(TblDesc.szTblType, szTblType);
```

```
   TblDesc.iFldCount = NumFields;
   TblDesc.pfldDesc = fldDesc;
   //Could add indexes, validity checks, and security descriptors here.
   rslt = Chk(DbiCreateTable(hTmpDb, TRUE, &TblDesc));
   return rslt;
}
```

Example 3: Create a InterBase table:

This example uses the following input:

fDbiCreateTable3(hDb);

```
DBIResult fDbiCreateTable3(hDBIDb hTmpDb)
{
   CHAR szTblName[] = "IB_TBL";
   CRTblDesc   TblDesc;    // Create Table Descriptor
   DBIResult rslt;
   UINT16 NumFields  = 2;
   FLDDesc fldDesc[] = {
            { // Field 1 - MEMO
               1, "MyAlpha", fldIBCHAR, fldUNKNOWN, 300, 0,
               0, 0, 0, fldvNOCHECKS, fldrREADWRITE
            },
            {  // FIELD 2 - BLOB
               2, "MyNumber", fldIBBLOB, fldUNKNOWN, 0, 0, 0,
               0, 0, fldvNOCHECKS, fldrREADWRITE
            }
   };
   memset((void *) &TblDesc, 0, sizeof(CRTblDesc));
   lstrcpy(TblDesc.szTblName, szTblName);
   lstrcpy(TblDesc.szTblType, szTblType);
   TblDesc.iFldCount = NumFields;
   TblDesc.pfldDesc = fldDesc;
   //Could add indexes, validity checks, and security descriptors here.
   rslt = Chk(DbiCreateTable(hTmpDb, TRUE, &TblDesc));
   return rslt;
}
```

Delphi Examples: DbiCreateTable

Create a table with a different level, block size, and fill factor than specified in the BDE configuration. Most Delphi users should use TTable.CreateTable. This example uses the following input:

```
  fDbiCreateTable(Database1.Handle, 'TableChange', 3, @FDesc, 7, 32768, 95);
```

```
procedure fDbiCreateTable(hTmpDb: hDBIDb; TableName: string; Fields: Word;
  pFlds: pFLDDesc; Level, BlockSize, FillFactor: Word);
var
  pOptDesc, pOrigDesc: pFLDDesc;
  pOptData, pOrigData: pBYTE;
  TblDesc: CRTblDesc;
  sLevel, sBlockSize, sFillFactor: string;
begin
  pOptDesc := AllocMem(3 * sizeof(FLDDesc));
  pOrigDesc := pOptDesc;
  pOptData := AllocMem(20);
  pOrigData := pOptData;
  try
    sLevel := IntToStr(Level);
    sBlockSize := IntToStr(BlockSize);
    sFillFactor := IntToStr(FIllFactor);
    // Set up first parameter
    pOptDesc.iOffset := 0;
    pOptDesc.iLen := Length(sLevel) + 1;
    StrPCopy(pOptDesc.szName, 'LEVEL');
    StrPCopy(PChar(pOptData), sLevel);
    Inc(pOptData, Length(sLevel) + 1);
    Inc(pOptDesc);
    // Set up second parameter
    pOptDesc.iOffset := Length(sLevel) + 1;
    pOptDesc.iLen := Length(sLevel) + 1 + Length(sBlockSize) + 1;
    StrPCopy(pOptDesc.szName, 'BLOCK SIZE');
    StrPCopy(PChar(pOptData), sBLockSize);
    Inc(pOptData, Length(sBlockSize) + 1);
    Inc(pOptDesc);
    // Set up third parameter
    pOptDesc.iOffset := Length(sLevel) + 1 + Length(sBlockSize) + 1;
    pOptDesc.iLen := Length(sLevel) + 1 + Length(sBlockSize) + 1 +
      Length(sFillFactor) + 1;
    StrPCopy(pOptDesc.szName, 'FILL FACTOR');
    StrPCopy(PChar(pOptData), sFillFactor);
    // Format the table descriptor
    FillChar(TblDesc, sizeof(TblDesc), #0);
    StrPCopy(TblDesc.szTblName, TableName);
    StrCopy(TblDesc.szTblType, szPARADOX);
    TblDesc.iOptParams := 3;
    TblDesc.pFldOptParams := pOrigDesc;
    TblDesc.pOptData := pOrigData;
    TblDesc.iFldCount := Fields;
    TblDesc.pFldDesc := pFlds;
    // Create the table
    Check(DbiCreateTable(hTmpDb, True, TblDesc));
  finally
    FreeMem(pOrigDesc, 3 * sizeof(FLDDesc));
```

```
    FreeMem(pOrigData, 20);
  end;
end;
```

DbiCreateTempTable {button C Examples,JI(`>example',`exdbicreatetemptable')} {button Delphi Examples,JI(`>example',`dexdbicreatetemptable')}

C syntax

```
DBIResult DBIFN DbiCreateTempTable (hDb, pcrTblDsc, phCursor);
```

Delphi syntax

```
function DbiCreateTempTable (hDb: hDBIDb; var crTblDsc: CRTblDesc; var
  hCursor: hDBICur): DBIResult stdcall;
```

Description

DbiCreateTempTable creates a temporary table that is deleted when the cursor is closed, unless the call is followed by a call to DbiMakePermanent or DbiSaveChanges.

Parameters

hDb Type: hDBIDb (Input)
Specifies the database handle. When a NULL hDb is specified, all temp tables will be created in the default working directory of the current session (the location of BDE executable or explicitly set by using DbiSetDirectory)--unless a private directory has been explicitly set on the current session by using DbiSetPrivateDir.

pcrTblDsc Type: pCRTblDesc (Input)
Pointer to the table descriptor structure (CRTblDesc). Usage is the same as in DbiCreateTable except that referential integrity cannot be created for a temporary table. Refer to DbiGetFieldTypeDesc and DbiGetIndexTypeDesc for more information on the legal values for these structures for each Borland Database Engine driver.

phCursor Type: phDBICur (Output)
Pointer to the cursor handle for the table.

Usage

Physical as well as logical field types are supported by the temporary table.

SQL: This function is not supported with SQL tables.

DbiResult return values

DBIERR_NONE The table was created successfully.

See also

DbiMakePermanent, DbiCreateTable, DbiCreateInMemTable

C Examples: DbiCreateTempTable

Example 1: Create a temporary table using IDAPI logical types in the field descriptor.

The temporary table can be made permanent later on. This example uses the following input:

fDbiCreateTempTable1(hDb, &hTmpCur);

```
DBIResult fDbiCreateTempTable1(hDBIDb hTmpDb, phDBICur phTmpCur)
{
   CHAR             szTblName[] = "TempPXTbl";
   CRTblDesc      TblDesc;    // Create Table Descriptor
   DBIResult        rslt;
   UINT16           NumFields  = 2;
   FLDDesc          fldDesc[] = {
            {     // Field 1 - ALPHA
               1, "MyAlpha", fldZSTRING, fldUNKNOWN, 10, 0,
               0, 0, 0, fldvNOCHECKS, fldrREADWRITE
            },
            {     // FIELD 2 - NUMERIC
               2, "MyNumber", fldFLOAT, fldUNKNOWN, 0, 0, 0,
               0, 0, fldvNOCHECKS, fldrREADWRITE
            }
   };
   memset((void *) &TblDesc, 0, sizeof(CRTblDesc));
   lstrcpy(TblDesc.szTblName, szTblName);
   lstrcpy(TblDesc.szTblType, szPARADOX);
   TblDesc.iFldCount = NumFields;
   TblDesc.pfldDesc = fldDesc;
   //Could add indexes, validity checks, and security descriptors here.
   rslt = Chk(DbiCreateTempTable(hTmpDb, &TblDesc, phTmpCur));
   return rslt;
}
```

Example 2: Create a temporary table using dBASE physical types in the field descriptor.

The temporary table can be made permanent later on. This example uses the following input:

fDbiCreateTempTable2(hDb, &hTmpCur);

```
DBIResult fDbiCreateTempTable2(hDBIDb hTmpDb, phDBICur phTmpCur)
{
   CHAR             szTblName[] = "TempdBASETbl";
   CRTblDesc        TblDesc;    // Create Table Descriptor
   DBIResult        rslt;
   UINT16           NumFields  = 2;
   FLDDesc          fldDesc[] = {
            {      // Field 1 - MEMO
               1, "MyAlpha", fldDBMEMO, fldUNKNOWN, 0, 0,
               0, 0, 0, fldvNOCHECKS, fldrREADWRITE
            },
            {      // FIELD 2 - BOOLEAN
               2, "MyNumber", fldDBBOOL, fldUNKNOWN, 0, 0, 0,
               0, 0, fldvNOCHECKS, fldrREADWRITE
            }
   };
   memset((void *) &TblDesc, 0, sizeof(CRTblDesc));
```

```
    lstrcpy(TblDesc.szTblName, szTblName);
    lstrcpy(TblDesc.szTblType, szDBASE);
    TblDesc.iFldCount = NumFields;
    TblDesc.pfldDesc = fldDesc;
    //Could add indexes, validity checks, and security descriptors here.
    rslt = Chk(DbiCreateTempTable(hTmpDb, &TblDesc, phTmpCur));
    return rslt;
}
```

Delphi Examples: DbiCreateTempTable

Example 1: Create a temporary table using BDE logical types in the field descriptor.

Note: This table can be made permanent later on.

```
procedure fDbiCreateTempTable(var hTmpDb: hDBIDb;var hTmpCur: hDBICur);
const
  fldDes: array[0..1] of FLDDesc = (
          ( // Field 1 - ALPHA
           iFldNum:      1;            { Field Number }
           szName:       'MyAlpha';    { Field Name }
           iFldType:     fldZSTRING;   { Field Type }
           iSubType:     fldUNKNOWN;   { Field Subtype }
           iUnits1:      10;           { Field Size }
           iUnits2:      0;
           iOffset:      0;
           iLen:         0;
           iNullOffset:  0;
           efldvVchk:    fldvNOCHECKS;
           efldrRights:  fldrREADWRITE
          ),
          ( // FIELD 2 - NUMERIC
           iFldNum:      2;
           szName:       'MyNumber';
           iFldType:     fldFLOAT;
           iSubType:     fldUNKNOWN;
           iUnits1:      0;
           iUnits2:      0;
           iOffset:      0;
           iLen:         0;
           iNullOffset:  0;
           efldvVchk:    fldvNOCHECKS;
           efldrRights:  fldrREADWRITE
          )
        );
var
  szTblName: array[0..DBIMAXTBLNAMELEN] of Char;
  TblDesc: CRTblDesc;    // Create Table Descriptor
  NumFields: LongInt;
begin
  StrPCopy(szTblName,'TempPXTbl');
  NumFields:= 2;
  FillChar(TblDesc,sizeof(CRTblDesc),#0);
  StrCopy(TblDesc.szTblName, szTblName);
  StrCopy(TblDesc.szTblType, szPARADOX);
  TblDesc.iFldCount:= NumFields;
  TblDesc.pfldDesc:= @fldDes;
  //Could add indexes, validity checks, and security descriptors here.
  Check(DbiCreateTempTable(hTmpDb, TblDesc, hTmpCur));
end;
```

Example 2: Create a temporary table using dBASE physical types in the field

descriptor.
Note: This table can be made permanent later on.

```
procedure fDbiCreateTempTable(hTmpDb: hDBIDb;hTmpCur: hDBICur);
const
  fldDes: array[0..1] of FLDDesc = (
    ( // Field 1 - MEMO
      iFldNum:      1;            { Field Number }
      szName:       'MyAlpha';    { Field Name }
      iFldType:     fldDBMEMO;    { Field Type }
      iSubType:     fldUNKNOWN;   { Field Subtype }
      iUnits1:      0;            { Field Size }
      iUnits2:      0;
      iOffset:      0;
      iLen:         0;
      iNullOffset:  0;
      efldvVchk:    fldvNOCHECKS;
      efldrRights:  fldrREADWRITE
      ),
    ( // FIELD 2 - BOOLEAN
      iFldNum:      2;            { Field Number }
      szName:       'MyNumber';   { Field Name }
      iFldType:     fldDBBOOL;    { Field Type }
      iSubType:     fldUNKNOWN;   { Field Subtype }
      iUnits1:      0;            { Field Size }
      iUnits2:      0;
      iOffset:      0;
      iLen:         0;
      iNullOffset:  0;
      efldvVchk:    fldvNOCHECKS;
      efldrRights:  fldrREADWRITE)
  );
var
   szTblName: array[0..DBIMAXTBLNAMELEN] of Char;
   TblDes: CRTblDesc;  // Create Table Descriptor
   NumFields: LongInt;
begin
  StrCopy(szTblName,'TempdBASETbl');
  NumFields:= 2;
  FillChar(TblDes,SizeOf(TblDes),#0);
  StrCopy(TblDes.szTblName, szTblName);
  StrCopy(TblDes.szTblType, szDBASE);
  TblDes.iFldCount := NumFields;
  TblDes.pfldDesc := @fldDes;
  //Could add indexes, validity checks, and security descriptors here.
  Check(DbiCreateTempTable(hTmpDb, TblDes, hTmpCur));
end;
```

DbiDateDecode {button C Examples,JI(`>example',`exdbidatedecode')} {button Delphi Examples,JI(`>example',`dexdbidatedecode')}

C syntax

```
DBIResult DBIFN DbiDateDecode (dateD, piMon, piDay, piYear);
```

Delphi syntax

```
function DbiDateDecode (dateD: DbiDate; var iMon: Word; var iDay: Word; var
  iYear: SmallInt): DBIResult stdcall;
```

Description

DbiDateDecode decodes DBIDATE into separate month, day, and year components.

Parameters

dateD Type: DBIDATE (Input)
Specifies the encoded date.

piMon Type: pUINT16 (Output)
Pointer to the client variable that receives the decoded month component. Valid values range from 1 through 12.

piDay Type: pUINT16 (Output)
Pointer to the client variable that receives the decoded day component. Valid values range from 1 through 31.

piYear Type: pINT16 (Output)
Pointer to the client variable that receives the decoded year component. Valid values range from -9999 to 9999.

Usage

This call enables the client to interpret date information returned from a call to DbiGetField.

DbiResult return values

DBIERR_NONE	The date was decoded successfully.
DBIERR_INVALIDHNDL	At least one of the following parameters is NULL: piMon, piDay, piYear.

See also

DbiGetField, DbiDateEncode, DbiTimeEncode, DbiTimeDecode, DbiTimeStampEncode, DbiTimeStampDecode

C Examples: DbiDateDecode

Decode a DBIDATE structure into month, day, and year numbers.
In this example the input is:

```
fDbiDateDecode(dbDate, &M, &D, &Y);

DBIResult fDbiDateDecode(DBIDATE DecodeDate, pUINT16 Month, pUINT16 Day,
  pINT16 Year)
{
   DBIResult   rslt;
   rslt = Chk(DbiDateDecode(DecodeDate, Month, Day, Year));
   return rslt;
}
```

Delphi Examples: DbiDateDecode

Decode a DBIDATE structure into month, day, and year numbers.

DbiDateDecode returns a string containing the date. Keep in mind that you need to use this function only when you are directly accessing BDE format dates. Otherwise, VCL handles this conversion for you.

This example uses the following input:

```
  fDbiDateDecode(MyDate, MyMonth, MyDay, MyYear);
```

The function is defined as:

```
function fDbiDateDecode(dateD: DbiDATE; var Month: word; var Day: word; var
  Year: SmallInt): string;
begin
  Check(DbiDateDecode(dateD, Month, Day, Year));
  Result := Format('%d/%d/%d', [Month, Day, Year]);
end;
```

DbiDateEncode {button C Examples,JI(`>example',`exdbidateencode')} {button Delphi Examples,JI(`>example',`dexdbidateencode')}

C syntax

```
DBIResult DBIFN DbiDateEncode (iMon, iDay, iYear, pdateD);
```

Delphi syntax

```
function DbiDateEncode (iMon: Word; iDay: Word; iYear: SmallInt; var dateD:
  DbiDate): DBIResult stdcall;
```

Description

DbiDateEncode encodes separate date components into DBIDATE for use by DbiPutField and other functions.

Parameters

iMon Type: UINT16 (Input)
Specifies the month. Valid values range from 1 through 12.

iDay Type: UINT16 (Input)
Specifies the day. Valid values range from 1 through 31.

iYear Type: INT16 (Input)
Specifies the year. Valid values range from -9999 to 9999.

pdateD Type: pDBIDATE (Output)
Pointer to the client buffer that receives the encoded date.

Usage

This function enables the client to construct a logical date value to use with the function DbiPutField.

DbiResult return values

DBIERR_NONE	The date was encoded successfully.
DBIERR_INVALIDHNDL	pDate is NULL.
DBIERR_INVALIDPARAM	The ranges of month and day parameters are wrong, according to the rules of the Gregorian calendar. iMon is zero or iMon is greater than 12 or iDay is zero or iDay is greater than 31.

See also

DbiDateDecode, DbiTimeEncode, DbiTimeDecode, DbiTimeStampEncode, DbiTimeStampDecode

C Examples: DbiDateEncode

Encode month, day, and year numbers into a DBIDATE structure.
In this example the input is:
fDbiDateEncode(2, 24, 67, &dbDate);

```
DBIResult fDbiDateEncode(UINT16 Month, UINT16 Day, INT16 Year, pDBIDATE
  EncodeDate)
{
   DBIResult   rslt;
   rslt = Chk(DbiDateEncode(Month, Day, Year, EncodeDate));
   return rslt;
}
```

Delphi Examples: DbiDateEncode

Encode month, day, and year numbers into a DBIDATE structure.

Keep in mind that you need to use this function only when you are directly accessing BDE format dates. Otherwise, VCL handles this conversion for you.

This example uses the following input:

```
  fDbiDateEncode(2, 8, 71, MyDate);
```

The function is defined as:

```
function fDbiDateEncode(Month : Word, Day : Word, Year : SmallInt, MyDate :
  DbiDate) : DbiDate;
begin
  Check(DbiDateEncode(Month, Day, Year, MyDate));
  Result := Mydate;
end;
```

DbiDeactivateFilter {button C Examples,JI(`>example',`exdbiactivatefilter')} {button Delphi Examples,JI(`>example',`dexdbidropfilter')}

C syntax

```
DBIResult DBIFN DbiDeactivateFilter (hCursor, [hFilter]);
```

Delphi syntax

```
function DbiDeactivateFilter (hCursor: hDBICur; hFilter: hDBIFilter):
  DBIResult stdcall;
```

Description

DbiDeactivateFilter temporarily disables the specified filter from affecting the record set by turning the filter off.

Parameters

hCursor Type: hDBICur (Input)
Specifies the valid cursor handle from an open table.

hFilter Type: hDBIFilter (Input)
Specifies the filter handle of the filter to deactivate. If NULL, then all filters for this cursor are deactivated.

Usage

Once a filter has been activated, that filter controls what is contained in the record set, and all operations on the associated cursor are affected. Once a filter is deactivated, all the records that were excluded by the filter are now accessible, subject to other active filters.

Prerequisites

The filter must have been previously added and activated. If a non-NULL filter is applied, it must be activated.

DbiResult return values

DBIERR_NONE	The filter specified by *hFilter* was successfully deactivated. If NULL was passed for the filter handle, all filters were deactivated.
DBIERR_INVALIDHNDL	The specified cursor handle is invalid or NULL.
DBIERR_NOSUCHFILTER	The specified filter handle is invalid.
DBIERR_NA	The filter was already deactivated.

See also

DbiAddFilter, DbiDeactivateFilter, DbiDropFilter

C Examples: DbiDeactivateFilter

An example for this function is under development and will be provided in an upcoming Help release.

Delphi Examples: DbiDeactivateFilter

An example for this function is under development and will be provided in an upcoming Help release.

DbiDeleteAlias {button C Examples,JI(`>example',`exdbideletealias')} {button Delphi Examples,JI(`>example',`dexdbideletealias')}

C syntax

```
DBIResult DbiDeleteAlias ( [hCfg], pszAliasName );
```

Delphi syntax

```
function DbiDeleteAlias (hCfg: hDBICfg; pszAliasName: PChar): DBIResult
  stdcall;
```

Description

DbiDeleteAlias deletes an alias from the configuration file specified by the parameter *hCfg*.

Parameters

hCfg Type: hDBICfg (Input)
Specifies the configuration file to be used. This parameter is required to be NULL, indicating that the alias is removed from the configuration file for the current session.

pszAliasName Type: pCHAR (Input)
Pointer to the alias name. This is the name of the new alias that is to be removed.

Usage

This function removes an alias that is either defined for use in the current session or stored in the configuration file. (See the DbiAddAlias parameter *bPersistent*.)

Prerequisites

DbiInit must be called prior to calling DbiDeleteAlias.

DbiResult return values

DBIERR_INVALIDPARAM	Null alias name.
DBIERR_NONE	The alias was deleted successfully.
DBIERR_OBJNOTFOUND	No alias was found matching pszAliasName.

See Also

DbiInit, DbiOpenCfgInfoList, DbiAddAlias

C Examples: DbiDeleteAlias

Delete an existing index from the configuration file:

```
DBIResult fDbiDeleteAlias(char *AliasName)
{
   DBIResult   rslt;
   rslt = Chk(DbiDeleteAlias(NULL, AliasName));
   return rslt;
}
```

DbiDeleteDriver {button C Examples,JI(`>example',`cexdbideletedriver')} {button Delphi Examples,JI(`>example',`dexdbideletedriver')}

C syntax
```
DBIResult DbiDeleteDriver ( [hCfg], pszDriverName, bSave );
```

Delphi syntax
```
function DbiDeleteDriver (hCfg: hDBICfg; pszDriverName: PChar; bSave: Bool):
  DBIResult stdcall;
```

Description
DbiDeleteDriver deletes a driver from the configuration file specified by the parameter *hCfg*.

Parameters
hCfg Type: hDBICfg (Input)
Specifies the configuration file to be used. This parameter is required to be NULL, indicating that the alias is removed from the configuration file for the current session.

pszDriverName Type: pCHAR (Input)
Pointer to the driver name. This is the name of the new driver that is to be removed.

bSave Type: BOOL (Input)
If TRUE, saves the change to the configuration file.

Usage
This function removes a driver that is either defined for use in the current session or stored in the configuration file. (See the DbiAddDriver parameter *bPersistent*.)

Prerequisites
DbiInit must be called prior to calling DbiDeleteDriver.

DbiResult return values
DBIERR_INVALIDPARAM	Null driver name.
DBIERR_NONE	The driver was deleted successfully.
DBIERR_OBJNOTFOUND	No driver was found matching pszDriverName.

See Also
DbiInit, DbiOpenCfgInfoList, DbiAddDriver

C Examples: DbiDeleteDriver

An example for this function is under development and will be provided in an upcoming Help release.

Delphi Examples: DbiDeleteDriver

An example for this function is under development and will be provided in an upcoming Help release.

Delphi Examples: DbiDeleteAlias

Delete an existing index from the configuration file of the current session:

```
procedure DoDbiDeleteAlias(AliasName: string);
begin
  Check(DbiDeleteAlias(nil, PChar(AliasName)));
end;

// Sample input:
  DoDbiDeleteAlias('SomeAlias');
```

DbiDeleteIndex{button C Examples,JI(`>example',`exdbideleteindex')} {button Delphi Examples,JI(`>example',`dexdbideleteindex')}

C syntax

```
DBIResult DBIFN DbiDeleteIndex (hDb, hCursor, pszTableName, [pszDriverType],
  pszIndexName, pszIndexTagName, iIndexId);
```

Delphi syntax

```
function DbiDeleteIndex (hDb: hDBIDb; hCursor: hDBICur; pszTableName: PChar;
  pszDriverType: PChar; pszIndexName: PChar; pszIndexTagName: PChar;
  iIndexId: Word): DBIResult stdcall;
```

Description

DbiDeleteIndex drops an index on a table.

Parameters

hDb Type: hDBIDb (Input)
Specifies the database handle.

hCursor Type: hDBICur (Input)
Specifies the cursor handle. If *hCursor* is specified, the operation is performed on the table associated with that cursor. If *hCursor* is NULL, *pszTableName* and *pszDriverType* determine the table to be used. This option is not supported with Access tables.

pszTableName Type: pCHAR (Input)
Pointer to the table name. If *hCursor* is NULL, *pszTableName* and *pszDriverType* determine the table to be used. (If both *pszTableName* and *hCursor* are specified, *pszTableName* is ignored.)

For Paradox, FoxPro, and dBASE, if *pszTableName* is a fully qualified name of a table, the *pszDriverType* parameter need not be specified. If the path is not included, the path name is taken from the current directory of the database associated with *hDb*.

For SQL databases, this parameter can be a fully qualified name that includes the owner name.

pszDriverType Type: pCHAR (Input)
Pointer to the driver type. Optional. For Paradox, FoxPro, and dBASE tables, this parameter is required if *pszTableName* has no extension. This parameter is ignored if the database associated with *hDb* is a SQL database. *pszDriverType* can be one of the following values: szDBASE, szMSACCESS, or szPARADOX.

pszIndexName Type: pCHAR (Input)
Pointer to the name of the index to be dropped. See IDXDesc for index naming rules.

pszIndexTagName Type: pCHAR (Input)
Pointer to the index tag name. Used only to identify dBASE .MDX or FoxPro .CDX indexes. (See the *pszIndexName* parameter description above.) This parameter is ignored for Paradox and SQL tables.

iIndexId Type: UINT16 (Input)
Specifies the index identifier, which is the number of the index to be used. The range for the index identifier is 1 to 511. Used for Paradox tables only and is ignored if *pszIndexName* is specified.

Usage

Used to drop an index. The client application can either specify the table by name or by opening a cursor on the table. If a cursor is specified, it must not be opened with the index to be deleted.

Prerequisites

If *hCursor* is specified, an exclusive cursor handle must be supplied. The index must exist. See the following driver-specific information for locking requirements. A currently active index cannot be dropped. If the table name is specified, the table must be able to be opened exclusively.

dBASE and FoxPro: The table must be opened exclusively on behalf of the client application.

Paradox: The table must be opened exclusively on behalf of the client application. (The client application must have permission to lock the table exclusively.)

SQL: The table must be open exclusively where table locking is supported by the driver.

Access: The table must be closed to to drop an index.

Oracle8: Not supported for object types (ADT, REF, nested table, and VARARRAY).

Completion state

If a cursor is specified, DbiDeleteIndex does not affect the order or the position of the cursor.

DbiResult return values

DBIERR_NONE	The index was successfully deleted.
DBIERR_INDEXNAMERQUIRED	An index name is required.
DBIERR_INDEXREADONLY	An illegal attempt was made to delete a read-only index.
DBIERR_ACTIVEINDEX	An illegal attempt was made to delete an active, primary index.
DBIERR_MUSTUSEBASEORDER	An illegal attempt was made to delete an active, secondary index.
DBIERR_INVALIDHNDL	Handle was invalid or NULL.
DBIERR_NEEDEXCLACCESS	Exclusive access is required to delete the index.
DBIERR_NOSUCHINDEX	The specified index does not exist.

See also

DbiAddIndex, DbiCloseIndex, DbiOpenIndex, DbiSwitchToIndex, DbiDoRestructure

C Examples: DbiDeleteIndex

Remove the specified index from the specified table. This example uses the following input:
fDbiDeleteIndex(hDb, "New Customer.db", "Place");

```
DBIResult fDbiDeleteIndex(hDBIDb hTmpDb, pCHAR szTableName, pCHAR
  szIndexName)
{
   DBIResult rslt;

   rslt = Chk(DbiDeleteIndex(hTmpDb, NULL, szTableName, NULL, szIndexName,
                  NULL, NULL));

   return rslt;
}
```

Delphi Examples: DbiDeleteIndex

Delete the specified index.

Delphi users should normally call TTable.DeleteIndex rather directly calling dbiDeleteIndex. The method TTable.DeleteIndex is defined as:

```
  procedure DeleteIndex(const Name: string);
```

The following example removes an alias called "ByCompany" from TTable Table1:

```
  Table1.DeleteIndex('ByCompany');
```

Delete the active index on a table.

Most Delphi users should use Table.DeleteIndex. This example uses the following input:

```
  fDbiDeleteIndex(CustTemp, False);
```

The procedure is defined as:

```
procedure fDbiDeleteIndex(Table: TTable; Tag: Boolean);
var
  ActiveIdx: IDXDesc;
begin
  if not Table.Exclusive then
    raise EDatabaseError.Create('Table must be opened exclusively to delete
  index');
  Check(DbiGetIndexDesc(Table.Handle, 0, ActiveIdx));
  // Cannot delete the active index, so change to default
  Table.IndexName := '';
  Table.IndexFieldNames := '';
  Check(DbiDeleteIndex(Table.DBHandle, Table.Handle, nil, nil,
  ActiveIdx.szName,
    ActiveIdx.szTagName, 0));
end;
```

DbiDeleteRecord {button C Examples,JI(`>example',`exdbideleterecord')} {button Delphi Examples,JI(`>example',`dexdbideleterecord')}

C syntax

```
DBIResult DBIFN DbiDeleteRecord (hCursor, [pRecBuf]);
```

Delphi syntax

```
function DbiDeleteRecord (hCursor: hDBICur; pRecBuf: Pointer): DBIResult
  stdcall;
```

Description

DbiDeleteRecord deletes the current record of the given cursor.

Parameters

hCursor Type: hDBICur (Input)
Specifies the cursor handle.

pRecBuf Type: pBYTE (Output)
Pointer to the client buffer that receives the deleted record. Optional.

Usage

dBASE and FoxPro: DbiDeleteRecord marks the record for deletion. The record is not physically removed from the table until the table is packed with DbiPackTable.

Paradox: After a record is deleted and committed, it cannot be recalled. The record is not deleted if the deletion would cause violation of referential integrity. For example, if the cursor is validly positioned on a record within the master table, and that record has linked values in a detail table, then the call to DbiDeleteRecord fails, and the position of the cursor remains unchanged.

Deleting a record does not reduce table size. The only way to gain disk space for records that have been deleted is to restructure the table with a call to DbiDoRestructure.

Access: After a record is deleted and committed, it cannot be recalled. The record is not deleted if the deletion would cause violation of referential integrity. For example, if the cursor is validly positioned on a record within the master table, and that record has linked values in a detail table, then the call to DbiDeleteRecord fails, and the position of the cursor remains unchanged.

SQL: Record deletions are done via optimistic locking. Unless a transaction is explicitly started using DbiBeginTran, a successful deletion is immediately committed.

Prerequisites

The cursor must be positioned on a record, not on a crack, beginning of file, or end of file. The user must have read/write access to the table. The record must not be locked by another session.

Completion state

After DbiDeleteRecord has successfully completed, the cursor is positioned on the crack between the records before and after the deleted record. A subsequent call to DbiGetNextRecord returns the record after the deleted record, while a subsequent call to DbiGetPriorRecord returns the record before the deleted record.

DbiResult return values

DBIERR_NONE The record was successfully deleted.

DBIERR_INVALIDHNDL The specified cursor handle is invalid or NULL.

DBIERR_BOF The cursor is not positioned on a record.

DBIERR_EOF	The cursor is not positioned on a record.
DBIERR_KEYORRECDELETED	The cursor is not positioned on a record.
DBIERR_NOCURRREC	The cursor is not positioned on a record.
DBIERR_RECLOCKED	The record or table is locked by another session.
DBIERR_NOTABLESUPPORT	A deletion cannot be made from a view. Some SQL drivers do not support deletions from non-uniquely indexed tables.
DBIERR_TABLEREADONLY	Table access denied; the cursor does not have write access to the table.
DBIERR_DETAILRECORDSEXIST	The table is the master table in a referential integrity link and the record to be deleted has associated detail records (Paradox only).
DBIERR_NOTSUFFTABLERIGHTS	Insufficient table rights to delete a record (Paradox only).
DBIERR_NOTSUFFSQLRIGHTS	Insufficient SQL rights to delete a record (SQL only).
DBIERR_MULTIPLEUNIQRECS	Attempt to delete a record that has a duplicate (SQL only).

See also

DbiGetRecord, DbiDoRestructure, DbiGetNextRecord, DbiGetPriorRecord, DbiGetRelativeRecord, DbiPackTable (dBASE and FoxPro only), DbiUndeleteRecord (dBASE and FoxPro only)

C Examples: DbiDeleteRecord

Delete the current record.

If *Pack* is set to true and the cursor is open exclusively on a dBASE or FoxPro table, the table is packed. This example uses the following input:

fDbiDeleteRecord(hdBASECur, pRecBuf, TRUE);

```
DBIResult fDbiDeleteRecord(hDBICur hTmpCur, pBYTE pTmpRecBuf, BOOL Pack)
{
   DBIResult        rslt;
   CURProps       CurProps;
   hDBIDb           hTmpDb;

   rslt = Chk(DbiDeleteRecord(hTmpCur, pTmpRecBuf));
   if (rslt != DBIERR_NONE)
      return rslt;
   if (Pack == TRUE)
   {
      rslt = Chk(DbiGetCursorProps(hTmpCur, &CurProps));
      if (rslt != DBIERR_NONE)
         return rslt;
   if (strcmp(CurProps.szTableType, szDBASE) == 0)
      {
         // Get the database handle from the cursor handle
         rslt = Chk(DbiGetObjFromObj(hTmpCur, objDATABASE, &hTmpDb));
         if (rslt != DBIERR_NONE)
            return rslt;
         rslt = Chk(DbiPackTable(hTmpDb, hTmpCur, NULL, NULL, TRUE));
      }
   }
   return rslt;
}
```

Delphi Examples: DbiDeleteRecord

Delete the current record.

Delphi users should instead use the TTable.Delete method rather than directly calling dbiDeleteRecord. This method is defined as:

```
procedure TTable.Delete;
```

The following code deletes the current record from TTable component Table1:

```
Table1.Delete;
```

DbiDeleteTable {button C Examples,JI(`>example',`exdbideletetable')} {button Delphi Examples,JI(`>example',`dexdbideletetable')}

C syntax

```
DBIResult DBIFN DbiDeleteTable (hDb, pszTableName, [pszDriverType]);
```

Delphi syntax

```
function DbiDeleteTable (hDb: hDBIDb; pszTableName: PChar; pszDriverType:
  PChar): DBIResult stdcall;
```

Description

DbiDeleteTable deletes the table given in *pszTableName*.

Parameters

hDb Type: hDBIDb (Input)
Specifies the database handle.

pszTableName Type: pCHAR (Input)
Pointer to the name of the table to delete. For Paradox, FoxPro, and dBASE, if *pszTableName* is a fully qualified name of a table, the *pszDriverType* parameter need not be specified. If the path is not included, the path name is taken from the current directory of the database associated with *hDb*.

For SQL databases, this parameter can be a fully qualified name that includes the owner name. This function cannot be used to delete SQL views.

pszDriverType Type: pCHAR (Input)
Pointer to the driver type of the table being deleted. Optional. For Paradox, FoxPro, and dBASE tables, this parameter is required if *pszTableName* has no extension. This parameter is ignored if the database associated with *hDb* is a SQL database. *pszDriverType* can be one of the following values: szDBASE, szMSACCESS, or szPARADOX.

Prerequisites

The client application must have permission to lock the table exclusively.

Paradox: If the table is encrypted, the master password must have been registered (using DbiAddPassword).

Completion state

The table and all associated family members are deleted. Deletes all files with *<tablename>.**

DbiResult return values

DBIERR_NONE	The table was successfully deleted.
DBIERR_INVALIDHNDL	The specified database handle is invalid or NULL.
DBIERR_NOSUCHFILE	The table does not exist.
DBIERR_NOSUCHTABLE	The table does not exist.
DBIERR_UNKNOWNTBLTYPE	The specified driver type is invalid.
DBIERR_NOTSUFFTABLERIGHTS	The user has insufficient rights to the table (Paradox only).
DBIERR_NOTSUFFFAMILYRIGHTS	The user has insufficient rights to family members (Paradox only).
DBIERR_LOCKED	The table is locked by another user.

See also

DbiCreateTable, DbiCopyTable, DbiAddPassword

C Examples: DbiDeleteTable

Delete a table.

Must have sufficient rights. This example uses the following input:

fDbiDeleteTable(hDb, "dBASE_TBL.DBF");

```
DBIResult fDbiDeleteTable(hDBIDb hDb, pCHAR TblName)
{
   DBIResult   rslt;
   rslt = Chk(DbiDeleteTable(hDb, TblName, NULL));
   return rslt;
}
```

Delete an opened table.

Once executed, the cursor is closed and the table deleted. This example uses the following input:

fDbiDeleteTable(hDb, &hCur);

```
DBIResult fDbiDeleteTable(hDBIDb hDb, phDBICur phCur)
{
   DBIResult   rslt;
   CURProps    Props;

   rslt = Chk(DbiGetCursorProps(*phCur, &Props));
   if (rslt != DBIERR_NONE)
      return rslt;

   rslt = Chk(DbiCloseCursor(phCur));
   if (rslt != DBIERR_NONE)
      return rslt;

   rslt = Chk(DbiDeleteTable(hDb, Props.szName, Props.szTableType));

   return rslt;
}
```

Delphi Examples: DbiDeleteTable

Delete a table.

Delphi users should use the TTable.DeleteTable method rather than directly calling DbiDeleteTable. This method is defined as:

```
procedure TTable.DeleteTable;
```

The following code deletes the table associated with the TTable object named Table1:

```
Table1.DeleteTable;
```

DbiDllExit

C syntax
```
DBIResult DBIFN DbiDllExit (VOID);
```

Delphi syntax
```
function DbiDllExit: DBIResult stdcall;
```

Description
DbiDllExit prepares the BDE to be disconnected within a DLL.

Usage
DbiDllExit should be called immedietely prior to DbiExit within the DLL. This function is only needed when the BDE is initialized (DbiInit) and un-initialized (DbiExit) within a DLL. It is not necessary to call DbiDllExit from within an executable: use only DbiExit.

DbiResult return values
DBIERR_NONE The connection to BDE has been successfully prepared for removal.

See also
DbiInit, DbiExit

DbiDoRestructure {button C Examples,JI(`>example',`exdbidorestructure')} {button Delphi Examples,JI(`>example',`dexdbidorestructure')}

C syntax

```
DBIResult DBIFN DbiDoRestructure (hDb, iTblDescCount, pTblDesc, pszSaveAs,
  [pszKeyviolName], [pszProblemsName], bAnalyzeOnly);
```

Delphi syntax

```
function DbiDoRestructure (hDb: hDBIDb; iTblDescCount: Word; pTblDesc:
  pCRTblDesc; pszSaveAs: PChar; pszKeyviolName: PChar; pszProblemsName:
  PChar; bAnalyzeOnly: Bool): DBIResult stdcall;
```

Description

DbiDoRestructure changes the properties of a table such as the following: modifying field types or field sizes, adding a field, deleting a field, rearranging fields; or changing indexes, security passwords, or referential integrity.

Parameters

hDb Type: hDBIDb (Input)
Specifies the database handle.

iTblDescCount Type: UINT16 (Input)
Specifies the number of table descriptors. Currently, only one table descriptor can be processed per call, so *iTblDescCount* must be set to 1.

pTblDesc Type: pCRTblDesc (Input)
Pointer to the client-allocated CRTblDesc structure, which identifies the source table, describes the new record structure (if modified), and lists all other changes to the table

pszSaveAs Type: pCHAR (Input)
Optional. If not NULL, creates a restructured table with this name and leaves the original unchanged.

pszKeyviolName Type: pCHAR (Input)
Optional. Pointer to the Key Violation table name. All records that cause an integrity violation are placed here. If NULL, no Key Violation table is created. If the user supplies a table name, that name is used. If a pointer to an empty string is specified, the table name created is returned in the user's area (must be at least DBIMAXPATHLEN+1 bytes).

pszProblemsName Type: pCHAR (Input)
Optional. Pointer to the Problems table name. If NULL, no Problems table is created. If the user supplies a table name, that name is used. If the user has overridden the default behavior with a callback, records are placed in a Problems table if they cannot be placed into the destination table without trimming data. If a pointer to an empty string is specified, the table name created is returned in the user's area (must be at least DBIMAXPATHLEN+1 bytes).

bAnalyzeOnly Type: BOOL (Input)
Not currently used.

Usage

Paradox: For Paradox, after a restructure an application can use the invariant field identification numbers to determine how each column of data has been affected by the restructure.

For example, a form on CUST table displays two fields: CUSTOMER and ADDRESS. A user then restructures the CUST table and adds a new field before CUSTOMER called CUSTOMERID and changes the name of the field CUSTOMER to CUSTOMERNAME. Even though the name and position of the original CUSTOMER field has changed, its invariant

field ID does not. When the form is reopened on the table, it can check the cursor property called *iRestrVersion*, if this has changed since the last time the form was used, it can fetch the field descriptors and use the *iFldNum* of each field descriptor to fetch the invariant field ID and compare these to the last invariant field IDs fetched before the restructure. This tells the application where each column of data has been moved regardless of any field renaming. Any new fields are given a new invariant field ID and no deleted field's ID is reused. Care must be taken not to use *iFldNum* as a field number in this case.

SQL, Access: Not currently supported for SQL or Access.

Prerequisites

The application must specify a completed CRTblDesc structure that defines the modifications to the table.

Completion state

When the restructure completes successfully, the following tables might be created:

- A Key Violations table (if *pszKeyviolName* was specified integrity violations occurred)
- A Problems table (if *pszProblemsName* was specified and there was data loss that the client disallowed by a callback)

DbiResult return values

DBIERR_NONE A table was successfully generated with the new structure.

Generally, errors returned are due to invalid descriptors or invalid transformations.

See also

DbiRegisterCallBack, DbiBatchMove for use of *pszKeyviolName* and *pszProblemsName*

C Examples: DbiDoRestructure

Example 1: Change first field type to fldINT16 and save table as NEW_CUST PX table to dBASE table.

This example uses the following input:
fDbiDoRestructure1(hDb, hCur, "CUSTOMER.DB");

```
DBIResult fDbiDoRestructure1(hDBIDb hDb, hDBICur hXCur, char *TblName)
{
   DBIResult     rslt;
   CRTblDesc   TblDesc;
   CURProps    CurProps;
   pFLDDesc    fldDesc;

   rslt = Chk(DbiGetCursorProps(hXCur, &CurProps));

   fldDesc = (pFLDDesc)malloc(CurProps.iFields * sizeof(FLDDesc));
   rslt = Chk(DbiGetFieldDescs(hXCur, fldDesc));

   fldDesc[0].iFldType = fldINT16;

   memset((void *) &TblDesc, 0, sizeof(CRTblDesc));
   lstrcpy(TblDesc.szTblName, TblName);
   lstrcpy(TblDesc.szTblType, szDBASE);
   TblDesc.pfldDesc = fldDesc;
   rslt = Chk(DbiDoRestructure(hDb, 1, &TblDesc, "NEW_CUST", NULL,
                               NULL, FALSE));
   return rslt;
}
```

Example 2: Pack a Paradox table.

This example uses the following input:
fDbiDoRestructure2(hDb, "CUSTOMER");

```
DBIResult fDbiDoRestructure2(hDBIDb hDb, char *TblName)
{
   DBIResult   rslt;
   CRTblDesc   TblDesc;

   memset((void *) &TblDesc, 0, sizeof(CRTblDesc));
   lstrcpy(TblDesc.szTblName, TblName);
   lstrcpy(TblDesc.szTblType, szPARADOX);
   TblDesc.bPack = TRUE;
   rslt = Chk(DbiDoRestructure(hDb, 1, &TblDesc, NULL, NULL,
                               NULL, FALSE));
   if(rslt == DBIERR_NONE)
      ShowMessage("Successful Pack");

   return rslt;
}
```

Example 3: Add validity checks and referential integrity to a table.

This example uses the following input:
fDbiDoRestructure3(hDb, "CUSTOMER");

```
DBIResult fDbiDoRestructure3(hDBIDb hDb, char *TblName)
{
   DBIResult   rslt;
   CRTblDesc   TblDesc;
```

```
   RINTDesc    pRintDesc[] = {{1, "Order No", rintDEPENDENT, "orders.db",
                              rintCASCADE, rintRESTRICT, 1, {2}, {1}}};
   VCHKDesc    pVchkDesc[] = {{1, TRUE, TRUE, FALSE, FALSE, 1000.00,
                              NULL, NULL, NULL, lkupNONE, NULL},
               // Setting the first field required with minimum value 1000.00
                             {2, TRUE, TRUE, FALSE, FALSE, NULL, NULL,
                              NULL, NULL, lkupNONE, NULL}};
               // Setting second field required.
   memset((void *) &TblDesc, 0, sizeof(CRTblDesc));
   lstrcpy(TblDesc.szTblName, TblName);
   lstrcpy(TblDesc.szTblType, szPARADOX);
   TblDesc.pvchkDesc = pVchkDesc;
   TblDesc.printDesc = pRintDesc;
   rslt = Chk(DbiDoRestructure(hDb, 1, &TblDesc, NULL, NULL,
                              NULL, FALSE));
   return rslt;
}
```

Example 4: Add a default value to an existing field in a Paradox table.

This example uses the following input:

fDbiDoRestructure4(hDb, "STOCK", 2, &DefValue);

```
DBIResult fDbiDoRestructure4(hDBIDb hTmpDb, pCHAR TblName, UINT16 Field,
  pVOID Value)
{
   DBIResult      rslt;
   CRTblDesc      TblDesc;
   VCHKDesc       VCHK;
   CROpType       Operation = crADD;

   memset(&VCHK, 0, sizeof(VCHK));
   VCHK.iFldNum = Field;
   VCHK.bHasDefVal = TRUE;
   memcpy(&VCHK.aDefVal, Value, sizeof(Value));

   memset(&TblDesc, 0, sizeof(TblDesc));
   strcpy(TblDesc.szTblName, TblName);
   strcpy(TblDesc.szTblType, szPARADOX);
   TblDesc.iValChkCount = 1;
   TblDesc.pecrValChkOp = &Operation;
   TblDesc.pvchkDesc = &VCHK;

   rslt = Chk(DbiDoRestructure(hTmpDb, 1, &TblDesc, NULL, NULL, NULL,
  FALSE));
   return rslt;
}
```

Example 5: Add a new field to a Paradox or dBASE table

This example uses the following input:

fDbiDoRestructure5(hDb, "STOCK.DB", NewFld, "Stock Add.DB");

Note: A field descriptor must be setup prior to calling this function. You must fill in szName, iFLdType, iSubType (optional), iUnits1 (optional), iUnits2 (optional).

```
DBIResult fDbiDoRestructure5(hDBIDb hTmpDb, pCHAR TblName, FLDDesc FldDesc,
  pCHAR NewTblName)
{
   DBIResult      rslt;
   CRTblDesc      TblDesc;
```

```
  pCROpType      AddOp;
  UINT16         i;
  pFLDDesc       pFldDesc;
  hDBICur        hCur;
  CURProps       Props;

  // Get an existing cursor on the source table.
  rslt = Chk(DbiGetCursorForTable(hTmpDb, TblName, NULL, &hCur));
  if (rslt != DBIERR_NONE)
     return rslt;

  // Get the amount of fields in the source table.
  rslt = Chk(DbiGetCursorProps(hCur, &Props));
  if (rslt != DBIERR_NONE)
     return rslt;

  // Get the existing field descriptor.
  pFldDesc = (pFLDDesc)malloc((Props.iFields + 1) * sizeof(FLDDesc));
  rslt = Chk(DbiGetFieldDescs(hCur, pFldDesc));
  if (rslt != DBIERR_NONE)
  {
     free(pFldDesc); return rslt;
  }

  // Close the source table so the restructure can occur.
  rslt = Chk(DbiCloseCursor(&hCur));
  if (rslt != DBIERR_NONE)
  {
     free(pFldDesc); return rslt;
  }

  // Move the new field descriptor to the end of the source field
 descriptor.
  memcpy(&pFldDesc[Props.iFields], &FldDesc, sizeof(FLDDesc));

  // Put a crADD at the same position ad the new field descriptor
  AddOp = (pCROpType)malloc((Props.iFields + 1) * sizeof(CROpType));
  memset(AddOp, crNOOP, (Props.iFields + 1) * sizeof(CROpType));
  AddOp[Props.iFields] = crADD;

  memset(&TblDesc, 0, sizeof(TblDesc));
  strcpy(TblDesc.szTblName, TblName);
  TblDesc.iFldCount = (UINT16)(Props.iFields + 1);
  TblDesc.pecrFldOp = AddOp;
  TblDesc.pfldDesc = pFldDesc;

  // Resync the field numbers in order.
  for (i = 0; i < Props.iFields; i++)
   pFldDesc[i].iFldNum = (UINT16)(i + 1);

  rslt = Chk(DbiDoRestructure(hTmpDb, 1, &TblDesc, NewTblName, NULL, NULL,
 FALSE));
  free(AddOp);
  free(pFldDesc);
  return rslt;
}
```

Example 6: Remove a validity descriptor on the specified field.

This example uses the following input:

fDbiDoRestructure6(hDb, "STOCK", 3);

```
DBIResult fDbiDoRestructure6(hDBIDb hTmpDb, pCHAR TblName, UINT16 Field)
{
   DBIResult      rslt;
   CRTblDesc      TblDesc;
   CROpType       Operation = crDROP;
   VCHKDesc       VCHK;

   memset(&VCHK, 0, sizeof(VCHK));
   VCHK.iFldNum = Field;

   memset(&TblDesc, 0, sizeof(TblDesc));
   strcpy(TblDesc.szTblName, TblName);
   strcpy(TblDesc.szTblType, szPARADOX);
   TblDesc.iValChkCount = 1;
   TblDesc.pecrValChkOp = &Operation;
   TblDesc.pvchkDesc = &VCHK;

   rslt = Chk(DbiDoRestructure(hTmpDb, 1, &TblDesc, NULL, NULL, NULL,
  FALSE));
   return rslt;
}
```

Delphi Examples: DbiDoRestructure

Example 1: Add a validity check to the specified field.
The field must be a longint, and the TTable must be open.

This example uses the following input:

```
  fDbiDoRestructure(Table4, Table4.Fields[0], @Min, @Max, nil, True);
```

(This input works for the EMPLOYEE.DB table.)

```
procedure fDbiDoRestructure(Tbl: TTable; Field: TField; MinVal, MaxVal,
  DefVal: pLongint; Required: Boolean);
var
  hDb: hDbiDb;
  TblDesc: CRTblDesc;
  VChk: pVChkDesc;
  Dir: string;
  NumVChks: Word;
  OpType: CROpType;
begin
  NumVChks := 0;
  SetLength(Dir, dbiMaxNameLen + 1);
  Check(DbiGetDirectory(Tbl.DBHandle, False, PChar(Dir)));
  SetLength(Dir, StrLen(PChar(Dir)));
  VChk := AllocMem(sizeof(VChkDesc));
  try
    FillChar(TblDesc, sizeof(CRTblDesc), #0);
    VChk.iFldNum := Field.Index + 1;
    Tbl.DisableControls;
    Tbl.Close;
    Check(DbiOpenDatabase(nil, nil, dbiReadWrite, dbiOpenExcl, nil, 0, nil,
  nil,
      hDb));
    Check(DbiSetDirectory(hDb, PChar(Dir)));
    with VChk^ do begin
      bRequired := Required;
      if (MinVal <> nil) then begin
        Inc(NumVChks);
        bHasMinVal := True;
        move(MinVal^, aMinVal, sizeof(MinVal^));
      end
      else
        bHasMinVal := False;
      if (MaxVal <> nil) then begin
        Inc(NumVChks);
        bHasMaxVal := True;
        move(MaxVal^, aMaxVal, sizeof(MaxVal^));
      end
      else
        bHasMaxVal := False;
      if (DefVal <> nil) then begin
        Inc(NumVChks);
        bHasDefVal := True;
        move(DefVal^, aDefVal, sizeof(DefVal^));
      end
      else
```

```
        bHasDefVal := False;
    end;
    TblDesc.iValChkCount := NumVChks;
    TblDesc.pVChkDesc := VChk;
    OpType := crADD;
    TblDesc.pecrValChkOp := @OpType;
    StrPCopy(TblDesc.szTblName, Tbl.TableName);
    StrCopy(TblDesc.szTblType, szParadox);
    Check(DbiDoRestructure(hDb, 1, @TblDesc, nil, nil, nil, False));
  finally
    Check(DbiCloseDatabase(hDb));
    FreeMem(VChk, sizeof(VChkDesc));
    Tbl.EnableControls;
    Tbl.Open;
  end;
end;
```

Example 2: Pack a Paradox (with DbiDoRestructure) or dBASE table.

This example will pack a Paradox or dBASE table therfore removing already deleted rows in a table. This function will also regenerate all out-of-date indexes (maintained indexes). This example uses the following input:

```
  PackTable(Table1)
```

The function is defined as follows:

```
// Pack a Paradox or dBASE table
// The table must be opened execlusively before calling this function...
procedure PackTable(Table: TTable);
var
  Props: CURProps;
  hDb: hDBIDb;
  TableDesc: CRTblDesc;
begin
  // Make sure the table is open exclusively so we can get the db handle...
  if not Table.Active then
    raise EDatabaseError.Create('Table must be opened to pack');
  if not Table.Exclusive then
    raise EDatabaseError.Create('Table must be opened exclusively to pack');

  // Get the table properties to determine table type...
  Check(DbiGetCursorProps(Table.Handle, Props));

  // If the table is a Paradox table, you must call DbiDoRestructure...
  if (Props.szTableType = szPARADOX) then begin
    // Blank out the structure...
    FillChar(TableDesc, sizeof(TableDesc), 0);
    // Get the database handle from the table's cursor handle...
    Check(DbiGetObjFromObj(hDBIObj(Table.Handle), objDATABASE,
  hDBIObj(hDb)));
    // Put the table name in the table descriptor...
    StrPCopy(TableDesc.szTblName, Table.TableName);
    // Put the table type in the table descriptor...
```

```
    StrPCopy(TableDesc.szTblType, Props.szTableType);
    // Set the Pack option in the table descriptor to TRUE...
    TableDesc.bPack := True;
    // Close the table so the restructure can complete...
    Table.Close;
    // Call DbiDoRestructure...
    Check(DbiDoRestructure(hDb, 1, @TableDesc, nil, nil, nil, False));
  end
  else
    // If the table is a dBASE table, simply call DbiPackTable...
    if (Props.szTableType = szDBASE) then
      Check(DbiPackTable(Table.DBHandle, Table.Handle, nil, szDBASE, True))
    else
      // Pack only works on PAradox or dBASE; nothing else...
      raise EDatabaseError.Create('Table must be either of Paradox or dBASE
' +
        'type to pack');

  Table.Open;
end;
```

Example 3: Alter a field in a Paradox or dBASE table.

This example will alter an existing field in a Paradox or dBASE table. NOTE: You must fill in all options in the ChangeRec with 0 or '' if the option is not used in the restructure. FillChar can be used to do this:

```
  Fillchar(MyChangeRec, sizeof(MyChangeRec), 0);
```

This example uses the following input:

```
  ChangeField(Table1, Table1.FieldByName('FOO'), MyChangeRec)
```

ChangeRec is defined as follows:

```
type
  ChangeRec = packed record
    szName: DBINAME;
    iType: Word;
    iSubType: Word;
    iLength: Word;
    iPrecision: Byte;
    end;
```

The function is defined as follows:

```
procedure ChangeField(Table: TTable; Field: TField; Rec: ChangeRec);
var
  Props: CURProps;
  hDb: hDBIDb;
  TableDesc: CRTblDesc;
  pFields: pFLDDesc;
  pOp: pCROpType;
  B: Byte;
begin
  // Initialize the pointers...
  pFields := nil;
  pOp := nil;
```

```
// Make sure the table is open exclusively so we can get the db handle...
if not Table.Active then
  raise EDatabaseError.Create('Table must be opened to restructure');
if not Table.Exclusive then
  raise EDatabaseError.Create('Table must be opened exclusively' +
    'to restructure');
Check(DbiSetProp(hDBIObj(Table.Handle), curxltMODE, Integer(xltNONE)));
// Get the table properties to determine table type...
Check(DbiGetCursorProps(Table.Handle, Props));
// Make sure the table is either Paradox or dBASE...
if (Props.szTableType <> szPARADOX) and (Props.szTableType <> szDBASE)
then
  raise EDatabaseError.Create('Field altering can only occur on Paradox' +
    ' or dBASE tables');
// Allocate memory for the field descriptor...
pFields := AllocMem(Table.FieldCount * sizeof(FLDDesc));
// Allocate memory for the operation descriptor...
pOp := AllocMem(Table.FieldCount * sizeof(CROpType));
try
  // Set the pointer to the index in the operation descriptor to put
  // crMODIFY (This means a modification to the record is going to
happen)...
  Inc(pOp, Field.Index);
  pOp^ := crMODIFY;
  Dec(pOp, Field.Index);
  // Fill the field descriptor with the existing field information...
  Check(DbiGetFieldDescs(Table.Handle, pFields));
  // Set the pointer to the index in the field descriptor to make the
  // midifications to the field
  Inc(pFields, Field.Index);
  // If the szName portion of the ChangeRec has something in it, change
it...
  if (Length(Rec.szName) > 0) then
    pFields^.szName := Rec.szName;
  // If the iType portion of the ChangeRec has something in it, change
it...
  if (Rec.iType > 0) then
    pFields^.iFldType := Rec.iType;
  // If the iSubType portion of the ChangeRec has something in it, change
it...
  if (Rec.iSubType > 0) then
    pFields^.iSubType := Rec.iSubType;
  // If the iLength portion of the ChangeRec has something in it, change
it...
  if (Rec.iLength > 0) then
    pFields^.iUnits1 := Rec.iLength;
  // If the iPrecision portion of the ChangeRec has something
  // in it, change it...
  if (Rec.iPrecision > 0) then
    pFields^.iUnits2 := Rec.iPrecision;
  Dec(pFields, Field.Index);
  for B := 1 to Table.FieldCount do begin
    pFields^.iFldNum := B;
    Inc(pFields, 1);
  end;
  Dec(pFields, Table.FieldCount);
```

```
    // Blank out the structure...
    FillChar(TableDesc, sizeof(TableDesc), #0);
    //  Get the database handle from the table's cursor handle...
    Check(DbiGetObjFromObj(hDBIObj(Table.Handle), objDATABASE,
  hDBIObj(hDb)));
    // Put the table name in the table descriptor...
    StrPCopy(TableDesc.szTblName, Table.TableName);
    // Put the table type in the table descriptor...
    StrPCopy(TableDesc.szTblType, Props.szTableType);
    // The following three lines are necessary when doing any field
  restructure
    // operations on a table...

    // Set the field count for the table
    TableDesc.iFldCount := Table.FieldCount;
    // Link the operation descriptor to the table descriptor...
    TableDesc.pecrFldOp := pOp;
    // Link the field descriptor to the table descriptor...
    TableDesc.pFldDesc := pFields;
    // Close the table so the restructure can complete...
    Table.Close;
    // Call DbiDoRestructure...
    Check(DbiDoRestructure(hDb, 1, @TableDesc, nil, nil, nil, False));
  finally
    if (pFields <> nil) then
      FreeMem(pFields);
    if (pOp <> nil) then
      FreeMem(pOp);
  end;
end;
```

Example 4: Add a master password to a Paradox table.

This example uses the following input:

```
  AddMasterPassword(Table1, 'MyNewPassword')
```

The procedure is:

```
procedure AddMasterPassword(Table: TTable; pswd: string);
const
  RESTRUCTURE_TRUE = WordBool(1);
var
  TblDesc: CRTblDesc;
  hDb: hDBIDb;
begin
  { Make sure that the table is opened and is exclusive }
  if not Table.Active or not Table.Exclusive then
    raise EDatabaseError.Create('Table must be opened in exclusive ' +
      'mode to add passwords');
  { Initialize the table descriptor }
  FillChar(TblDesc, SizeOf(CRTblDesc), #0);
  with TblDesc do begin
    { Place the table name in descriptor }
    StrPCopy(szTblName, Table.TableName);
    { Place the table type in descriptor }
    StrCopy(szTblType, szPARADOX);
    { Master Password, Password }
```

```
    StrPCopy(szPassword, pswd);
    { Set bProtected to True }
    bProtected := RESTRUCTURE_TRUE;
  end;
  { Get the database handle from the cursor handle }
  Check(DbiGetObjFromObj(hDBIObj(Table.Handle), objDATABASE, hDBIObj(hDb)));
  { Close the table }
  Table.Close;
  { Add the master password to the Paradox table }
  Check(DbiDoRestructure(hDb, 1, @TblDesc, nil, nil, nil, False));
  { Add the new password to the session }
  Session.AddPassword(pswd);
  { Re-Open the table }
  Table.Open;
end;
```

DbiDropFilter {button C Examples,JI(`>example',`exdbiactivatefilter')} {button Delphi Examples,JI(`>example',`dexdbidropfilter')}

C syntax
```
DBIResult DBIFN DbiDropFilter (hCursor, [hFilter]);
```

Delphi syntax
```
function DbiDropFilter (hCursor: hDBICur; hFilter: hDBIFilter): DBIResult
  stdcall;
```

Description
DbiDropFilter drops the specified filter and frees all resources associated with the filter.

Parameters
hCursor Type: hDBICur (Input)
Specifies the cursor handle.

hFilter Type: hDBIFilter (Input)
Specifies the filter handle.

Usage
The filter is automatically deactivated before being dropped, and automatically dropped when the cursor is closed. Providing a NULL filter handle drops all filters for this cursor. If no filters are activated and NULL has been specified for the filter handle, no error condition is returned.

Prerequisites
The filter must have been previously added.

DbiResult return values

DBIERR_NONE	The filter specified by the filter handle was successfully dropped. If NULL is passed for the filter handle, all filters, if any, were dropped.
DBIERR_INVALIDHNDL	The specified cursor handle is invalid or NULL.
DBIERR_NOSUCHFILTER	The filter handle (*hFilter*) is invalid.

See also
DbiActivateFilter, DbiDeactivateFilter, DbiAddFilter

C Examples: DbiDropFilter

An example for this function is under development and will be provided in an upcoming Help release.

Delphi Examples: DbiDropFilter

Deactivate and drop a filter on the specified table. This example uses the following input:

```
fDbiDropFilter(Table1, hFilter);
```

The procedure is defined as:

```
procedure fDbiDropFilter(Table: TTable; var hFilter: hDBIFilter);
var
  Props: CURProps;
begin
  Check(DbiGetCursorProps(Table.Handle, Props));
  // Check to see if there are any active filters on the cursor
  if (Props.iFilters = 0) then
    raise EDatabaseError.Create('There are no active filters on the
  specified cursor');
  if( hFilter <> nil) then begin
    // Deactivate and drop filter
    Check(DbiDeactivateFilter(Table.Handle, hFilter));
    Check(DbiDropFilter(Table.Handle, hFilter));
  end
  else
    raise EDatabaseError.Create('Filter handle is invalid or already
  dropped');
end;
```

DbiDropPassword {button C Examples,JI(`>example',`exdbidroppassword')} {button Delphi Examples,JI(`>example',`dexdbidroppassword')}

C syntax
```
DBIResult DBIFN DbiDropPassword (pszPassword);
```

Delphi syntax
```
function DbiDropPassword (pszPassword: PChar): DBIResult stdcall;
```

Description
DbiDropPassword removes a password from the current session. This function is used by the Paradox driver only.

Parameters
pszPassword Type: pCHAR (Input)
Pointer to the password to be dropped. If NULL is specified, all passwords for the session are dropped.

Usage
This function removes the rights to access previously encrypted tables with that password; it does not cause tables to become decrypted.

DbiResult return values

DBIERR_NONE	The password specified by pszPassword was successfully dropped.
DBIERR_INVALIDPASSWORD	The specified password is empty or too long.
DBIERR_OBJNOTFOUND	pszPassword was not found.

See also
DbiAddPassword

C Examples: DbiDropPassword

Remove all passwords from the current session. This example uses the following input:
fDbiDropPassword();

```
DBIResult fDbiDropPassword(VOID)
{
    return Chk(DbiDropPassword(NULL));
}
```

Delphi Examples: DbiDropPassword

Drop a password.

Delphi users should use TSession.RemovePassword method rather than directly calling dbiDropPassword. The method TSession.RemovePassword is defined as:

```
  procedure RemovePassword(const Password: string);
```

The following code removes a password called "Hip Hop" from TSession Session:

```
Session.RemovePassword('Hip Hop');
```

DbiEmptyTable {button C Examples,JI(`>example',`exdbiemptytable')} {button Delphi Examples,JI(`>example',`dexdbiemptytable')}

C syntax

```
DBIResult DBIFN DbiEmptyTable (hDb, hCursor, pszTableName, [pszDriverType]);
```

Delphi syntax

```
function DbiEmptyTable (hDb: hDBIDb; hCursor: hDBICur; pszTableName: PChar;
  pszDriverType: PChar): DBIResult stdcall;
```

Description

DbiEmptyTable deletes all records from the given table.

Parameters

hDb Type: hDBIDb (Input)
Specifies the database handle.

hCursor Type: hDBICur (Input)
Specifies the cursor on the table. Optional. If *hCursor* is specified, the operation is performed on the table associated with the cursor. If *hCursor* is NULL, *pszTableName* and *pszDriverType* determine the table to be used.

pszTableName Type: pCHAR (Input)
Pointer to the table name. Optional. If *hCursor* is NULL, *pszTableName* and *pszDriverType* determine the table to be used. (If both *pszTableName* and *hCursor* are specified, *pszTableName* is ignored.)

For Paradox, FoxPro, and dBASE, if *pszTableName* is a fully qualified name of a table, the *pszDriverType* parameter need not be specified. If the path is not included, the path name is taken from the current directory of the database associated with *hDb*.

For SQL databases, this parameter can be a fully qualified name that includes the owner name.

pszDriverType Type: pCHAR (Input)
Pointer to the driver type. Optional. For Paradox , FoxPro, and dBASE tables, this parameter is required if *pszTableName* has no extension. This parameter is ignored if the database associated with *hDb* is a SQL database. *pszDriverType* can be one of the following values: szDBASE, szMSACCESS, or szPARADOX.

Usage

This function is used to remove all records from the specified table.

Paradox: The operation is not performed if there are any conflicting referential integrity constraints on the table.

Prerequisites

If a cursor is passed in, it must have been opened in exclusive mode. For Paradox tables, if the table is encrypted, a table-level password with prvINSDEL or prvFULL rights must have been registered.

Completion state

No records remain in the table. However, all resources (for example, indexes and validity checks) remain. The table and index should now be at their respective minimum sizes.

DbiResult return values

DBIERR_NONE The table was successfully emptied.

DBIERR_INVALIDHNDL The specified database handle or the specified cursor handle is

	invalid or NULL.
DBIERR_NEEDEXCLACCESS	The table was not emptied because the user does not have exclusive access to this table.
DBIERR_NOSUCHTABLE	The table specified in pszTableName and pszDriverType does not exist.
DBIERR_INVALIDPARAM	The pointer to the table name is NULL, or the table name is an empty string.
DBIERR_NOTSUFFTABLERIGHTS	The user does not have permission to perform this operation (Paradox only).
DBIERR_NOTSUFFSQLRIGHTS	Insufficient SQL rights to perform this operation (SQL only).
DBIERR_DETAILRECEXISTEMPTY	There are conflicting referential integrity constraints on the table (Paradox only).

See also

DbiOpenTable, DbiAddPassword

C Examples: DbiEmptyTable

Creates a copy of a source table, then empties it. Both tables must be of the same type. This example uses the following input:

fDbiEmptyTable(hDb, "CUSTOMER.DB", "NEW CUSTOMER.DB");

```
DBIResult fDbiEmptyTable(hDBIDb hTmpDb, pCHAR szSrcTblName, pCHAR
  szDstTblName)
{
   DBIResult rslt;
   hDBICur hCursor = 0;

   // Creating a dummy table to empty.
   rslt = Chk(DbiCopyTable(hTmpDb, TRUE, szSrcTblName, dbiNOLOCK,
  szDstTblName));
   if (rslt != DBIERR_NONE)
      return rslt;
   // Open the dummy table so it can be emptied.
   rslt = Chk(DbiOpenTable(hTmpDb, szDstTblName, NULL, NULL, NULL, 0,
  dbiREADWRITE,
               dbiOPENEXCL, xltFIELD, FALSE, NULL, &hCursor));
   if (rslt != DBIERR_NONE)
      return rslt;
   rslt = Chk(DbiEmptyTable(hTmpDb, hCursor, NULL, NULL));
   if (rslt != DBIERR_NONE)
      return rslt;

   rslt = Chk(DbiCloseCursor(&hCursor));
   if (rslt != DBIERR_NONE)
      return rslt;

   return rslt;
}
```

Delphi Examples: DbiEmptyTable

Delete all records from the given table.

Delphi users should use the TTable.EmptyTable method rather than directly calling dbiEmptyTable. The method TTable.EmptyTable is defined as:

```
procedure TTable.EmptyTable;
```

The following code empties TTable Table1:

```
Table1.EmptyTable;
```

DbiEndDelayedUpdates {button C Examples,JI(`>example',`exdbienddelayedupdates')} {button Delphi Examples,JI(`>example',`dexdbienddelayedupdates')}

C syntax

```
DBIResult DBIFN DbiEndDelayedUpdates(phCursor);
```

Delphi syntax

```
function DbiEndDelayedUpdates (var hCursor: hDBICur): DBIResult stdcall;
```

Description

DbiEndDelayedUpdates takes the cursor out of cached updates mode, and returns a new cursor handle.

Parameters

phCursor Type: phDBICur (Input/Output)
Specifies the cached updates cursor handle and returns a new cursor handle.

Usage

Use DbiEndDelayedUpdates to terminate the cached updates mode.

Prerequisites

A call to DbiBeginDelayedUpdates must have been made.

Completion state

If this function is called while cached updates are pending, all changes are discarded.

DbiResult return values

DBIERR_NONE The cached updates mode was ended and a standard cursor handle was successfully created.

See also

DbiBeginDelayedUpdates, DbiApplyDelayedUpdates, Cached Updates

C Examples: DbiEndDelayedUpdates

An example for this function is under development and will be provided in an upcoming Help release.

Delphi Examples: DbiEndDelayedUpdates

An example for this function is under development and will be provided in an upcoming Help release.

DbiEndLinkMode {button C Examples,JI(`>example',`exdbiendlinkmode')} {button Delphi Examples,JI(`>example',`dexdbiendlinkmode')}

C syntax

```
DBIResult DBIFN DbiEndLinkMode (phCursor);
```

Delphi syntax

```
function DbiEndLinkMode (var hCursor: hDBICur): DBIResult stdcall;
```

Description

DbiEndLinkMode takes cursor out of Link mode, and returns a new cursor handle.

Parameters

phCursor Type: phDBICur (Input/Output)
Specifies the linked cursor handle, and returns a new cursor handle.

Prerequisites

A previous call to DbiBeginLinkMode must have been made. DbiUnlinkDetail should be called to unlink the cursor before DbiEndLinkMode is called.

Usage

DbiEndLinkMode takes a cursor out of Link mode. For example, if a detail cursor is taken out of link mode, it is no longer constrained by the master cursor.

Warning: The cursor handle passed in as input can no longer be used.

DbiResult return values

DBIERR_NONE Linked cursor mode was successfully ended.

See also

DbiBeginLinkMode, DbiLinkDetail, DbiUnlinkDetail

C Examples: DbiEndLinkMode

An example for this function is under development and will be provided in an upcoming Help release.

Delphi Examples: DbiEndLinkMode

End the master/detail link and close the associated cursors. This example uses the following input:

```
  fDbiEndLinkMode(hMas, hDet);
```

The procedure is defined as:

```
procedure fDbiEndLinkMode(var hMasCur, hDetCur: hDBICur);
begin
  Check(DbiUnlinkDetail(hDetCur));
  Check(DbiEndLinkMode(hMasCur));
  Check(DbiEndLinkMode(hDetCur));
  Check(DbiCloseCursor(hMasCur));
  Check(DbiCloseCursor(hDetCur));
end;
```

DbiEndTran {button C Examples,JI(`>example',`exdbiendtran')} {button Delphi Examples,JI(`>example',`dexdbiendtran')}

C syntax

```
DBIResult DBIFN DbiEndTran (hDb, hXact, eEnd);
```

Delphi syntax

```
function DbiEndTran (hDb: hDBIDb; hXact: hDBIXact; eEnd: eXEnd): DBIResult
  stdcall;
```

Description

DbiEndTran ends a transaction on a SQL server table or a local (Paradox, FoxPro, Access, and dBASE) table.

Parameters

hDb Type: hDBIDb (Input)
Specifies the database handle.

hXact Type: hDBIXact (Input)
Specifies the transaction handle.

eEnd Type: eXEnd (Input)
Specifies the transaction end type.

Usage

Ends a transaction that was previously requested. If a commit is done, all changes performed within the transaction against the associated database are made permanent. If an abort is done, all changes performed against the associated database are undone.

xendCOMMIT and xendABORT currently keep cursors if the driver and the database can support it. For xendCOMMIT and xendABORT, if the database cannot support keeping cursors, four possibilities exist for each server cursor opened on behalf of the BDE user:

- A cursor for an open query with pending results is buffered locally. Other than prematurely reading the data, no visible effect remains.
- A cursor opened on a table supporting direct positioning is closed. No other behavior is affected.
- A cursor opened on a table that does not support direct positioning is opened initially in a different transaction or connection context, if the database supports this. This cursor remains open because it exists in a different context from the requested transaction.
- If none of the previous possibilities apply, the cursor is closed and subsequent access to the BDE objects associated with the server cursor returns an error.

For local transactions, xendCOMMITKEEP is not supported by DbiEndTran.

InterBase, Sybase: It is recommended that after a rollback on a dead table you close all cursors that can be closed, reopening if needed.

Prerequisites

DbiBeginTran must have been called first.

DbiResult return values

DBIERR_NONE The transaction has ended successfully.

See also

DbiBeginTran

eEnd
Possible transaction end type values are:

Value	Description
xendCOMMIT	Commit the transaction.
xendCOMMITKEEP	Commit the transaction and keep cursors.
xendABORT	Roll back the transaction.

C Examples: DbiEndTran

End the specified transaction:

This example uses the following input:

fDbiEndTran(hDb, xTran, xendCOMMIT);

```
DBIResult fDbiEndTran(hDBIDb hTmpDb, hDBIXact hXact, eXEnd XEnd)
{
   DBIResult   rslt;
   rslt = Chk(DbiEndTran(hTmpDb, hXact, XEnd));
   return rslt;
}
```

Delphi Examples: DbiEndTran

End the specified transaction:

Delphi users should use the TDataBase.Commit,TDataBase.RollBack methods rather than directly calling DbiEndTran. These methods are defined as:

```
  procedure TDataBase.Commit;
  procedure TDataBase.RollBack;
```

The following code ends a transaction on a TDataBase object called DataBase1 and rolls back changes to the pre-transaction state:

```
{ cancels all modifications made to DataBase1 since last call to
  StartTransaction }
  DataBase1.RollBack
```

The following code ends a transaction on a TDataBase object called DataBase1 and commits changes to the table:

```
{ commits all modifications made to DataBase1 since last call to
  StartTransaction. }
  DataBase1.Commit;
end;
```

End the master/detail link and close the associated cursors. This example uses the following input:

```
  fDbiEndLinkMode(hMas, hDet);
```

The procedure is defined as:

```
procedure fDbiEndLinkMode(var hMasCur, hDetCur: hDBICur);
begin
  Check(DbiUnlinkDetail(hDetCur));
  Check(DbiEndLinkMode(hMasCur));
  Check(DbiEndLinkMode(hDetCur));
  Check(DbiCloseCursor(hMasCur));
  Check(DbiCloseCursor(hDetCur));
end;
```

DbiExit {button C Examples,JI(`>example',`exdbiexit')} {button Delphi Examples,JI(`>example',`dexdbiexit')}

C syntax
```
DBIResult DBIFN DbiExit (VOID);
```

Delphi syntax
```
function DbiExit: DBIResult stdcall;
```

Description
DbiExit disconnects the client application from BDE.

Usage
DbiExit uninitializes BDE for use by this client and releases all resources allocated by the client application. DbiExit should be the last DBI/BDE call made by the client application.

Completion state
All databases and cursors are closed, and any temporary tables are removed. If the exit is done while in a SQL transaction, the active transaction is usually rolled back. (Some SQL drivers commit.) Since the connection to BDE has been removed, the user must reinitialize BDE before any BDE functions can be called.

DbiResult return values
DBIERR_NONE The connection to BDE has been successfully removed.

See also
DbiInit, DbiDllExit

C Examples: DbiExit

Exit BDE

This example uses the following input:

fDbiExit();

```
DBIResult fDbiExit(VOID)
{
   DBIResult   rslt;
   rslt = Chk(DbiExit());
   return rslt;
}
```

Delphi Examples: DbiExit

Exit BDE

You should not call dbiExit in a Delphi application if you have any of the "Data Access" or "Data Controls" VCL components in your project. Those components will automatically call dbiInit and dbiExit.

If you are not using VCL database controls and have called dbiInit yourself, then the following code properly deinitializes the engine:

```
Check(DbiExit);.
```

DbiExtractKey {button C Examples,JI(`>example',`exdbiextractkey')} {button Delphi Examples,JI(`>example',`dexdbiextractkey')}

C syntax

```
DBIResult DBIFN DbiExtractKey (hCursor, [pRecBuf], pKeyBuf);
```

Delphi syntax

```
function DbiExtractKey (hCursor: hDBICur; pRecBuf: Pointer; pKeyBuf:
  Pointer): DBIResult stdcall;
```

Description

DbiExtractKey retrieves the key value for the current record of the given cursor or from the supplied record buffer.

Parameters

hCursor Type: hDBICur (Input)
Specifies the cursor handle. The cursor must be opened with an active index.

pRecBuf Type: pBYTE (Input)
Pointer to the record buffer from which to extract the key. Optional; if NULL, DbiExtractKey extracts the key from the current record.

pKeyBuf Type: pBYTE (Output)
Pointer to the client buffer receiving the key value. The length of the key value can be determined by retrieving the Index Descriptor (IDXDesc) and using *iKeyLen* or *iKeySize* in the CURProps structure.

Prerequisites

An index must be active. To retrieve the key from the current record, the cursor must be on a valid record.

Completion state

The extracted key value is returned in *pKeyBuf*. The returned key can be used as input to functions such as DbiSetToKey, DbiSetRange, and DbiCompareKey.

Note: In case a field map is active on the cursor, and does not include one or more of the index fields, those index fields become blanks in the extracted key if a record buffer was supplied.

Note: The key length is not affected by a field map.

DbiResult return values

DBIERR_NONE	The key value was retrieved successfully.
DBIERR_INVALIDHNDL	The specified cursor handle is invalid or NULL.
DBIERR_NOASSOCINDEX	The cursor does not have an index active.
DBIERR_NOCURRREC	The cursor is not positioned on a record.

See also

DbiGetCursorProps, DbiSetToKey, DbiSetRange, DbiCompareKeys, DbiGetRecordForKey

C Examples: DbiExtractKey

An example for this function is under development and will be provided in an upcoming Help release.

Delphi Examples: DbiExtractKey

Extract the key value for the current record.
This example assumes the field is of type character and uses the following input:

```
  fDbiExtractKey(Table1.Handle, KeyValue);
```

The procedure is defined as:

```
procedure fDbiExtractKey(hTmpCur: hDBICur; var KeyBuff: string);
var
  P: PChar;
  Props: CurProps;
begin
  Check(DbiGetCursorProps(hTmpCur,Props));
  GetMem(P, Props.IkeySize);
  Check(DbiExtractKey(hTmpCur, nil, P));
  KeyBuff:= StrPas(P);
  FreeMem(p, Props.IkeySize);
end;
```

DbiForceRecordReread {button C Examples,JI(`>example',`exdbiforcerecordreread')} {button Delphi Examples,JI(`>example',`dexdbiforcerecordreread')}

C syntax

```
DBIResult DBIFN DbiForceRecordReread (hCursor, pRecBuf);
```

Delphi syntax

```
function DbiForceRecordReread (hCursor: hDBICur; pRecBuff: Pointer):
  DBIResult stdcall;
```

Description

DbiForceRecordReread rereads a single record from the server on demand. It refreshes one row only, rather than clearing the cache.

Parameters

hCursor Type: hDBICur (Input)
Specifies the cursor handle.

pRecBuf Type: pBYTE (Output)
Location of record buffer.

Usage

Use DbiForceRecordReread as an alternative to using DbiForceReread, which allows users to refresh their cursor by re-executing the query on the server. DbiForceReread may be an expensive call because the complete contents of the local cache must be updated. Based on the optimistic record locking method, individual records (rows) may be reread from the server if a record lock is requested. However this is based on a number of factors, including record age (how long since the record has been retrieved from the server).

Using DbiForceRecordReread, a valid record is reread from the server, based on the index or record address. The refreshed record value will be placed in *pRecBuf*. The behavior is similar to DbiGetRecord with a lock, except an optimistic record lock is not obtained, and the record is always reread from the server. Keep in mind that the record is always reread using the current index (or record address), which must be unique.

Prerequisites

A table must be open.

Completion state

A valid record is reread from the server and the refreshed record value is placed in *pRecBuf*.

DbiResult return values

DBIERR_NONE	Buffers were refreshed successfully.
DBIERR_INVALIDHNDL	The specified cursor handle is invalid or NULL.

See also

DbiForceReread

C Examples: DbiForceRecordReread

Update the record buffer with current record information:

This example uses the following input:

fDbiForceRecordReread(hCur, pRecBuf);

```
DBIResult fDbiForceRecordReread(hDBICur hTmpCur, pBYTE pTmpRecBuf)
{
   DBIResult rslt;
   rslt = Chk(DbiForceRecordReread(hTmpCur, pTmpRecBuf));
   return rslt;
}
```

Delphi Examples: DbiForceRecordReread

Update the record buffer with current record information.

This example uses the following input:

```
  fDbiForceRecordReread(hCur, pRecBuf);
```

The procedure is defined as:

```
procedure fDbiForceRecordReread(hTmpCur : hDBICur, pTmpRecBuf : pBYTE)
begin
  Check(DbiForceRecordReread(hTmpCur, pTmpRecBuf));
end;
```

DbiForceReread {button C Examples,JI(`>example',`exdbiforcereread')} {button Delphi Examples,JI(`>example',`dexdbiforcereread')}

C syntax
```
DBIResult DBIFN DbiForceReread (hCursor);
```

Delphi syntax
```
function DbiForceReread (hCursor: hDBICur): DBIResult stdcall;
```

Description
DbiForceReread refreshes all buffers for the table associated with the cursor in case remote updates took place.

Parameters
hCursor Type: hDBICur (Input)
Specifies the cursor handle.

Usage
DbiForceReread is used to ensure that the client application is using current data. All subsequent retrieval operations will get new data.

Note: This function ensures only that the buffered data is current at the time of the call. Periodically use DbiForceReread or DbiCheckRefresh to ensure current data. Use record locking to prevent other users from updating records being modified by this cursor.

Note: This function is supported only on cursors for DbiOpenTable and "live" local (Paradox, FoxPro, Access, or dBASE) query cursors. "Dead" table cursors, and tables with no unique index are not supported.

Alternatively you can use DbiForceRecordReread to reread a single record from the server on demand, refreshing one row only, rather than clearing the cache.

In order to notify the client application that the table data was actually changed by a remote user, a callback of the type cbTABLECHANGED can be installed. This callback will be invoked whenever a change is detected.

Prerequisites
SQL: There must be a unique row identifier such as an index.

DbiResult return values
DBIERR_NONE Buffers were refreshed successfully.

DBIERR_INVALIDHNDL The specified cursor handle is invalid or NULL.

See also
DbiCheckRefresh, DbiRegisterCallback, DbiForceRecordReread

C Examples: DbiForceReread

Force the cache to be cleared for the specified cursor:

```
DBIResult fDbiForceReread(hDBICur hTmpCur)
{
   DBIResult rslt;
   rslt = Chk(DbiForceReread(hTmpCur));
   return rslt;
}
```

Delphi Examples: DbiForceReread

Refresh all buffers associated with TTable component T:

```
procedure ForceReread(T: TTable);
begin
  Check(DbiForceReread(T.Handle));
end;
```

DbiFormFullName {button C Examples,JI(`>example',`exdbiformfullname')} {button Delphi Examples,JI(`>example',`dexdbiformfullname')}

C syntax

```
DBIResult DBIFN DbiFormFullName (hDb, pszTableName, pszDriverType,
  pszFullName);
```

Delphi syntax

```
function DbiFormFullName (hDb: hDBIDb; pszTableName: PChar; pszDriverType:
  PChar; pszFullName: PChar): DBIResult stdcall;
```

Description

DbiFormFullName returns the fully qualified table name.

Parameters

hDb Type: hDBIDb (Input)
Specifies the database handle.

pszTableName Type: pCHAR (Input)
Pointer to the table name.

pszDriverType Type: pCHAR (Input)
Pointer to the driver type.

pszFullName Type: pCHAR (Output)
Pointer to the client buffer that receives the fully qualified table name. *pszFullName* should be allocated for DBIMAXTBLNAMELEN.

Usage

If the given table name contains a beginning drive letter followed by a colon, this function simply returns the same table name that was passed in without changing it. Otherwise, this function qualifies the table name using the directory associated with the supplied database handle. You can use DbiSetDirectory to change this directory. The table name need not be an existing file.

DbiResult return values

DBIERR_NONE The table name has been successfully returned.

DBIERR_INVALIDFILENAME The specified table name is invalid. This might occur if the combined length of the directory and table name are greater than DBIMAXTBLNAMELEN). Output *pszFullname* is left unchanged.

See also

DbiSetDirectory

C Examples: DbiFormFullName

An example for this function is under development and will be provided in an upcoming Help release.

Delphi Examples: DbiFormFullName

Return the fully qualified table name and path.
This example uses the following input:

```
  FullName:=fDbiFormFullName(Table1);
```

The function is defined as:

```
function fDbiFormFullName(Tbl: TTable): string;
var
  Props: CurProps;
begin
  Check(DbiGetCursorProps(Tbl.Handle,Props));
  SetLength(Result, DBIMAXPATHLEN);
  Check(DbiFormFullName(Tbl.DBHandle, PChar(Tbl.TableName),
    Props.szTableType, PChar(Result)));
end;
```

DbiFreeBlob {button C Examples,JI(`>example',`exdbigetblob')} {button Delphi Examples,JI(`>example',`dexdbigetblob')}

C syntax

```
DBIResult DBIFN DbiFreeBlob (hCursor, pRecBuf, iField);
```

Delphi syntax

```
function DbiFreeBlob (hCursor: hDBICur; pRecBuf: Pointer; iField: Word):
  DBIResult stdcall;
```

Description

DbiFreeBlob closes the BLOB handle obtained by DbiOpenBlob. The BLOB handle is located within the specified record buffer.

Parameters

hCursor Type: hDBICur (Input)
Specifies the cursor handle for the table. The table must contain a BLOB field.

pRecBuf Type: pBYTE (Input)
Specifies the pointer to the record buffer containing the BLOB handle. DbiOpenBlob sets the BLOB handle in the record buffer.

iField Type: UINT16 (Input)
Specifies the valid field number of the open BLOB field. If set to 0, the DbiFreeBlob call closes all open BLOBs associated with the record buffer.

Usage

The BLOB handle is closed, and all resources allocated to the BLOB with DbiOpenBlob are released.

This function must be called after calling DbiModifyRecord, DbiInsertRecord, or DbiAppendRecord (only if a BLOB has been opened), in order to free BLOB resources. DbiModifyRecord, DbiInsertRecord, or DbiAppendRecord do not automatically release BLOB resources after record modification. However, if DbiFreeBlob is called prior to calling DbiModifyRecord, DbiInsertRecord, or DbiAppendRecord, then any changes made to the BLOB are lost.

This function does not affect the contents of the BLOB on disk.

Prerequisites

The current record buffer must contain a BLOB field, and the BLOB must have been opened with DbiOpenBlob.

Completion state

After a BLOB handle has been freed, subsequent calls to DbiFreeBlob for the same handle result in an error.

DbiResult return values

DBIERR_NONE	The BLOB field was freed successfully.
DBIERR_INVALIDHNDL	The specified cursor handle is invalid or NULL.
DBIERR_INVALIDPARAM	The specified record buffer is NULL.
DBIERR_OUTOFRANGE	The number specified in iField is greater than the number of fields in the table.
DBIERR_BLOBNOTOPENED	The specified BLOB field has not been opened via a call to DbiOpenBlob. This error is returned if the BLOB has already been freed with a previous DbiFreeBlob call.
DBIERR_INVALIDBLOBHANDLE	The logical BLOB handle in the record buffer is invalid.

DBIERR_NOTABLOB	The specified field number does not correspond to a BLOB field.

See also

DbiOpenTable, DbiOpenBlob, DbiPutBlob, DbiTruncateBlob, DbiGetBlob, DbiGetBlobSize, DbiInsertRecord, DbiAppendRecord, DbiModifyRecord

DbiGetBlob {button C Examples,JI(`>example',`exdbigetblob')} {button Delphi Examples,JI(`>example',`dexdbigetblob')}

C syntax

```
DBIResult DBIFN DbiGetBlob (hCursor, pRecBuf, iField, iOffSet, iLen, pDest,
  piRead);
```

Delphi syntax

```
function DbiGetBlob (hCursor: hDBICur; pRecBuf: Pointer; iField: Word;
  iOffSet: Longint; iLen: Longint; pDest: Pointer; var iRead: Longint):
  DBIResult stdcall;
```

Description

DbiGetBlob retrieves data from the specified BLOB field.

Parameters

hCursor Type: hDBICur (Input)
Specifies the cursor handle.

pRecBuf Type: pBYTE (Input)
Pointer to the record buffer containing the BLOB handle. The record buffer is returned from a call to DbiGetNextRecord, DbiGetPriorRecord, DbiGetRelativeRecord, or DbiGetRecord. DbiOpenBlob sets the BLOB handle in the record buffer.

iField Type: UINT16 (Input)
Specifies the ordinal number of the BLOB field in the record.

iOffSet Type: UINT32 (Input)
Specifies the start location for retrieval within the BLOB field. If 0 is specified, retrieval starts from the beginning of the field. If the value exceeds the length of the BLOB field, an error is returned. If any value greater than 0 is specified, then only a portion of the BLOB field is retrieved.

iLen Type: UINT32 (Input)
Specifies the number of bytes to retrieve. *iLen* must be between 0 and the length of the BLOB field. *iLen* may be larger than 64K.

pDest Type: pBYTE (Output)
Pointer to the client buffer that receives the BLOB data.

piRead Type: pUINT32 (Output)
Pointer to the client variable that receives the actual number of bytes read. The actual number can be less than the number of bytes requested if the end of the BLOB is reached.

Usage

Any portion of the data within the BLOB field can be retrieved, starting from the position specified in *iOffSet*, and extending to the number of bytes specified in *iLen*. *pRecBuf* should contain a BLOB handle obtained by calling DbiOpenBlob. DbiGetBlob can access data larger than 64Kb, depending on the size you allocate for the buffer.

Prerequisites

The current record buffer must contain a BLOB field which has been opened by a call to DbiOpenBlob.

Completion state

piRead points to the number of bytes of BLOB data retrieved, and *pDest* points to the retrieved BLOB data.

DbiResult return values

DBIERR_NONE The BLOB field was successfully retrieved.

DBIERR_BLOBNOTOPENED	The specified BLOB field has not been opened via call to DbiOpenBlob.
DBIERR_INVALIDBLOBHANDLE	The logical BLOB handle supplied in the record buffer is invalid.
DBIERR_NOTABLOB	The specified field number does not correspond to a BLOB field.
DBIERR_INVALIDBLOBOFFSET	The start location specified in iOffSet is greater than the length of the BLOB field.
DBIERR_ENDOFBLOB	The end of the BLOB has been reached. Check piRead to see if any data was returned.

See also

DbiOpenBlob, DbiPutBlob, DbiFreeBlob, DbiTruncateBlob, DbiGetBlobSize

C Examples: DbiGetBlob

Display the specified field's memo:

The field specified in uFldNum *must* be a valid memo blob. This example uses the following input:

fBlobExample1(hCur, pRecBuf, 7);

```
DBIResult fBlobExample1 (hDBICur hTmpCur, pBYTE pTmpRecBuf, UINT16 uFldNum)
{
   DBIResult rslt;
   char      *BlobInfo;  // Holds Blob information
   UINT32    BlobSize; // Input / Output Blob size in Bytes

   rslt = Chk(DbiOpenBlob(hTmpCur, pTmpRecBuf, uFldNum, dbiREADONLY));
   if (rslt != DBIERR_NONE)
      return rslt;

   rslt = Chk(DbiGetBlobSize(hTmpCur, pTmpRecBuf, uFldNum, &BlobSize));
   if (rslt != DBIERR_NONE)
      return rslt;

   BlobInfo = (char *)malloc(BlobSize * sizeof(BYTE));

   rslt = Chk(DbiGetBlob(hTmpCur, pTmpRecBuf, uFldNum, 0, BlobSize,
                     (pBYTE)BlobInfo, &BlobSize));
   if (rslt == DBIERR_NONE)
      MessageBox(0, BlobInfo, "This is the Blob Information", MB_OK);

   free(BlobInfo);

   rslt = Chk(DbiFreeBlob(hTmpCur, pTmpRecBuf, uFldNum));

   return rslt;
}
```

Delphi Examples: DbiGetBlob

Display the specified field's memo.

The field specified in BlobIndex must be a valid memo blob and the BlobBuffer must be allocated. This example uses the following input:

```
  fDbiGetBlob(BIOLIFE_TABLE, BIOLIFE_TABLE.FieldByName('Notes').Index,
    BlobBuffer);
```

The procedure is defined as:

```
procedure fDbiGetBlob(InDataSet: TDataSet; BlobIndex: Word; var BlobInfo:
  string);
var
 NumRead: longint;
begin
  // Parameter iField of DbiOpenBlob requires an ordinal field number
  Inc(BlobIndex);
  InDataSet.UpdateCursorPos;
  Check(DbiOpenBlob(InDataSet.Handle, InDataSet.ActiveBuffer, BlobIndex,
  dbiReadOnly));
  Check(DbiGetBlobSize(InDataSet.Handle, InDataSet.ActiveBuffer, BlobIndex,
  NumRead));
  SetLength(BlobInfo, NumRead);
  Check(DbiGetBlob(InDataSet.Handle, InDataSet.ActiveBuffer, BlobIndex, 0,
    NumRead, PChar(BlobInfo), longint(NumRead)));
  Check(DbiFreeBlob(InDataSet.Handle, InDataSet.ActiveBuffer, BlobIndex));
end;
```

DbiGetBlobHeading {button C Examples,JI(`>example',`exdbigetblobheading')} {button Delphi Examples,JI(`>example',`dexdbigetblobheading')}

C syntax

```
DBIResult DBIFN DbiGetBlobHeading (hCursor, iField, pRecBuf, pDest);
```

Delphi syntax

```
function DbiGetBlobHeading (hCursor: hDBICur; iField: Word; pRecBuf:
  Pointer; pDest: Pointer): DBIResult stdcall;
```

Description

DbiGetBlobHeading retrieves information about a BLOB field from the BLOB heading in the record buffer.

Parameters

hCursor Type: hDBICur (Input)
Specifies the cursor handle.

iField Type: UINT16 (Input)
Specifies the ordinal number of the BLOB field within the record.

pRecBuf Type: pBYTE (Input)
Pointer to the client buffer containing the BLOB heading.

pDest Type: pBYTE (Output)
Pointer to the client buffer that receives the retrieved BLOB heading. The client buffer must be large enough to accommodate the retrieved information.

Usage

This function is valid only for table types that support BLOB headings, that is, Paradox only. When the table is created, the client can specify the number of bytes of the BLOB field information to be stored in the tuple itself. This information is also contained in the normal storage area of the BLOB; it is actually duplicated. The benefit of storing some of the BLOB field in the tuple is that the BLOB field does not have to be opened to retrieve this information. If the BLOB is small, it can be contained fully in the record making access faster.

Paradox: With formatted BLOB fields, the formatting information in the first eight bytes of the field is not stored within the tuple. It is functionally the same as if DbiGetBlob were called with an *iOffSet* of 8 and an *iLen* the length of the tuple area.

dBASE or FoxPro: This function is not supported for dBASE or FoxPro tables.

Access: This function is not supported for Access tables.

SQL: This function is not supported for SQL tables.

Prerequisites

This call does not require a prior call to DbiOpenBlob. (This call can be understood as the functional equivalent of a DbiGetField call for BLOB fields).

Completion state

If the BLOB does not have a heading, DbiGetBlobHeading returns an error.

DbiResult return values

DBIERR_NONE	The BLOB heading was retrieved successfully.
DBIERR_NOTABLOB	The specified field number does not correspond to a BLOB field.
DBIERR_NOTSUFFFIELDRIGHTS	The application does not have sufficient rights to this field.

DBIERR_NOTSUPPORTED	This function is not supported by SQL, dBASE, FoxPro, or Access.

See also

DbiPutBlob, DbiTruncateBlob, DbiFreeBlob, DbiGetBlob, DbiGetBlobSize

C Examples: DbiGetBlobHeading

An example for this function is under development and will be provided in an upcoming Help release.

Delphi Examples: DbiGetBlobHeading

Display the specified field's memo heading.

The field specified in BlobIndex must be a valid memo blob and the BlobBuffer must be allocated. Used only with Paradox memo fields. This example uses the following input:

```
fDbiGetBlobHeading(BIOLIFE_TABLE, BIOLIFE_TABLE.FieldByName('Notes').Index,
  BlobBuffer);
```

The procedure is defined as:

```
procedure fDbiGetBlobHeading(InDataSet: TDataSet; BlobIndex: Word; var P:
  PChar);
var
 NumRead: longint;
begin
  Inc(BlobIndex);  // Parameter iField of DbiOpenBlob requires an ordinal
  field number
  InDataSet.UpdateCursorPos;
  Check(DbiOpenBlob(InDataSet.Handle, InDataSet.ActiveBuffer, BlobIndex,
  dbiReadOnly));
  Check(DbiGetBlobSize(InDataSet.Handle, InDataSet.ActiveBuffer, BlobIndex,
  NumRead));
  Check(DbiGetBlobHeading(InDataSet.Handle, BlobIndex,
  InDataSet.ActiveBuffer, P));
  Check(DbiFreeBlob(InDataSet.Handle, InDataSet.ActiveBuffer, BlobIndex));
end;
```

DbiGetBlobSize{button C Examples,JI(`>example',`exdbigetblob')} {button Delphi Examples,JI(`>example',`dexdbigetblob')}

C syntax

```
DBIResult DBIFN DbiGetBlobSize (hCursor, pRecBuf, iField, piSize);
```

Delphi syntax

```
function DbiGetBlobSize (hCursor: hDBICur; pRecBuf: Pointer; iField: Word;
  var iSize: Longint): DBIResult stdcall;
```

Description

DbiGetBlobSize retrieves the size of the specified BLOB field in bytes.

Parameters

hCursor Type: hDBICur (Input)
Specifies the cursor handle.

pRecBuf Type: pBYTE (Input)
Pointer to the record buffer containing the BLOB handle. The client application must first allocate the buffer and fetch a valid record. A call to DbiOpenBlob then obtains the BLOB handle.

iField Type: UINT16 (Input)
Specifies the ordinal number of the BLOB field within the specified record buffer.

piSize Type: pUINT32 (Output)
Pointer to the client variable that receives the BLOB size in bytes.

Usage

This function is used to get the size of a BLOB.

Prerequisites

The current record buffer must contain a BLOB field which has been opened by a call to DbiOpenBlob.

Completion state

piSize points to the retrieved size of the BLOB field.

DbiResult return values

DBIERR_NONE	The BLOB size was successfully retrieved.
DBIERR_BLOBNOTOPENED	The specified BLOB field has not been opened with a call to DbiOpenBlob.
DBIERR_INVALIDBLOBHANDLE	The logical BLOB handle supplied in the record buffer is invalid.
DBIERR_NOTABLOB	The specified field number does not correspond to a BLOB field.

See also

DbiOpenBlob, DbiPutBlob, DbiGetBlob, DbiFreeBlob, DbiTruncateBlob

DbiGetBookMark {button C Examples,JI(`>example',`exdbigetbookmark')} {button Delphi Examples,JI(`>example',`dexdbigetbookmark')}

C syntax

```
DBIResult DBIFN DbiGetBookMark (hCur, pBookMark);
```

Delphi syntax

```
function DbiGetBookMark (hCur: hDBICur; pBookMark: Pointer): DBIResult
  stdcall;
```

Description

DbiGetBookMark saves the current position of a cursor in the client-supplied bookmark buffer. This position is called a bookmark.

Parameters

hCur Type: hDBICur (Input)
Specifies the cursor handle.

pBookMark Type: pBYTE (Output)
Pointer to the client-allocated bookmark buffer.

Usage

A bookmark contains internal information about the current position of the cursor. This information can be passed to DbiSetToBookMark to reposition the same or compatible cursor. If a bookmark is stable, it is guaranteed that the cursor can be repositioned there. Whether or not the bookmark is stable can be determined from the *bBookMarkStable* property returned by DbiGetCursorProps.

dBASE and FoxPro: For dBASE and FoxPro tables, the bookmark is always stable.

Paradox: For Paradox tables, the bookmark is stable only if the table has a primary key.

SQL: For SQL tables, the bookmark is stable only if the table has a unique index or unique row identifier.

Prerequisites

DbiGetCursorProps should be called to retrieve the *iBookMarkSize* property and the bookmark buffer should be allocated to accommodate the bookmark.

Note: The size of a bookmark depends on the current index and can change if DbiSwitchToIndex is called.

Completion state

The bookmark buffer pointed to by *pBookMark* contains the saved cursor position. The bookmark is valid only with a cursor that is using the same table and ordered with the same index.

DbiResult return values

DBIERR_NONE The bookmark was returned successfully.

DBIERR_INVALIDHNDL The specified cursor handle is invalid or NULL, or the pointer to the bookmark buffer is NULL.

See also

DbiSetToBookMark, DbiCompareBookMarks, DbiGetCursorProps

C Examples: DbiGetBookMark

Set a bookmark on the current position of a cursor:

Note: The table must have a primary index. This example uses the following input:
fDbiGetBookMark(hPXCur, &pBookmark);

```
DBIResult fDbiGetBookMark(hDBICur hCur, ppBYTE ppBookMark)
{
   DBIResult   rslt;
   CURProps    CurProps;
   rslt = Chk(DbiGetCursorProps(hCur, &CurProps));
   if (rslt != DBIERR_NONE)
     return rslt;
   *ppBookMark = (pBYTE)malloc(CurProps.iBookMarkSize);
   rslt = Chk(DbiGetBookMark(hCur, *ppBookMark));
   return rslt;
}
```

Delphi Examples: DbiGetBookMark

Set a bookmark on the current position of a cursor:

Delphi users should use the GetBookmark method associated with descendents of TDataSet including TTable, TQuery, and TStoredProc rather than directly calling DbiGetBookmark. This method is defined as:

```
function GetBookmark: TBookmark;
```

The following saves the current record information of the dataset to allow you to return to that record with a later call to the GotoBookmark method.

```
Table1.GetBookmark;
```

DbiGetCallBack{button C Examples,JI(`>example',`exdbigetcallback')} {button Delphi Examples,JI(`>example',`dexdbigetcallback')}

C syntax

```
DBIResult DBIFN DbiGetCallBack (hCursor, ecbType, piClientData, piCbBufLen,
  ppCbBuf, ppfCb);
```

Delphi syntax

```
function DbiGetCallBack (hCursor: hDBICur; ecbType: CBType; var iClientData:
  Longint; var iCbBufLen: Word; var pCbBuf: Pointer; ppfCb: ppfDBICallBack):
  DBIResult stdcall;
```

Description

DbiGetCallBack returns a pointer to the function previously registered by the client (using DbiRegisterCallBack) for the given callback type.

Parameters

hCursor Type: hDBICur (Input)
Specifies the cursor handle. If NULL, *hCursor* specifies that the callback is session-wide, rather than cursor-level.

ecbType Type: CBType (Input)
Specifies the type of callback.

piClientData Type: pUINT32 (Input)
Pointer to the passthrough client data (used by the client function).

piCbBufLen Type: pUINT16 (Input)
Pointer to the callback buffer length.

ppCbBuf Type: ppVOID (Input)
Pointer to the callback buffer pointer.

ppfCb Type: ppfDBICallBack (Output)
Pointer to the client variable that receives a pointer to the callback function that was previously registered for this type. The buffer receives a NULL pointer if no function was registered.

Usage

This function is typically used to find out whether the specified callback function was registered for the given cursor handle or the currently active session.

DbiResult return values

DBIERR_NONE The callback function for the given cursor handle has been successfully retrieved.

See also

DbiRegisterCallBack

C Examples: DbiGetCallBack

An example for this function is under development and will be provided in an upcoming Help release.

Delphi Examples: DbiGetCallBack

An example for this function is under development and will be provided in an upcoming Help release.

DbiGetClientInfo {button C Examples,JI(`>example',`exdbigetclientinfo')} {button Delphi Examples,JI(`>example',`dexdbigetclientinfo')}

Syntaax

```
DBIResult DBIFN DbiGetClientInfo (pclientInfo);
```

Delphi syntax

```
function DbiGetClientInfo (var clientInfo: CLIENTInfo): DBIResult stdcall;
```

Description

DbiGetClientInfo retrieves system-level information about the client application.

Parameters

pclientInfo Type: pCLIENTInfo (Output)
Pointer to the client-allocated CLIENTInfo structure.

Usage

This function can be used to determine if other sessions are present when exclusive access is required to a table. It can also be used to determine the current language driver and to get the working directory.

Completion state

The output buffer pointed to by *pclientInfo* contains client environment information.

DbiResult return values

DBIERR_NONE Client application information was returned successfully.

See also

DbiGetSysVersion, DbiGetSysConfig, DbiGetSysInfo

C Examples: DbiGetClientInfo

Obtain client info:

If *ClientStr* is not null, this function also creates a string with the client information. This example uses the following input:

fDbiGetClientInfo(&Client, Buffer);

```
DBIResult fDbiGetClientInfo(pCLIENTInfo pCInfo, pCHAR ClientStr)
{
   DBIResult   rslt;
   rslt = Chk(DbiGetClientInfo(pCInfo));
   if ((rslt == DBIERR_NONE) && (ClientStr != NULL))
      wsprintf(ClientStr, "Name: %s, Sessions: %d, Working Dir: %s,
  Language: %s",
         pCInfo->szName, pCInfo->iSessions, pCInfo->szWorkDir, pCInfo-
  >szLang);
   return rslt;
}
```

Delphi Examples: DbiGetClientInfo

Display a message box containing system-level information about client application.

This example uses the following input:

```
  ShowClientInfo;
```

The procedure is defined as:

```
procedure ShowClientInfo;
const
  InfoStr = 'Name: %s'#13#10'Number of sessions: %d'#13#10 +
    'Working directory: %s'#13#10'Language: %s';
var
  ClientInf: ClientInfo;
begin
  Check(DbiGetClientInfo(ClientInf));
  with ClientInf do
    ShowMessage(Format(InfoStr, [szName, iSessions, szWorkDir, szLang]));
end;
```

DbiGetCurrSession {button C Examples,JI(`>example',`exdbigetcurrsession')} {button Delphi Examples,JI(`>example',`dexdbigetcurrsession')}

C syntax

```
DBIResult DBIFN DbiGetCurrSession (phSes);
```

Delphi syntax

```
function DbiGetCurrSession (var hSes: hDBISes): DbiResult stdcall;
```

Description

DbiGetCurrSession returns the handle associated with the current session.

Parameters

phSes Type: phDBISes (Output)
Pointer to the current session handle.

Completion state

This function returns the handle to current session, that is, the handle identified by the most recent call to DbiSetCurrSession or DbiStartSession. If neither of these calls has been made, DbiGetCurrSession returns the handle to the default session.

DbiResult return values

DBIERR_NONE The current session handle has been retrieved successfully.

DBIERR_INVALIDHNDL *phSes* is NULL.

See also

DbiSetCurrSession, DbiStartSession, DbiCloseSession, DbiGetSysInfo, DbiGetSesInfo

C Examples: DbiGetCurrSession

Return the handle associated with the current session:

This function returns the handle of the current session. If *pSesInfo* is not null, DbiGetCurrSession retrieves session information. This example uses the following input:

fDbiGetCurrSession(&hSes, &SesInfo);

```
DBIResult fDbiGetCurrSession(phDBISes pTmpSes, pSESInfo pSesInfo)
{
   DBIResult      rslt;
   rslt = Chk(DbiGetCurrSession(pTmpSes));
   if (rslt == DBIERR_NONE)
   {
      if (pSesInfo != NULL)
         rslt= Chk(DbiGetSesInfo(pSesInfo));
   }
   return rslt;
}
```

Delphi Examples: DbiGetCurrSession

Return the handle associated with the current session:

If a call to DbiStartSession has occurred previously, then this function returns a handle to the current session. This example uses the following input:

```
  fDbiGetCurrSession(hSes);
```

The procedure is defined as:

```
procedure fDbiGetCurrSession(var hTmpSes: hDBISes);
begin
  Check(DbiGetCurrSession(hTmpSes));
end;
```

DbiGetCursorForTable {button C Examples,JI(`>example',`exdbigetcursorfortable')} {button Delphi Examples,JI(`>example',`dexdbigetcursorfortable')}

C syntax

```
DBIResult DBIFN DbiGetCursorForTable ([hDb], pszTableName, [pszDriverType],
  phCursor);
```

Delphi syntax

```
function DbiGetCursorForTable (hDb: hDBIDb; pszTableName: PChar;
  pszDriverType: PChar; var hCursor: hDBICur): DBIResult stdcall;
```

Description

DbiGetCursorForTable returns an existing cursor for the given table within the current session.

Parameters

hDb Type: hDBIDb (Input)
Specifies the database handle. Optional. If supplied, DbiFormFullName is called to create a fully qualified table name.

pszTableName Type: pCHAR (Input)
Pointer to the table name.

pszDriverType Type: pCHAR (Input)
Pointer to the driver type. Optional. If supplied, used with *hDb* in a call to DbiFormFullName.

phCursor Type: phDBICur (Output)
Pointer to a cursor handle.

Usage

If more than one cursor is opened on the table, the first cursor found on the table is returned. There is no implied ordering of cursors on a table.

DbiResult return values

DBIERR_NONE	The cursor for the table was retrieved successfully.
DBIERR_INVALIDHNDL	The specified database handle is invalid or NULL.
DBIERR_INVALIDPARAM	The specified table name or the pointer to the table name is NULL.
DBIERR_NOSUCHTABLE	The specified table name is invalid.
DBIERR_OBJNOTFOUND	A valid cursor could not be found.

See also

DbiFormFullName

C Examples: DbiGetCursorForTable

Retrieve a cursor for a table.

The table should have at least one open cursor already on it. This example uses the following input:

fDbiGetCursorForTable("STOCK.DB", &hSTOCKCur);

```
DBIResult fDbiGetCursorForTable(pCHAR TblName, hDBIDb hDb, phDBICur phCur)
{
   DBIResult   rslt;
   rslt = Chk(DbiGetCursorForTable(hDb, TblName, szPARADOX, phCur));
   return rslt;
}
```

Delphi Examples: DbiGetCursorForTable

Return an existing cursor for the given table within the current session.
This function also returns the name of the index on which the table is open. This example uses the following input:

```
  OutputStr:= fDbiGetCursorForTable(Table1.DBHandle, Table1.TableName,
  MyNewCursor);
```

The function is defined as:

```
function fDbiGetCursorForTable(hTmpDb: hDbiDb; TblName: string; var hNewCur:
  hDBICur): string;
var
  IndexDesc: IdxDesc;
begin
  Check(DbiGetCursorForTable(hTmpDb, PChar(TblName), '', hNewCur));
  Check(DbiGetIndexDesc(hNewCur, 0, IndexDesc));
  Result := StrPas(IndexDesc.szName);
end;
```

DbiGetCursorProps {button C Examples,JI(`>example',`exdbigetcursorprops')} {button Delphi Examples,JI(`>example',`dexdbigetcursorprops')}

C syntax

```
DBIResult DBIFN DbiGetCursorProps (hCursor, pcurProps);
```

Delphi syntax

```
function DbiGetCursorProps (hCursor: hDBICur; var curProps: CURProps):
  DBIResult stdcall;
```

Description

DbiGetCursorProps returns the properties of the cursor.

Parameters

hCursor Type: hDBICur (Input)
Specifies the cursor handle.

pcurProps Type: pCURProps (Output)
Pointer to the client-allocated CURProps structure.

Usage

This function retrieves the most commonly used cursor properties. Additional properties can be obtained by using DbiGetProp. This function can be called immediately after DbiOpenTable to retrieve information necessary to allocate the record buffer and the array for the field descriptors in the table.

FoxPro: To see if a cursor is referencing a FoxPro table, check if *iTblLevel* is equal to the constant FOXLEVEL25.

DbiResult return values

DBIERR_NONE Cursor properties for *hCursor* were successfully retrieved.

DBIERR_INVALIDHNDL The specified cursor handle is invalid or NULL.

See also

DbiGetProp, DbiSetProp, Getting and Setting Properties, CURProps

C Examples: DbiGetCursorProps

Return a string containing cursor properties.

OutName must have sufficient space to hold the return string. This example uses the following input:

fDbiGetCursorProps(hCursor, Name);

```
DBIResult fDbiGetCursorProps(hDBICur hTmpCur, pCHAR OutName)
{
   DBIResult      rslt;
   CURProps       Prop;
   rslt = Chk(DbiGetCursorProps(hTmpCur, &Prop));
   if (rslt != DBIERR_NONE)
      return rslt;
   wsprintf(OutName, "Name: %s, Name Size: %d\r\nTableType: %s"
       ", Fields: %d\r\nRecord Buffer Size: %d, Key Size: %d\r\nIndexes: %d"
       ", Validity Checks: %d\r\nRef Integ Checks: %d, Passwords: %d",
       Prop.szName, Prop.iFNameSize, Prop.szTableType, Prop.iFields,
       Prop.iRecBufSize, Prop.iKeySize, Prop.iIndexes, Prop.iValChecks,
       Prop.iRefIntChecks, Prop.iPasswords);
   return rslt;
}
```

Delphi Examples: DbiGetCursorProps

Example 1: Return the size of the record buffer needed to hold information for one record.

Note: Delphi programs should use TTable.RecordSize.

This example uses the following input:

```
  RecBuf := AllocMem(fDbiGetCursorProps1(Table1.Handle));
```

The function is defined as:

```
function fDbiGetCursorProps1(hTmpCur: hDbiCur): Word;
var
  Prop : CURProps;
begin
  Check(DbiGetCursorProps(hTmpCur, Prop));
  Result := Prop.iRecBufSize;
end;
```

Example 2: Return information about the table open on the specified cursor.

This example uses the following input:

```
  fDbiGetCursorProps2(Table1.Handle, TmpList);
```

The procedure is defined as:

```
procedure fDbiGetCursorProps2(hTmpCur: hDbiCur; CurList: TStringList);
var
  Prop : CURProps;
begin
  Check(DbiGetCursorProps(hTmpCur, Prop));
  with CurList do begin
    Add('Table Name: ' + Prop.szName);
    Add('Table Type: ' + Prop.szTableType);
    Add('Fields: ' + IntToStr(Prop.iFields));
    Add('Record Buffer Size: ' + IntToStr(Prop.iRecBufSize));
    Add('Indexes: ' + IntToStr(Prop.iIndexes));
    Add('Validity Checks: ' + IntToStr(Prop.iValChecks));
    Add('Referential Integ Checks: ' + IntToStr(Prop.iRefIntChecks));
    Add('Table Level: ' + IntToStr(Prop.iTblLevel));
    Add('Language Driver: ' + Prop.szLangDriver);
  end;
end;
```

DbiGetDatabaseDesc {button C Examples,JI(`>example',`exdbigetdatabasedesc')} {button Delphi Examples,JI(`>example',`dexdbigetdatabasedesc')}

C syntax
```
DBIResult DBIFN DbiGetDatabaseDesc (pszName, pdbDesc);
```

Delphi syntax
```
function DbiGetDatabaseDesc (pszName: PChar; pdbDesc: pDBDesc): DBIResult
  stdcall;
```

Description
DbiGetDatabaseDesc retrieves the description of the specified database from the configuration file.

Parameters
pszName Type: pCHAR (Input)
Pointer to the database name.

pdbDesc Type: pDBDesc (Output)
Pointer to the client-allocated DBDesc structure.

Prerequisites
A valid database (alias) name must be specified.

Completion state
The output buffer contains the database description.

DbiResult return values
DBIERR_NONE The database description for *pszName* was retrieved successfully.

DBIERR_OBJNOTFOUND The database named in *pszName* was not found.

See also
DbiOpenDatabaseList

C Examples: DbiGetDatabaseDesc

Get database description.

If *DBStr* is not null, this function also creates a string with the database information. This example uses the following input:

fDbiGetDatabaseDesc("BDEDEMOS", &DbDesc, Buffer)

```
DBIResult fDbiGetDatabaseDesc(pCHAR DBName, pDBDesc pDB, pCHAR DBStr)
{
   DBIResult   rslt;
   rslt = Chk(DbiGetDatabaseDesc(DBName, pDB));
   if ((rslt == DBIERR_NONE) && (DBStr != NULL))
      wsprintf(DBStr, "Name: %s, Description: %s, Physical Name: %s, Type:
  %s",
         pDB->szName, pDB->szText, pDB->szPhyName, pDB->szDbType);
   return rslt;
}
```

Delphi Examples: DbiGetDatabaseDesc

Retrieve the description of the specified database from the configuration file.
This example uses the following input:

```
  ShowDatabaseDesc('IBLOCAL');
```

The procedure is defined as:

```
procedure ShowDatabaseDesc(DBName: string);
const
  DescStr = 'Driver Name: %s'#13#10'AliasName: %s'#13#10 +
    'Text: %s'#13#10'Physical Name/Path: %s';
var
  dbDes: DBDesc;
begin
  Check(DbiGetDatabaseDesc(PChar(DBName), @dbDes));
  with dbDes do
    ShowMessage(Format(DescStr, [szDbType, szName, szText, szPhyName]));
end;
```

DbiGetDateFormat {button C Examples,JI(`>example',`exdbigetdateformat')} {button Delphi Examples,JI(`>example',`dexdbigetdateformat')}

C syntax

```
DBIResult DBIFN DbiGetDateFormat (pfmtDate);
```

Delphi syntax

```
function DbiGetDateFormat (var fmtDate: FMTDate): DBIResult stdcall;
```

Description

DbiGetDateFormat gets the date format for the current session.

Parameters

pfmtDate Type: pFMTDate (Output)
Pointer to the client-allocated FMTDate structure.

Usage

The date format is used by QBE for input and wildcard character matching. It is also used by batch operations (such as DbiDoRestructure and DbiBatchMove) to handle data type coercion between character and date types. The default date format can be changed by editing the system configuration file. The date format for the current session can be changed using DbiSetDateFormat.

DbiResult return values

DBIERR_NONE The date format was successfully retrieved.

DBIERR_INVALIDHNDL *pfmtDate* is NULL.

See also

DbiGetNumberFormat, DbiGetTimeFormat, DbiSetDateFormat

C Examples: DbiGetDateFormat

An example for this function is under development and will be provided in an upcoming Help release.

Delphi Examples: DbiGetDateFormat

Retrieve the date separator from the current session
The date separator is displayed it in a dialog box.

```
procedure TForm1.Button3Click(Sender: TObject);
var
  fmt: fmtdate;
  s: string;
begin
  Check(dbiGetDateFormat(fmt));
  s:=fmt.szDateSeparator;
  ShowMessage('Date is seperated by a ' + s + ' Character');
end;
```

DbiGetDirectory {button C Examples,JI(`>example',`exdbigetdirectory')} {button Delphi Examples,JI(`>example',`dexdbigetdirectory')}

C syntax
```
DBIResult DBIFN DbiGetDirectory (hDb, bDefault, pszDir);
```

Delphi syntax
```
function DbiGetDirectory (hDb: hDBIDb; bDefault: Bool; pszDir: PChar):
  DBIResult stdcall;
```

Description
DbiGetDirectory retrieves the current directory or the default directory, depending on the value specified in *bDefault*.

Parameters
hDb Type: hDBIDb (Input)
Specifies the database handle. Must be associated with a standard database.

bDefault Type: BOOL (Input)
The bDefault parameter specifies whether to retrieve the default directory or the current working directory.

pszDir Type: pCHAR (Output)
Pointer to the client-allocated buffer which receives the directory string. The buffer must be large enough to hold the directory string (DBIMAXPATHLEN + 1).

Usage
This function is valid only for a Paradox, FoxPro, or dBASE database. The default directory can be set when DbiInit is called as part of the DBIEnv structure. If DbiSetDirectory is not called, then the default directory is the same as the application startup directory.

SQL: DbiGetDirectory is not applicable to SQL databases.

Prerequisites
A valid database handle must be obtained.

Completion state
The output buffer contains the directory string.

DbiResult return values
DBIERR_NONE The directory was returned successfully.

DBIERR_INVALIDHNDL The specified database handle is invalid or NULL.

See also
DbiSetDirectory, DbiInit, DbiOpenDatabase

bDefault
bDefault can be one of the following values:

***bDefault* value**	**Directory to retrieve**
TRUE	Default directory
FALSE	Current working directory

C Examples: DbiGetDirectory

An example for this function is under development and will be provided in an upcoming Help release.

Delphi Examples: DbiGetDirectory

Return the current working directory.

This example uses the following input:

```
  ReturnString:= fDbiGetDirectory(Table1.DBHandle);
```

The function is defined as:

```
function fDbiGetDirectory(hDB: hDbiDb): string;
var
  Dir: string;
begin
  SetLength(Dir, dbiMaxPathLen + 1);
  Check(DbiGetDirectory(hDB, False, PChar(Dir)));
  SetLength(Dir, StrLen(PChar(Dir)));
  Result:= Dir;
end;
```

DbiGetDriverDesc {button C Examples,JI(`>example',`exdbigetdriverdesc')} {button Delphi Examples,JI(`>example',`dexdbigetdriverdesc')}

C syntax

```
DBIResult DBIFN DbiGetDriverDesc (pszDriverType, pdrvType);
```

Delphi syntax

```
function DbiGetDriverDesc (pszDriverType: PChar; var drvType: DRVType):
  DBIResult stdcall;
```

Description

DbiGetDriverDesc retrieves a description of a driver.

Parameters

pszDriverType Type: pCHAR (Input)
Pointer to the driver name string.

pdrvType Type: pDRVType (Output)
Pointer to the client-allocated DRVType structure.

DbiResult return values

DBIERR_NONE	The driver description was retrieved successfully.
DBIERR_INVALIDPARAM	Invalid *pszDriverType* argument
DBIERR_INVALIDHNDL	Invalid handles to function errors (bad cursors or database handles).

See also

DbiOpenDriverList

C Examples: DbiGetDriverDesc

Obtain information about a certain driver.

```
DBIResult fDbiGetDriverDesc(pCHAR DrvName)
{
   DBIResult       rslt;
   DRVType        DrvDesc;
   rslt = Chk(DbiGetDriverDesc(DrvName, &DrvDesc));
   return rslt;
}
```

Delphi Examples: DbiGetDriverDesc

Obtain information about a certain driver.

This example uses the following input:

```
  fDbiGetDriverDesc('INTRBASE', TmpList);
```

The procedure is defined as:

```
Procedure fDbiGetDriverDesc(DrvName: string; DriverList: TStringList);
var
   DrvDesc : DRVType;
begin
   Check(DbiGetDriverDesc(PChar(DrvName), DrvDesc));
   with DriverList do begin
     Add('Driver Type: ' + DrvDesc.szType);
     Add('Text: ' + DrvDesc.szText);
     Add('Database Type: ' + DrvDesc.szDbType);
   end;
end;
```

DbiGetErrorContext {button C Examples,JI(`>example',`exdbigeterrorcontext')} {button Delphi Examples,JI(`>example',`dexdbigeterrorcontext')}

C syntax

```
DBIResult DBIFN DbiGetErrorContext (eContext, pszContext);
```

Delphi syntax

```
function DbiGetErrorContext (eContext: SmallInt; pszContext: PChar):
  DBIResult stdcall;
```

Description

After receiving an error code back from a call, DbiGetErrorContext allows the client to probe BDE for more specific error information.

Parameters

eContext Type: INT16 (Input)
Specifies the context type.

pszContext Type: pCHAR (Output)
Pointer to the client-allocated buffer that receives the context string. The buffer must be at least as large as (DBIMAXMSGLEN+1).

Usage

DbiGetErrorContext allows the client to receive more information about the error just received, such as which table failed to open. The client inputs the error context type and the function returns a character string.

For example, a client tries to open a nonexistent table using DbiOpenTable, and receives a return of DBIERR_NOSUCHFILE. The error context is logged by the BDE. Other error contexts can be logged as well, so rather than force the user to scan each error context individually, the BDE provides a way to search for a particular context type. In this example, the user wants to know the table name associated with the error condition, and calls DbiGetErrorContext (ecTABLENAME, buffer), which returns the full path name of the table. If there is no table name associated with the error, the buffer is empty.

Note: If all that is required is a formatted error message for the end user,

DbiGetErrorInfo is a more convenient way to get it.

Prerequisites

Calls other than error handling functions may be made after the call that produced the error, but the current context information is lost. If DbiOpenTable fails, no functions that require a cursor can be called because none was returned. But other functions and another DbiOpenTable can be called.

DbiResult return values

DBIERR_NONE The error context was successfully returned.

See also

DbiGetErrorInfo, DbiGetErrorEntry, DbiGetErrorString

eContext
eContext can be one of the following values:

Value	Description
ecTOKEN	Token (For QBE)
ecTABLENAME	Table name
ecFIELDNAME	Field name
ecIMAGEROW	Image row (For QBE)
ecUSERNAME	For example, in lock conflicts, user involved
ecFILENAME	File name
ecINDEXNAME	Index name
ecDIRNAME	Directory name
ecKEYNAME	Key name
ecALIAS	Alias
ecDRIVENAME	Drive name (C:)
ecNATIVECODE	Native error code
ecNATIVEMSG	Native error message
ecLINENUMBER	Line number
ecCAPABILITY	Capability

C Examples: DbiGetErrorContext

An example for this function is under development and will be provided in an upcoming Help release.

Delphi Examples: DbiGetErrorContext

Show error context string.

After a dbi function returns an error, you can use this procedure to display the error context string associated with the context type specified in eContext. This example uses the following input:

```
  ShowErrorContext(ecTABLENAME);
```

The procedure is defined as:

```
procedure ShowErrorContext(eContext: Integer);
var
  Ctxt: string;
begin
  SetLength(Ctxt, DBIMAXMSGLEN);
  DbiGetErrorContext(eContext, PChar(Ctxt));
  SetLength(Ctxt, StrLen(PChar(Ctxt)));
  if (Ctxt > '') then
    ShowMessage(format('Error context string: %s',[Ctxt]));
end;
```

DbiGetErrorEntry {button C Examples,JI(`>example',`exdbigeterrorentry')} {button Delphi Examples,JI(`>example',`dexdbigeterrorentry')}

C syntax

```
DBIResult DBIFN DbiGetErrorEntry (uEntry, pulNativeError, pszError);
```

Delphi syntax

```
function DbiGetErrorEntry (uEntry: Word; var ulNativeError: Longint;
  pszError: PChar): DBIResult stdcall;
```

Description

DbiGetErrorEntry returns the error description (including native server errors returned from SQL systems) of a specified error stack entry.

Parameters

uEntry Type: UINT16 (Input)
Specifies the error stack entry.

pulNativeError Type: pUINT32 (Output)
Pointer to the client variable that receives the native error code (if any).

pszError Type: pCHAR (Output)
Pointer to the client-allocated buffer that receives the error string (if any).

Usage

Error stack entries begin with 1. Each stack entry contains a DBIERR, and possibly a native error code and a native error message. DBIERR_NONE is returned for stack entries beyond the current error stack, so this successful return can be used as a loop termination. For example, if error entry 1 returns DBIERR_NONE, there are no errors on the stack. Both the native error code and the native error message result are optional. The stack can be traversed multiple times, or combined with other error interface calls, but non-error routine BDE calls reset the error stack.

DbiResult return values

DBIERR_NONE The error stack entry is empty.

Any other error return value indicates what the error code is that is contained in the error stack entry.

See also

DbiGetErrorInfo, DbiGetErrorString

C Examples: DbiGetErrorEntry

An example for this function is under development and will be provided in an upcoming Help release.

Delphi Examples: DbiGetErrorEntry

Get the error for the specified entry and return the result in a ClientError string. If a native error also exists in the entry, return it as the function result. Raise an EDatabaseError exception if an attempt is made to go beyond the end of the error stack. This example uses the following input:

```
  NativeError := fDbiGetErrorEntry(1, ClientStr);
```

The function is defined as:

```
function fDbigetErrorEntry(Entry: Word; var ClientError: string): Longint;
var
  L: Longint;
  rslt: DBIResult;
begin
  SetLength(ClientError, DBIMAXMSGLEN + 1);
  rslt := DbiGetErrorEntry(Entry, L, PChar(ClientError));
  SetLength(ClientError, StrLen(PChar(ClientError)));
  Result := L;
  if (rslt = DBIERR_NONE) then
    raise EDatabaseError.Create('No errors at stack entry ' +
  IntToStr(Entry));
end;
```

DbiGetErrorInfo {button C Examples,JI(`>example',`exdbigeterrorinfo')} {button Delphi Examples,JI(`>example',`dexdbigeterrorinfo')}

C syntax
```
DBIResult DBIFN DbiGetErrorInfo (bFull, pErrInfo);
```

Delphi syntax
```
function DbiGetErrorInfo (bFull: Bool; var ErrInfo: DBIErrInfo): DBIResult
  stdcall {$ENDIF};
```

Description
DbiGetErrorInfo provides descriptive error information about the last error that occurred, and error contexts for the first four error messages on the error stack.

Parameters
bFull Type: BOOL (Input)
Not currently used.

pErrInfo Type: pDBIErrInfo (Output)
Pointer to the client DBIErrInfo structure.

Usage
Error information consists of the DBIResult error code, an error message in ANSI characters corresponding to the code, and up to four associated error contexts. For example, if the error message is "Table Not Found," the user might want to know the table name. The BDE engine logged the table name with the error context ecTABLENAME, which can be found in one of the contexts contained in the DBIErrInfo structure.

Prerequisites
This function is designed for immediate display to the user, so unlike the function DbiGetErrorContext, the client does not need to be concerned about the different types of error contexts. If the client wishes to interpret certain error codes and contexts (for example, the ALIAS error context), DbiGetErrorContext should be used.

DbiResult return values
DBIERR_NONE Error information was retrieved successfully.

See also
DbiGetErrorContext

C Examples: DbiGetErrorInfo

An example for this function is under development and will be provided in an upcoming Help release.

Delphi Examples: DbiGetErrorInfo

Get descriptive error information about the last error

In addition the the most recent error, this function displays error contexts for up to four error messages on the error stack. This example uses the following input:

```
fDbiGetErrorInfo(DbiOpenLDList(hCur), ErrorList);
```

The procedure is defined as:

```
procedure fDbiGetErrorInfo(ErrorCode: DbiResult; ErrorList: TStringList);
var
  ErrorInfo: DBIErrInfo;
  ErrorString: string;
begin
  if (ErrorCode <> dbiERR_NONE) then begin
    ErrorList.Clear;
    Check(DbiGetErrorInfo(True,ErrorInfo));
    if (ErrorCode = ErrorInfo.iError) then begin
      ErrorList.Add('Error Number: ' + IntToStr(ErrorInfo.iError));
      ErrorList.Add('Error Code: ' + StrPas(ErrorInfo.szErrcode));
      if (StrLen(ErrorInfo.szContext[1]) <> 0) then
        ErrorList.Add('Error Context1: ' + StrPas(ErrorInfo.szContext[1]));
      if (StrLen(ErrorInfo.szContext[2]) <> 0) then
        ErrorList.Add('Error Context2: ' + StrPas(ErrorInfo.szContext[2]));
      if (StrLen(ErrorInfo.szContext[3]) <> 0) then
        ErrorList.Add('Error Context3: ' + StrPas(ErrorInfo.szContext[3]));
      if (StrLen(ErrorInfo.szContext[4]) <> 0) then
        ErrorList.Add('Error Context4: ' + StrPas(ErrorInfo.szContext[4]));
    end
    else begin
      SetLength(ErrorString, dbiMaxMsgLen + 1);
      Check(DbiGetErrorString(ErrorCode, PChar(ErrorString)));
      SetLength(ErrorString, StrLen(PChar(ErrorString)));
      ErrorList.Add(ErrorString);
    end;
  end;
end;
```

DbiGetErrorString {button C Examples,JI(`>example',`exdbigeterrorstring')} {button Delphi Examples,JI(`>example',`dexdbigeterrorinfo')}

C syntax

```
DBIResult DBIFN DbiGetErrorString (rslt, pszError);
```

Delphi syntax

```
function DbiGetErrorString (rslt: DBIResult; pszError: PChar): DBIResult
  stdcall;
```

Description

DbiGetErrorString returns the message associated with a given error code.

Parameters

rslt Type: DBIResult (Input)
Specifies the error code.

pszError Type: pCHAR (Output)
Pointer to the client buffer that receives the message string for the given error code.

Usage

This function maps an error code in *rslt* to the corresponding error string. For example, if DbiGetErrorString is called with the error code DBIERR_EOF, it returns the string "At End of Table." BDE keeps the error strings as Windows string resources, so the client can translate/customize them as needed (using a resource editor such as Resource Workshop).

Note: This function has no context, so it is not limited to error codes that were returned by previous BDE calls. In contrast, DbiGetErrorInfo returns information only on the last error logged by BDE.

Prerequisites

The client must allocate a buffer at least as large as (DBIMAXMSGLEN+1).

DbiResult return values

DBIERR_NONE The error string was retrieved successfully.

See also

DbiGetErrorInfo, DbiGetErrorEntry, DbiGetErrorContext

C Examples: DbiGetErrorString

Check the BDE error stack for error information:

This example uses the following input:

fError(DbiSaveChanges(hCur));

```
DBIResult fError(DBIResult ErrorValue)
{
   char        dbi_status[DBIMAXMSGLEN * 5] = {'\0'}; // Error String
   DBIMSG      dbi_string = {'\0'};
   DBIErrInfo  ErrInfo;    // Contains information about the error

   if (ErrorValue != DBIERR_NONE)
   {
      // Note - make certain to call DbiGetErrorInfo() right after
      // the error because it will give information about only the
      // most recent error.
      DbiGetErrorInfo(TRUE, &ErrInfo);

      if (ErrInfo.iError == ErrorValue)
      {
         strcpy(dbi_status, ErrInfo.szErrCode);

         // Need to check how much information was provided -
         //   different errors return different amounts of information.
         if (strcmp(ErrInfo.szContext1, ""))
            wsprintf(dbi_status, "%s\r\n    %s", dbi_status,
  ErrInfo.szContext1);
         if (strcmp(ErrInfo.szContext2, ""))
            wsprintf(dbi_status, "%s\r\n    %s", dbi_status,
  ErrInfo.szContext2);
         if (strcmp(ErrInfo.szContext3, ""))
            wsprintf(dbi_status, "%s\r\n    %s", dbi_status,
  ErrInfo.szContext3);
         if (strcmp(ErrInfo.szContext4, ""))
            wsprintf(dbi_status, "%s\r\n    %s", dbi_status,
  ErrInfo.szContext4);
      }
      else
      {
         DbiGetErrorString(ErrorValue, dbi_string);
         strcpy(dbi_status, dbi_string);
      }
      // Display error in snipit and in a MessageBox
      MessageBox(NULL, dbi_status, "BDE Error - Example Only",
  MB_ICONEXCLAMATION);
   }
   return ErrorValue;
}
```

Delphi Examples: DbiGetErrorString

An example for this function is under development and will be provided in an upcoming Help release.

DbiGetExactRecordCount

C syntax
```
DBIResult DBIFN DbiGetExactRecordCount (hCursor, piRecCount);
```

Delphi syntax
```
function DbiGetExactRecordCount (hCursor: hDBICur; var iRecCount: Longint):
  DBIResult stdcall;
```

Description
DbiGetExactRecordCount retrieves the current exact number of records associated with the cursor.

Parameters
hCursor Type: hDBICur (Input)
Specifies the cursor handle.

piRecCount Type: pUINT32 (Output)
Pointer to the client variable which receives the number of records associated with the cursor.

Usage
This function is meant to get the exact number of records associated with the cursor.

Use DbiGetExactRecordCount instead of DbiGetRecordCount if:

1. A filter is active on the specified cursor
2. A range is active on the specified cursor (Paradox tables: use DbiGetRecordCount)
3. A live result is requested on a cursor handle from any of the DbiQ functions

Note: This function should only be used when necessary. With SQL servers, the entire result set will need to be read to determine the record count which can result in extermely slow responce. An alternative is to use a SELECT COUNT query.

DbiResult return values
DBIERR_NONE The record count was retrieved successfully.

DBIERR_INVALIDHNDL The specified cursor handle is invalid or NULL.

See also
DbiGetRecordCount

DbiGetField {button C Examples,JI(`>example',`exdbigetfield')} {button Delphi Examples,JI(`>example',`dexdbigetfield')}

C syntax

```
DBIResult DBIFN DbiGetField (hCursor, iField, pRecBuf, [pDest], [pbBlank]);
```

Delphi syntax

```
function DbiGetField (hCursor: hDBICur; iField: Word; pRecBuff: Pointer;
  pDest: Pointer; var bBlank: Bool): DBIResult stdcall;
```

Description

DbiGetField retrieves the data contents of the requested field from the record buffer.

Parameters

hCursor Type: hDBICur (Input)
Specifies the cursor handle.

iField Type: UINT16 (Input)
Specifies the ordinal number of the field within the record. Field numbers start with 1.

pRecBuf Type: pBYTE (Input)
Pointer to the record buffer.

pDest Type: pBYTE (Output)
Pointer to the client buffer that receives the data from the requested field. Optional.

pbBlank Type: pBOOL (Output)
Pointer to the client variable. Set to TRUE if the field is blank; otherwise, FALSE. Optional.

Usage

To determine if a field is blank or if a BLOB is NULL, DbiGetField can be called with *pDest* set to NULL. *pbBlank* is returned indicating whether the field is blank or nonblank.

The data that DbiGetField returns is based on the current translation mode of the cursor. If the record translation is set to xltNONE, DbiGetField returns the raw data in the driver's physical format. This is called a BDE physical type. If the translation mode is set to xltFIELD, the data is returned in a generic form (for example, a Paradox numeric value is returned as an 8-byte double). This is called a BDE logical type.

DbiGetField cannot be used to return the data contents of a BLOB field, although it can be used to determine if the BLOB field is empty.

Completion state

The output buffer pointed to by *pDest* (if supplied) contains the requested field. The output buffer pointed to by *pbBlank* (if supplied) indicates whether the field is blank.

DbiResult return values

DBIERR_NONE Data contents were retrieved successfully.

DBIERR_INVALIDHNDL The specified cursor handle is invalid or NULL.

See also

DbiPutField, DbiInsertRecord, DbiGetNextRecord, DbiGetPriorRecord, DbiGetRelativeRecord, DbiGetRecord

C Examples: DbiGetField

Example 1: Get the field value by field number.

This example uses the following input:

fDbiGetField1(hPXCur, pPXRecBuf, 1, (pBYTE)&DFloat);

```
DBIResult fDbiGetField1(hDBICur hTmpCur, pBYTE pTmpRecBuf, INT16 FldNum,
  pBYTE Info)
{
   DBIResult      rslt;
   rslt = Chk(DbiGetField(hTmpCur, FldNum, pTmpRecBuf, Info, NULL));
   return rslt;
}
```

Example 2: Get the field value specified by a field name.

If *Info* is NULL, this function will check to see if a field exists. If an invalid field name is given, an error is returned. This example uses the following input:

fDbiGetField2(hPXCur, pPXRecBuf, "STOCK NO", (pBYTE)&DFloat);

```
DBIResult fDbiGetField2(hDBICur hTmpCur, pBYTE pTmpRecBuf, pCHAR FldName,
  pBYTE Info)
{
   DBIResult      rslt;
   CURProps       CurProps;
   pFLDDesc       pFldDesc;
   UINT16         Field;
   BOOL           Found = FALSE;
   rslt = Chk(DbiGetCursorProps(hTmpCur, &CurProps));
   if (rslt != DBIERR_NONE)
      return rslt;
   pFldDesc = (pFLDDesc)malloc(CurProps.iFields * sizeof(FLDDesc));
   if (pFldDesc == NULL)
      return DBIERR_NOMEMORY;
   rslt = Chk(DbiGetFieldDescs(hTmpCur, pFldDesc));
   if (rslt != DBIERR_NONE)
   {
      free(pFldDesc);
      return rslt;
   }
   for(Field = 0; Field < CurProps.iFields; Field++)
   {
      if (strcmpi(pFldDesc[Field].szName, FldName) == 0)
      {
         Found = TRUE;
         if (Info != NULL)
            rslt = Chk(DbiGetField(hTmpCur, pFldDesc[Field].iFldNum,
  pTmpRecBuf, Info, NULL));
      }
   }
   if (Found == FALSE)
     rslt = DBIERR_INVALIDFIELDNAME;
   free(pFldDesc);
   return rslt;
}
```

Delphi Examples: DbiGetField

Retrieve the data contents of the requested field from the record buffer:
Delphi users should not need to directly call dbiGetField because Delphi provides a variety of ways to retrieve the value of a particular field. Use the Delphi online help to browse the Value and As... properties of TField. Also see the FieldValues[] array property of TTable.

Get a field in a table and return it in a Variant type variable.
Some field types are not supported and will cause an exception. Most Delphi users should use TField objects to retrieve table information. This example uses the following input:

```
  MStr := fDbiGetField(Table1.Handle, Table1.Fields[0].Index + 1);
```

The function is:

```
function fDbiGetField(hTmpCur: hDBICur; FieldNo: Word): Variant;
var
  Props: CURProps;
  pFlds, pOldFlds: pFLDDesc;
  pRecBuf: pBYTE;
  FieldString: string;
  FieldINT16: Smallint;
  FieldINT32: Longint;
  FieldUINT16: Word;
  FieldFLOAT: Double;
  Blank: Boolean;
begin
  if (FieldNo < 1) then
    raise EDatabaseError.Create('Field number index is 1 based');

  Check(DbiGetCursorProps(hTmpCur, Props));
  pFlds := AllocMem(Props.iFields * sizeof(FLDDesc));
  pOldFlds := pFlds;
  pRecBuf := AllocMem(Props.iRecBufSize * sizeof(BYTE));
  try
    Check(DbiGetFieldDescs(hTmpCur, pFlds));
    Inc(pFlds, FieldNo - 1);
    Check(DbiGetRecord(hTmpCur, dbiNOLOCK, pRecBuf, nil));
    case pFlds.iFldType of
      fldDATE, fldTIME, fldTIMESTAMP, fldUNKNOWN, fldBLOB, fldBOOL, fldBCD:
        raise EDBEngineError.Create(DBIERR_NOTSUPPORTED);
      fldZSTRING:
      begin
        SetLength(FieldString, pFlds.iUnits1 + 1);
        Check(DbiGetField(hTmpCur, FieldNo, pRecBuf,
          pBYTE(PChar(FieldString)), Blank));
        SetLength(FieldString, StrLen(PChar(FieldString)));
        Result := FieldString;
      end;
      fldINT16:
      begin
        Check(DbiGetField(hTmpCur, FieldNo, pRecBuf, pBYTE(@FieldINT16),
          Blank));
        Result := FieldINT16;
      end;
      fldUINT16:
      begin
```

```
        Check(DbiGetField(hTmpCur, FieldNo, pRecBuf, pBYTE(@FieldUINT16),
          Blank));
        Result := FieldUINT16;
      end;
      fldFLOAT:
      begin
        Check(DbiGetField(hTmpCur, FieldNo, pRecBuf, pBYTE(@FieldFLOAT),
          Blank));
        Result := FieldFLOAT;
      end;
      fldINT32, fldUINT32:
      begin
        Check(DbiGetField(hTmpCur, FieldNo, pRecBuf, pBYTE(@FieldINT32),
          Blank));
        Result := FieldINT32;
      end;
    end;
  finally
    FreeMem(pOldFlds, Props.iFields * sizeof(FLDDesc));
    FreeMem(pRecBuf, Props.iRecBufSize * sizeof(BYTE));
  end;
end;
```

DbiGetFieldDescs {button C Examples,JI(`>example',`exdbiputfield')} {button Delphi Examples,JI(`>example',`dexdbigetfielddescs')}

C syntax

```
DBIResult DBIFN DbiGetFieldDescs (hCursor, pfldDesc);
```

Delphi syntax

```
function DbiGetFieldDescs (hCursor: hDBICur; pfldDesc: pFLDDesc): DBIResult
  stdcall;
```

Description

DbiGetFieldDescs retrieves a list of descriptors for all the fields in the table associated with *hCursor*. The structure returned is identical in form to the fields contained in DbiOpenFieldList.

Parameters

hCursor Type: hDBICur (Input)
Specifies the cursor handle.

pfldDesc Type: pFLDDesc (Output)
Pointer to the client FLDDesc structures, one for each of the fields in the table associated with the specified cursor.

Usage

The field descriptors returned are in accordance with the translation mode set for the cursor. If the translation mode is xltNONE, the physical field descriptors are returned. If the translation mode is xltFIELD, the logical field descriptors are returned.

Use DbiGetCursorProps to get the number of field in the table.

DbiResult return values

DBIERR_NONE The field Descriptions were returned successfully.

DBIERR_INVALIDHNDL The specified cursor handle is invalid or NULL.

See also

DbiGetCursorProps, DbiOpenFieldList

Delphi Examples: DbiGetFieldDescs

Retrieve a list of descriptors for all fields in the table associated with the given TTable:

This function prints out the field numbers and names of the table. This example uses the following input:

```
  fDbiGetFieldDescs(Table1);
```

The procedure is:

```
procedure ShowFields(T: TTable);
var
   curProp: CURProps;
   pfldDes, pCurFld: pFLDDesc;
   // pfldDes is a pointer to a list of field descriptors.
   // It must be allocated with (iFields * sizeof(FLDDesc))
   // where iFields is a field in the structure curProps
   // from DbiGetCursorProps
   // pCurFld is a pointer the description of one field in the list.
   i: Integer;          // counter
   MemSize: Integer;
   FieldList: string;
begin
  Check(DbiGetCursorProps(T.Handle, curProp));
  // Get enough memory for one field desc times the # of fields
  MemSize := curProp.iFields * SizeOf(FLDDesc);
  pfldDes := AllocMem(MemSize);
  try
    pCurFld := pfldDes;
    Check(DbiGetFieldDescs(T.Handle, pfldDes));
    I := 0;
    FieldList := '';
    while (i < curProp.iFields) do begin
      FieldList := FieldList + Format('%d - %s'#13#10,[pCurFld^.iFldNum,
        pCurFld^.szName]);
      // increment pointer to the next record
      inc(pCurFld);
      inc(i);
    end;
  finally
    ShowMessage(FieldList);
    FreeMem(pfldDes, MemSize);
  end;
```

DbiGetFieldTypeDesc {button C Examples,JI(`>example',`exdbigetfieldtypedesc')} {button Delphi Examples,JI(`>example',`dexdbigetfieldtypedesc')}

C syntax

```
DBIResult DBIFN DbiGetFieldTypeDesc (pszDriverType, pszTableType,
  pszFieldType, pfldType);
```

Delphi syntax

```
function DbiGetFieldTypeDesc (pszDriverType: PChar; pszTableType: PChar;
  pszFieldType: PChar; var fldType: FLDType): DBIResult stdcall;
```

Description

DbiGetFieldTypeDesc retrieves a description of the specified field type.

Parameters

pszDriverType Type: pCHAR (Input)
Pointer to the driver type. Use DbiOpenDriverList to find the valid driver types.

pszTableType Type: pCHAR (Input)
Pointer to the table type. Use DbiOpenTableTypesList to find the valid table types.

pszFieldType Type: pCHAR (Input)
Pointer to the field type. Use DbiOpenFieldTypesList to find the valid field types.

pfldType Type: pFLDType (Output)
Pointer to the client FLDType structure.

DbiResult return values

DBIERR_NONE The field type Description was retrieved successfully.

See also

DbiOpenFieldTypesList, DbiOpenTableTypesList, DbiOpenDriverList

C Examples: DbiGetFieldTypeDesc

Obtain the descriptor for a possible field type.

```
DBIResult fDbiGetFieldTypeDesc(hDBICur hTmpCur)
{
   DBIResult      rslt;
   hDBICur        hTmpListCur;
   FLDType        fldType;
   FLDDesc        fldDesc;
   TBLType        tblType;
   pCHAR          DrvType = szPARADOX;
   pCHAR          info;
   rslt = Chk(DbiOpenTableTypesList(DrvType, &hTmpListCur));
   rslt = DbiSetToBegin(hTmpListCur);
   rslt = Chk(DbiGetNextRecord(hTmpListCur, dbiNOLOCK,
                               (pBYTE)&tblType, NULL));
   info = (pCHAR)malloc(DBIMAXMSGLEN);
   rslt = Chk(DbiSetToBegin(hTmpCur));
   rslt = Chk(DbiGetNextRecord(hTmpCur, dbiNOLOCK,
                               (pBYTE)&fldDesc, NULL));
   rslt = Chk(DbiGetFieldTypeDesc(DrvType, (pCHAR)tblType.szName,
                                  (pCHAR)fldDesc.szName, &fldType));
   info[0] = '\0';
   strcat(info, "\r\n\r\n");
   strcat(info, tblType.szName);
   strcat(info, ":\r\n");
   strcat(info, fldType.szName);
   strcat(info, ": ");
   strcat(info, fldType.szText);
   return rslt;
}
```

Delphi Examples: DbiGetFieldTypeDesc

Retrieve a description of the specified field type.

This example uses the following input:

```
fDbiGetFieldTypeDesc(szPARADOX, 'PDOX 7.0', 'ALPHA', MyFieldType);
```

The procedure is:

```
procedure fDbiGetFieldTypeDesc(DriverType, TableType, FieldType: PChar;
  var FieldTypeInfo: TStringList);

  function BoolVal(InBool: Boolean): string;
  begin
    if InBool then Result:= 'True'
    else Result:= 'False';
  end;

var
  FieldTypeRec: FLDType;
begin
  Check(DbiGetFieldTypeDesc(DriverType, TableType, FieldType,
  FieldTypeRec));
  FieldTypeInfo.Add
    ('Field ID Type: ' + IntToStr(FieldTypeRec.iId));
  FieldTypeInfo.Add('Symbolic Name: ' + StrPas(FieldTypeRec.szName));
  FieldTypeInfo.Add('Descriptive Text: ' + StrPas(FieldTypeRec.szText));
  FieldTypeInfo.Add('Physical / Native Type: ' + IntToStr
  (FieldTypeRec.iPhyType));
  FieldTypeInfo.Add('Default Translated Type: ' + IntToStr
  (FieldTypeRec.iXltType));
  FieldTypeInfo.Add('Default Translated Subtype: ' + IntToStr
  (FieldTypeRec.iXltSubType));
  FieldTypeInfo.Add('Maximum Units Allowed (1): ' + IntToStr
  (FieldTypeRec.iMaxUnits1));
  FieldTypeInfo.Add('Maximum Units Allowed (2): ' + IntToStr
  (FieldTypeRec.iMaxUnits2));
  FieldTypeInfo.Add('Physical Size: ' + IntToStr (FieldTypeRec.iPhySize));
  FieldTypeInfo.Add('Field Required: ' + BoolVal(FieldTypeRec.bRequired));
  FieldTypeInfo.Add('Supports user-specified default: ' +
  BoolVal(FieldTypeRec.bDefaultVal));
  FieldTypeInfo.Add('Supports Min Val constraint: ' +
  BoolVal(FieldTypeRec.bMinVal));
  FieldTypeInfo.Add('Supports Max Val constraint: ' +
  BoolVal(FieldTypeRec.bMaxVal));
  FieldTypeInfo.Add('Supports Referential Integerity: ' +
  BoolVal(FieldTypeRec.bRefIntegrity));
  FieldTypeInfo.Add('Supports Other Checks: ' +
  BoolVal(FieldTypeRec.bOtherChecks));
  FieldTypeInfo.Add('Can Be Keyed: ' + BoolVal(FieldTypeRec.bKeyed));
  FieldTypeInfo.Add('Multiple Fields of this Type: ' +
  BoolVal(FieldTypeRec.bMultiplePerTable));
  FieldTypeInfo.Add('Minimum Units Required (1): ' + IntToStr
  (FieldTypeRec.iMinUnits1));
  FieldTypeInfo.Add('Minimum Units Required (2): ' + IntToStr
  (FieldTypeRec.iMinUnits2));
  FieldTypeInfo.Add('Field Type Can be Created: ' +
```

```
  BoolVal(FieldTypeRec.bCreateable));
end;
```

DbiGetFilterInfo {button C Examples,JI(`>example',`exdbigetfilterinfo')} {button Delphi Examples,JI(`>example',`dexdbigetfilterinfo')}

C syntax

```
DBIResult DBIFN DbiGetFilterInfo (hCursor, hFilter, iFilterId, iFilterSeqNo,
  pFilterinfo);
```

Delphi syntax

```
function DbiGetFilterInfo (hCur: hDBICur; hFilter: hDBIFilter; iFilterId:
  Word; iFilterSeqNo: Word; var Filterinfo: FilterInfo): DBIResult stdcall;
```

Description

DbiGetFilterInfo retrieves information about a specified filter.

Parameters

hCursor Type: hDBICur (Input)
Specifies the cursor handle.

hFilter Type: hDBIFilter (Input)
Specifies the filter handle. Filter handles are not preserved for cloned cursors. Optional, specify a filter handle, filter identification number, or filter sequence number to identify the filter. The default is NULL.

iFilterId Type: UINT16 (Input)
Specifies the filter identification number. Optional, specify a filter handle, filter identification number, or filter sequence number to identify the filter. The default is 0.

iFilterSeqNo Type: UINT16 (Input)
Specifies the filter sequence number. Optional, specify a filter handle, filter identification number, or filter sequence number to identify the filter. The default is 0.

pFilterinfo Type: pFILTERInfo (Output)
Pointer to the client FILTERInfo structure.

DbiResult return values

DBIERR_NONE	Filter information was retrieved successfully.
DBIERR_INVALIDHNDL	The specified cursor handle is invalid or NULL.

C Examples: DbiGetFilterInfo

An example for this function is under development and will be provided in an upcoming Help release.

Delphi Examples: DbiGetFilterInfo

Return the filter information for the specified table and filter handles. This example uses the following input:

```
  FInfo := fDbiGetFilterInfo(CustomerTbl.Handle, hFilter);
```

The function is:

```
function fDbiGetFilterInfo(hTmpCur: hDBICur; hFilter: hDBIFilter):
  FILTERInfo;
var
  Props: CURProps;
begin
  Check(DbiGetCursorProps(hTmpCur, Props));
  if (Props.iFilters = 0) then
    raise EDatabaseError.Create('Ther is not filter associated with the
  cursor');
  Check(DbiGetFilterInfo(hTmpCur, hFIlter, 0, 0, Result));
end;
```

DbiGetIndexDesc {button C Examples,JI(`>example',`exdbigetindexdesc')} {button Delphi Examples,JI(`>example',`dexdbigetindexdesc')}

C syntax

```
DBIResult DBIFN DbiGetIndexDesc (hCursor, iIndexSeqNo, pidxDesc);
```

Delphi syntax

```
Function DbiGetIndexDesc (hCursor: hDBICur; iIndexSeqNo: Word; var idxDesc:
  IDXDesc): DBIResult stdcall;
```

Description

DbiGetIndexDesc retrieves the properties of the given index associated with *hCursor*.

Parameters

hCursor Type: hDBICur (Input)
Specifies the cursor handle.

iIndexSeqNo Type: UINT16 (Input)
Specifies the ordinal number of the index in the list of open indexes of the cursor. DbiGetIndexSeqNo can be called to obtain this number for a given index. If *iIndexSeqNo* is 0, the properties of the active index are returned.

pidxDesc Type: pIDXDesc (Output)
Pointer to the client-allocated IDXDesc structure.

Usage

This function is used to find the properties of an open index for this cursor. Use DbiGetCursorProps to get the number of open indexes (iIndexes). *iIndexSeqNo* must be between zero and iIndexes.

Note: If a field map is active, the field numbers in *aiKeyFld* list the mapped field numbers, however, if a key field is not part of the field map, it is a negative number.

Oracle: For performance reasons, bPrimary is not set in the FLDDesc structure. To determine if a primary index exists on a table, use DbiSetProp with the curGETEXTENDEDINFO property before calling DbiGetIndexDescs.

Prerequisites

A valid cursor handle must be on one or more open indexes.

DbiResult return values

DBIERR_NONE	The properties of the specified index were returned successfully.
DBIERR_INVALIDHNDL	The specified handle is invalid or NULL.
DBIERR_NOTINDEXED	Table has no associated indexes.
DBIERR_NOSUCHINDEX	*iIndexSeqNo* is invalid.

See also

DbiOpenIndex, DbiCloseIndex, DbiGetCursorProps, DbiGetIndexSeqNo, DbiOpenIndexList, DbiGetIndexDescs

C Examples: DbiGetIndexDesc

Get the name and the amount of fields for the index open on the current cursor.
IXDesc must be of sufficient size to hold the index description. This example uses the following input:

fDbiGetIndexDesc(hCur, Buffer);

```
DBIResult fDbiGetIndexDesc(hDBICur hTmpCur, pCHAR IXDesc)
{
   DBIResult   rslt;
   IDXDesc     IdxDesc;
   rslt = Chk(DbiGetIndexDesc(hTmpCur, 0, &IdxDesc));
   wsprintf(IXDesc, "Index name: %s; Fields in Key: %d", IdxDesc.szName,
               IdxDesc.iFldsInKey);
   return rslt;
}
```

Delphi Examples: DbiGetIndexDesc

Get the properties of a specific index associated with a cursor:
This function returns the IDXDesc properties specified by the IndexName parameter of TTable T's index.

```
function GetIndexDesc(T: TTable; IndexName: string): IDXDesc;
var
  hNewCur: hDbiCur;
  iIndexId: LongInt;
  InfoStr: string;
  pInfoStr: array[0..100] of char;
begin
  Check(DbiCloneCursor(T.Handle, False, False, hNewCur));
  try
    iIndexId := 1;
    Check(DbiSwitchToIndex(hNewCur, PChar(IndexName), nil, iIndexId,
  False));
    Check(DbiGetIndexDesc(hNewCur, 0, Result)); //'0' specifies the active
  index
  finally
    Check(DbiCloseCursor(hNewCur));
  end;
end;
```

DbiGetIndexDescs {button C Examples,JI(`>example',`exdbigetindexdescs')} {button Delphi Examples,JI(`>example',`dexdbigetindexdescs')}

C syntax

```
DBIResult DBIFN DbiGetIndexDescs (hCursor, pidxDesc);
```

Delphi syntax

```
function DbiGetIndexDescs (hCursor: hDBICur; var idxDesc: IDXDesc):
  DBIResult stdcall;
```

Description

DbiGetIndexDescs retrieves index properties for all indexes associated with this cursor.

Parameters

hCursor Type: hDBICur (Input)
Specifies the cursor handle.

pidxDesc Type: pIDXDesc (Output)
Pointer to the client-allocated IDXDesc structure.

Usage

The client must allocate a buffer large enough to hold all index descriptors. The number of indexes can be obtained by using DbiGetCursorProps and examining the iIndexes property.

dBASE: DbiGetIndexDescs won't include dBASE non-production indexes if the indexes are not opened.

Oracle: For performance reasons, bPrimary is not set in the FLDDesc structure. To determine if a primary index exists on a table, use DbiSetProp with the curGETEXTENDEDINFO property before calling DbiGetIndexDescs.

Prerequisites

A valid cursor handle must be obtained, and at least one index must exist.

DbiResult return values

DBIERR_NONE Index Descriptions were returned successfully.

DBIERR_INVALIDHNDL The specified handle is invalid or NULL.

See also

DbiGetIndexDesc, DbiOpenIndex, DbiCloseIndex, DbiGetIndexSeqNo, DbiGetCursorProps, DbiOpenIndexList, DbiGetIndexForField

C Examples: DbiGetIndexDescs

Get the properties of all the indexes for the table open with the specified cursor

IndexDesc is allocated within the function. This example uses the following input:

fDbiGetIndexDescs(hCur, &pIdxDesc);

```
DBIResult fDbiGetIndexDescs(hDBICur hTmpCur, pIDXDesc *IndexDesc)
{
   DBIResult   rslt;
   CURProps    CurProps;
   rslt = Chk(DbiGetCursorProps(hTmpCur, &CurProps));
   if (rslt != DBIERR_NONE)
      return rslt;
   *IndexDesc = (pIDXDesc)malloc(CurProps.iIndexes * sizeof(IDXDesc));
   if (IndexDesc == NULL)
     return DBIERR_NOMEMORY;
   rslt = Chk(DbiGetIndexDescs(hTmpCur, *IndexDesc));
   return rslt;
}
```

Delphi Examples: DbiGetIndexDescs

Get the properties of all indexes for TTable T:

This function loops through all the indexes and shows the names and fields in the key. This example uses the following input:

```
  ShowIndexDescs(Table1);
```

The procedure is:

```
procedure ShowIndexDescs(T: TTable);
const
  IDXStr = '%sIndex name: %s. Number of fields in key: %d'#13#10;
var
  CurProp: CURProps;
  pIndexDesc, pTmpMem: pIdxDesc;
  i, MemSize: Integer;
  ShowString, IDXName: string;
begin
  Check(DbiGetCursorProps(T.Handle, CurProp));
  MemSize := CurProp.iIndexes * sizeof(IDXDesc);
  pIndexDesc := AllocMem(MemSize);
  try
    pTmpMem := pIndexDesc;
    Check(DbiGetIndexDescs(T.Handle, pIndexDesc));
    i := 0;
    ShowString := '';
    while (i < curProp.iIndexes) do begin
      with pTmpMem^ do begin
        // primary index does not have a name for PARADOX tables }
        if bPrimary and (StrComp(curProp.szTableType, szParadox) = 0) then
          IDXName := 'Primary'
        else
          IDXName := szName;
        ShowString := Format(IDXStr, [ShowString, IDXName, iFldsInKey])
      end;
      // increment pointer to the next record
      inc(pTmpMem);
      inc(i);
    end;
  finally
    FreeMem(pIndexDesc, MemSize);
    ShowMessage(ShowString);
  end;
end;
```

DbiGetIndexForField {button C Examples,JI(`>example',`exdbigetindexforfield')} {button Delphi Examples,JI(`>example',`dexdbigetindexforfield')}

C syntax

```
DBIResult DBIFN DbiGetIndexForField (hCursor, iFld, bProdTagOnly,
  [pidxDesc]);
```

Delphi syntax

```
function DbiGetIndexForField (hCursor: hDBICur; iFld: Word; bProdTagOnly:
  Bool; var idxDesc: IDXDesc): DBIResult stdcall;
```

Description

DbiGetIndexForField returns the description of any useful index on the specified field. You can also use it just to check if an index exists for the given field.

Parameters

hCursor Type: hDBICur (Input)
Specifies the cursor handle.

iFld Type: UINT16 (Input)
Specifies the field number.

bProdTagOnly Type: BOOL (Input)
For dBASE only. If set to TRUE, only dBASE production tags are searched.

pidxDesc Type: pIDXDesc (Output)
Pointer to the client-allocated IDXDesc structure.

Usage

Paradox: If multiple indexes exist on the field, the following order of precedence is followed: primary index, secondary index on the specified field only, and secondary composite index with the specified field as the first component.

dBASE or FoxPro: For dBASE or FoxPro tables, only simple indexes are considered because there are no composite indexes. Expression indexes are not considered.

Access: The first index found is used.

SQL: For SQL tables, if multiple indexes are created for the field, the first useful index is returned. (An attempt is made to return the unique index with the least number of fields in the key. If there is no unique index, an index with the least number of fields in the key is returned.)

Prerequisites

A valid cursor handle must be obtained on a base table, not on a query or in-memory or temporary table.

Completion state

The index Description is returned in the specified IDXDesc structure.

DbiResult return values

DBIERR_NONE	The index descriptors were returned successfully.
DBIERR_INVALIDHNDL	The specified handle is invalid or NULL.
DBIERR_NOSUCHINDEX	No index on this field.

See also

DbiOpenIndex, DbiCloseIndex, DbiDeleteIndex, DbiAddIndex

C Examples: DbiGetIndexForField

Get the Index descriptor for the current field (if any).

If IdxDesc is null, this function can be used to check if an index exists. It returns DBIERR_NOSUCHINDEX if no index exists. This example uses the following input:

fDbiGetIndexForField(hPXCur, 1, &IdxDesc);

```
DBIResult fDbiGetIndexForField(hDBICur hTmpCur, INT16 Field, pIDXDesc
  IdxDesc)
{
   DBIResult   rslt;
   rslt = DbiGetIndexForField(hTmpCur, Field, FALSE, IdxDesc);
   if (rslt == DBIERR_NOSUCHINDEX)
      return rslt;
   else
      Chk(rslt);
   return rslt;
}
```

Delphi Examples: DbiGetIndexForField

Return the description of any useful index on the specified field.

You can also use this function can just to check if an index exists for the given field. When you pass a handle of the table, a valid field number, and a TStringList, the procedure appends the information accessed from a IdxDesc Record to the TStringList. This example uses the following input:

```
fDbiGetIndexForField(DBASEAnimals.handle, 1, False, MyIndexInfo);
```

The procedure is:

```
procedure fDbiGetIndexForField(hCursor: hDBICur; Field: TField; IndexInfo:
  TStringList);

  function BoolVal(InBool: Boolean): String;
  begin
    if InBool then Result:= 'True'
    else Result:= 'False';
  end;

var
  KeyArray: string;
  x: Word;
  MyidxDesc: IdxDesc;
begin
  Check(DbiGetIndexForField(hCursor, Field.Index + 1, True, MyidxDesc));
  with IndexInfo do begin
    Add('Index Name: ' + MyidxDesc.szname);
    Add('Index Number: ' + IntToStr(MyidxDesc.iIndexId));
    Add('Tag Name (dBASE): ' + MyidxDesc.szTagName);
    Add('Index Format: ' + MyidxDesc.szformat);
    Add('Primary: ' + BoolVal(MyidxDesc.bPrimary));
    Add('Descending: ' + BoolVal(MyidxDesc.bDescending));
    Add('Maintained: ' + BoolVal(MyidxDesc.bMaintained));
    Add('Subset: ' + BoolVal(MyidxDesc.bSubset));
    Add('ExpIdx: ' + BoolVal(MyidxDesc.bExpIdx));
    Add('Fields In Key: ' + IntToStr(MyidxDesc.iFldsInKey));
    Add('Key Length: ' + IntToStr(MyidxDesc.iKeyLen));
    Add('Out of Date: ' + BoolVal(MyidxDesc.bOutofDate));
    Add('Key Expression Type: ' + IntToStr(MyidxDesc.iKeyExpType));
    for x:= 0 to (MyidxDesc.iFldsInKey -1) do
      KeyArray:= KeyArray + IntToStr(MyidxDesc.aiKeyFld[x]) + ', ';
    Add('Field Numbers used in Key: ' + KeyArray);
    Add('Key Expression: ' + MyidxDesc.szKeyExp);
    Add('Key Condition: ' + MyidxDesc.szKeyCond);
    Add('Case Insensitive: ' + BoolVal(MyidxDesc.bCaseInsensitive));
    Add('iBlockSize: ' + IntToStr(MyidxDesc.iBlockSize));
    Add('iRestrNum: ' + IntToStr(MyidxDesc.iRestrNum));
  end;
end;
```

DbiGetIndexSeqNo {button C Examples,JI(`>example',`exdbigetindexseqno')} {button Delphi Examples,JI(`>example',`dexdbigetindexseqno')}

C syntax

```
DBIResult DBIFN DbiGetIndexSeqNo (hCursor, pszIndexName, pszTagName,
  iIndexId, piIndexSeqNo);
```

Delphi syntax

```
function DbiGetIndexSeqNo (hCursor: hDBICur; pszIndexName: PChar;
  pszTagName: PChar; iIndexId: Word; var iIndexSeqNo: Word): DBIResult
  stdcall;
```

Description

DbiGetIndexSeqNo retrieves the ordinal number of the index in the index list of the specified cursor.

Parameters

hCursor Type: hDBICur (Input)
Specifies the cursor handle.

pszIndexName Type: pCHAR (Input)
Pointer to the index name.

pszTagName Type: pCHAR (Input)
For dBASE and FoxPro only. Pointer to the index tag name.

iIndexId Type: UINT16 (Input)
Specifies the index ID, if required to identify an index.

p*iIndexSeqNo* Type: pUINT16 (Output)
Pointer to the client variable which receives the index sequence number.

Usage

dBASE or FoxPro: For dBASE or FoxPro tables, the ordinal number of the index in the index list can be affected by the opening and closing of indexes on the cursor. *pszIndexName* and *pszTagName* are used to specify the index.

Paradox, Access: The index can be specified by name or ID.

SQL: The index must be specified by name.

Completion state

The sequence number of the specified index is returned. The result of this function can be used as input for DbiGetIndexDesc.

DbiResult return values

DBIERR_NONE	The index sequence number was returned successfully.
DBIERR_INVALIDHNDL	The specified handle is invalid or NULL.
DBIERR_NOSUCHINDEX	The index is not open, or does not exist.

See also

DbiGetIndexDesc

C Examples: DbiGetIndexSeqNo

An example for this function is under development and will be provided in an upcoming Help release.

Delphi Examples: DbiGetIndexSeqNo

Return the index sequence number for the current index. This example only works with Paradox and SQL tables and uses the following input:

```
  IXNum := fDbiGetIndexSeqNo(InterBaseCustTbl);
```

The function is:

```
function fDbiGetIndexSeqNo(Table: TTable): Word;
begin
  Check(DbiGetIndexSeqNo(Table.Handle, PChar(Table.IndexName), nil, 0,
  Result));
end;
```

DbiGetIndexTypeDesc {button C Examples,JI(`>example',`exdbigetindextypedesc')} {button Delphi Examples,JI(`>example',`dexdbigetindextypedesc')}

C syntax

```
DBIResult DBIFN DbiGetIndexTypeDesc (pszDriverType, pszIndexType, pidxType);
```

Delphi syntax

```
function DbiGetIndexTypeDesc (pszDriverType: PChar; pszIndexType: PChar; var
  idxType: IDXType): DBIResult stdcall;
```

Description

DbiGetIndexTypeDesc retrieves a description of the index type.

Parameters

pszDriverType Type: pCHAR (Input)
Pointer to the driver type.

pszIndexType Type: pCHAR (Input)
Pointer to the index type. Use DbiOpenIndexTypesList to find the valid index types.

pidxType Type: pIDXType (Output)
Pointer to the client-allocated IDXType structure.

DbiResult return values

DBIERR_NONE The index type description was returned successfully.

See also

DbiOpenIndexTypesList

C Examples: DbiGetIndexTypeDesc

An example for this function is under development and will be provided in an upcoming Help release.

Delphi Examples: DbiGetIndexTypeDesc

An example for this function is under development and will be provided in an upcoming Help release.

DbiGetLdName {button C Examples,JI(`>example',`exdbigetldname')} {button Delphi Examples,JI(`>example',`dexdbigetldname')}

C syntax

```
DBIResult DBIFN DbiGetLdName (pszDriver, pObjName, pLdName);
```

Delphi syntax

```
function DbiGetLdName (pszDriver: PChar; pObjName: PChar; pLdName: PChar):
  DBIResult stdcall;
```

Description

DbiGetLdName retrieves the name of the language driver associated with the specified object name (table name).

Parameters

pszDriver Type: pCHAR (Input)
Pointer to the driver name.

pObjName Type: pCHAR (Input)
Pointer to the table name.

pLdName Type: pCHAR (Output)
Pointer to the client buffer that receives the language driver name associated with the specified table. This buffer should be at least (DBIMAXNAMELEN + 1) in size.

Usage

If *pObjName* is NULL, the name of the driver's default language driver is returned.

Standard: The returned language driver name can be used as an optional parameter for DbiCreateTable as a way to override the default language driver at create time.

SQL: If *pObjName* is not NULL, it must be of the form :dbalias:objName.

DbiResult return values

DBIERR_NONE The name of the language driver was retrieved successfully.

See also

DbiCreateTable

C Examples: DbiGetLdName

An example for this function is under development and will be provided in an upcoming Help release.

Delphi Examples: DbiGetLdName

Obtain the current dBASE language driver and display its name in a dialog box.

```
procedure fDbiGetLDName;
var
  S: string;
begin
  SetLength(S, dbiMaxNameLen + 1);
  Check(DbigetLDName(szDBASE, nil, PChar(S)));
  SetLength(S, StrLen(PChar(S)));
  ShowMessage('Current dBase Language driver is ' + S);
end;
```

DbiGetLdObj {button C Examples,JI(`>example',`exdbigetldobj')} {button Delphi Examples,JI(`>example',`dexdbigetldobj')}

C syntax

```
DBIResult DBIFN DbiGetLdObj (hCursor,*ppLdObj);
```

Delphi syntax

```
function DbiGetLdObj (hCursor: hDBICur; var pLdObj: Pointer): DBIResult
  stdcall;
```

Description

DbiGetLdObj returns the language driver object associated with the given cursor.

Parameters

hCursor Type: hDBICur (Input)
Specifies the cursor handle.

**ppLdObj* Type: pVOID (Output)
Pointer to the client variable that receives the pointer to the language driver.

Usage

The object pointer returned in this function can be used with DbiNativeToAnsi and DbiAnsiToNative.

Completion state

If a valid cursor is passed to this function, the returned object pointer has a lifetime equivalent to the cursor's lifetime. In other words, if the cursor is closed (and no other cursors are open on the same table), the language driver object is destroyed and can no longer be accessed through this object pointer.

If the *hCursor* parameter is NULL, a pointer to the system language driver is returned. This pointer is valid for the duration of the session and can be used regardless of which cursors are opened or closed.

DbiResult return values

DBIERR_NONE The language driver object was returned successfully.

See also

DbiNativeToAnsi, DbiAnsiToNative

C Examples: DbiGetLdObj

An example for this function is under development and will be provided in an upcoming Help release.

Delphi Examples: DbiGetLdObj

Obtain the language driver information for TDataSet descendant D.
TStrings LdObjList is filled with the language driver information. This example uses the following input:

```
fDbiGetLdObj(Table1, Listbox1.Items);
```

The procedure is:

```
procedure fDbiGetLdObj(D: TDataSet; LdObjList: TStrings);
var
  MypLdObj: pLDDesc;
begin
  Check(DbiGetLdObj(D.Handle, Pointer(MypLdObj)));
  with MypLdObj^, LdObjList do begin
    Add(Format('Name: %s', [szName]));
    Add(Format('Description: %s', [szDesc]));
    Add(Format('Code Page: %d', [iCodePage]));
    case PrimaryCpPlatform of
      1: Add('Primary Platform: DOS(OEM) platform');
      2: Add('Primary Platform: Windows (ANSI) platform');
      6: Add('Primary Platform: HP UNIX (ROMAN8) platform');
    else
      Add(Format('Primary Platform: Other (%d)', [PrimaryCpPlatform]));
    end;
  end;
end;
```

DbiGetLinkStatus {button C Examples,JI(`>example',`exdbigetlinkstatus')} {button Delphi Examples,JI(`>example',`dexdbigetlinkstatus')}

C syntax

```
DBIResult DBIFN DbiGetLinkStatus (hCursor, phCursorMstr, phCursorDet,
  phCursorSib);
```

Delphi syntax

```
function DbiGetLinkStatus (hCursor: hDBICur; var hCursorMstr: hDBICur; var
  hCursorDet: hDBICur; var hCursorSib: hDBICur): DBIResult stdcall;
```

Description

DbiGetLinkStatus returns the master, detail, and sibling cursors, if any, of the specified linked cursor.

Parameters

hCursor Type: hDBICur (Input)
Specifies the cursor handle.

phCursorMstr Type: phDBICur (Output)
Pointer to the master cursor, if any.

phCursorDet Type: phDBICur (Output)
Pointer to the first detail cursor, if any.

phCursorSib Type: phDBICur (Output)
Pointer to the next sibling detail cursor.

Usage

Used to find all links for the given cursor. If the cursor has a master, the master is returned. If the cursor has one or more details, the first detail is returned. If the cursor has siblings, the next sibling is returned. The master, detail, and sibling cursor handle can be used as an input to this function. If handle is not applicable, NULL is returned.

Prerequisites

The cursor must be a linked cursor. A linked cursor is created with DbiBeginLinkMode, DbiLinkDetail, or DbiLinkDetailToExp.

DbiResult return values

DBIERR_NONE	The linked cursor status was returned successfully.
DBIERR_INVALIDHNDL	The specified handle is invalid, not a linked cursor, or NULL.

See also

DbiBeginLinkMode, DbiLinkDetail, DbiLinkDetailToExp

C Examples: DbiGetLinkStatus

An example for this function is under development and will be provided in an upcoming Help release.

Delphi Examples: DbiGetLinkStatus

An example for this function is under development and will be provided in an upcoming Help release.

DbiGetNetUserName{button C Examples,JI(`>example',`exdbigetnetusername')} {button Delphi Examples,JI(`>example',`dexdbigetnetusername')}

C syntax

```
DBIResult DBIFN DbiGetNetUserName (pszNetUserName);
```

Delphi syntax

```
function DbiGetNetUserName (pszNetUserName: PChar): DBIResult stdcall;
```

Description

DbiGetNetUserName returns the user's network login name. User names are available for all networks supported by Microsoft Windows.

Parameters

pszNetUserName Type: pCHAR (Output)
Pointer to the client variable that receives the user network login name string.

DbiResult return values

DBIERR_NONE The user network login name was successfully retrieved.

DBIERR_INVALIDHNDL *pszNetUserName* is NULL.

C Examples: DbiGetNetUserName

Get the network user name.

If there is no network, this function returns NONE as the user. This example uses the following input:

fDbiGetNetUserName(UserName);

```
DBIResult fDbiGetNetUserName(pCHAR NetName)
{
   DBIResult      rslt;
   rslt = Chk(DbiGetNetUserName(NetName));
   if (rslt == DBIERR_INVALIDHNDL)
   {
      strcpy(NetName, "NONE");
      rslt = DBIERR_NONE;
   }
   else
      Chk(rslt);
   return rslt;
}
```

Delphi Examples: DbiGetNetUserName

Return the user's network login name.
User names are available for all networks supported by Microsoft Windows.

This example uses the following input:

```
  MyName := GetMyNetUserName;
```

The function is:

```
function GetMyNetUserName: string;
begin
   SetLength(Result, dbiMaxUserNameLen + 1);
   Check(DbiGetNetUserName(PChar(Result)));
   SetLength(Result, StrLen(PChar(Result)));
end;
```

DbiGetNextRecord {button C Examples,JI(`>example',`exdbigetnextrecord')} {button Delphi Examples,JI(`>example',`dexdbigetnextrecord')}

C syntax

```
DBIResult DBIFN DbiGetNextRecord (hCursor, [eLock], [pRecBuf], [precProps]);
```

Delphi syntax

```
function DbiGetNextRecord (hCursor: hDBICur; eLock: DBILockType; pRecBuff:
  Pointer; precProps: pRECProps): DBIResult stdcall;
```

Description

DbiGetNextRecord retrieves the next record in the table associated with *hCursor*.

Parameters

hCursor Type: hDBICur (Input)
Specifies the cursor handle.

eLock Type: DBILockType (Input)
Specifies the lock request type. Optional.

pRecBuf Type: pBYTE (Output)
Pointer to the client buffer that receives the record data. Optional. If NULL, no data is returned.

precProps Type: pRECProps (Output)
Pointer to the client-allocated RECProps structure. For dBASE, FoxPro, and Paradox drivers only. Optional. If NULL, no record properties are returned.

Usage

If a record buffer is provided, DbiGetNextRecord reads the data for the record into the record buffer. If the *precProps* argument is supplied, record properties are returned (dBASE, FoxPro, Access, and Paradox only). If filters are active, the next record that meets the filter criteria is retrieved. The record can be locked if an explicit lock is specified (using *eLock*), and the function call fails if the requested lock cannot be acquired. (Exceptions: see the discussion of SQL-specific locking behavior that follows.)

Field data can be retrieved using DbiGetField or DbiOpenBlob or DbiGetBlob for BLOB fields.

dBASE or FoxPro: If the *precProps* argument is supplied, the record number can be retrieved for the record (via the iPhyRecNum field of *precProps*). dBASE and FoxPro do not support the concept of sequence number.

Paradox: If the *precProps* argument is supplied, the sequence number can be retrieved for the record (via the *iSeqNum* field of *RECProps*). Paradox does not support the concept of record number.

SQL: Record properties are not supported for SQL drivers. If *precProps* is supplied, no properties are returned. For more information on locking, see Locking Strategy

Completion state

If the cursor is at the beginning of a table (after a opening a table or calling DbiSetToBegin), DbiGetNextRecord positions the cursor on the first record of the table. If the cursor is currently positioned on the last record in the table, DbiGetNextRecord returns an EOF error.

DbiResult return values

DBIERR_NONE	The next record was successfully retrieved.
DBIERR_EOF	The cursor was positioned at the crack at the end of the file or on the last record. It is now positioned at the crack at the end of the file.

DBIERR_INVALIDHNDL	The specified cursor handle is invalid or NULL.
DBIERR_ALREADYLOCKED	The record is already locked by the same user in the same session.
DBIERR_FILELOCKED	The table is already locked by another user (Paradox, FoxPro, Access, and dBASE only).

See also

DbiGetRecord, DbiGetPriorRecord, DbiGetRelativeRecord

eLock
eLock can be one of the following values:

Value	Description
dbiNOLOCK	No lock
dbiREADLOCK	Read lock
dbiWRITELOCK	Write lock

C Examples: DbiGetNextRecord

Retrieve the next record for the specified cursor.

In the case of local tables only, if *pRecNum* is not null, the corresponding record number is returned. This example uses the following input:

fDbiGetNextRecord(hCursor, pRecBuf, &RecNum);

```
DBIResult fDbiGetNextRecord(hDBICur hTmpCur, pBYTE pTmpRecBuf, pUINT32
  pRecNum)
{
   DBIResult      rslt;
   CURProps       CurProps;
   RECProps       RecProps;
   rslt = Chk(DbiGetNextRecord(hTmpCur, dbiNOLOCK, pTmpRecBuf, &RecProps));
   if (rslt != DBIERR_NONE)
      return rslt;
   if (pRecNum != NULL)
   {
      rslt = Chk(DbiGetCursorProps(hTmpCur, &CurProps));
      if (rslt != DBIERR_NONE)
         return rslt;
      if (strcmp(CurProps.szTableType,  szPARADOX) == 0)
         *pRecNum = RecProps.iSeqNum;
      else
      {
         if (strcmp(CurProps.szTableType,  szDBASE) ==0)
            *pRecNum = RecProps.iPhyRecNum;
         else
            *pRecNum =0;
      }
   }
   return rslt;
}
```

Delphi Examples: DbiGetNextRecord

Retrieve the next record in the table associated with the cursor:

Use Delphi's TTable methods to get records from a table (Next, Prior, First, Last, MoveBy, and so on.) For information about retrieving record numbers from a Paradox, FoxPro, Access, or dBASE table, see DbiGetRecord.

DbiGetNumberFormat {button C Examples,JI(`>example',`exdbigetnumberformat')} {button Delphi Examples,JI(`>example',`dexdbigetnumberformat')}

C syntax
```
DBIResult DBIFN DbiGetNumberFormat (pfmtNumber);
```

Delphi syntax
```
function DbiGetNumberFormat (var fmtNumber: FMTNumber): DBIResult stdcall;
```

Description
DbiGetNumberFormat returns the number format for the current session.

Parameters
pfmtNumber Type: pFMTNumber (Output)
Pointer to the client-allocated FMTNumber structure.

Usage
The number format is used by QBE for input and wildcard character matching. It is also used by batch operations (such as DbiDoRestructure and DbiBatchMove) to handle data type coercion between character and numeric types.

DbiResult return values
DBIERR_NONE The number format was successfully retrieved.

DBIERR_INVALIDHNDL *pfmtNumber* is NULL.

See also
DbiGetDateFormat, DbiGetTimeFormat, DbiSetNumberFormat

C Examples: DbiGetNumberFormat

An example for this function is under development and will be provided in an upcoming Help release.

Delphi Examples: DbiGetNumberFormat

Return the number format for the current session.

The number format is appended to the TStringList passed in. This example uses the following

```
  input:fDbiGetNumberFormat(MyNumberFormat);
```

The procedure is:

```
procedure fDbiGetNumberFormat(var NumberFormat: TStringList);
var
  FormatNumber: fmtNumber;
begin
  Check(DbiGetNumberFormat(FormatNumber));
  with NumberFormat do begin
    Add('Decimal Separator: ' + FormatNumber.cDecimalSeparator);
    Add('Thousand Separator: ' + FormatNumber.cThousandSeparator);
    Add('Decimal Digits: ' + IntToStr(FormatNumber.iDecimalDigits));
    if (fmtNumber.bLeadingZero) then
      Add('Leading Zero: True')
    else
      Add('Leading Zero: False');
  end;
end;
```

DbiGetObjFromName {button C Examples,JI(`>example',`exdbigetobjfromname')} {button Delphi Examples,JI(`>example',`dexdbigetobjfromname')}

C syntax

```
DBIResult DBIFN DbiGetObjFromName (eObjType, [pszObjName], phObj);
```

Delphi syntax

```
function DbiGetObjFromName (eObjType: DBIOBJType; pszObjName: PChar; var
  hObj: hDBIObj): DBIResult stdcall;
```

Description

DbiGetObjFromName returns an object handle of the specified type or with the given name, if any.

Parameters

eObjType Type: DBIOBJType (Input)
Specifies the type of object.

pszObjName Type: pCHAR (Input)
Pointer to the name of the object. Optional.

phObj Type: phDBIObj (Output)
Pointer to the object handle.

Usage

Some handles can be retrieved only by name, such as handles associated with cursors. For those, *pszObjName* is not optional. There can be more than one cursor open for a given table name; DbiGetObjFromName returns the handle to one of those cursors. To get a session handle, the session name need not be specified; by default, a handle to the currently active session is returned.

DbiResult return values

DBIERR_NONE	The object handle was returned successfully.
DBIERR_NOTSUPPORTED	Object is not supported for this function.
DBIERR_OBJNOTFOUND	Named object was not found.

pszObjName

The following chart lists the supported object types and whether or not the object name is required:

eObjType	**Name**
objSYSTEM	not needed
objSESSION	optional
objDRIVER	required
objDATABASE	optional
objCURSOR	required
objCLIENT	not needed

C Examples: DbiGetObjFromName

An example for this function is under development and will be provided in an upcoming Help release.

Delphi Examples: DbiGetObjFromName

An example for this function is under development and will be provided in an upcoming Help release.

DbiGetObjFromObj {button C Examples,JI(`>example',`exdbigetobjfromobj')} {button Delphi Examples,JI(`>example',`dexdbigetobjfromobj')}

C syntax

```
DBIResult DBIFN DbiGetObjFromObj (hObj, eObjType, phObj);
```

Delphi syntax

```
function DbiGetObjFromObj (hObj: hDBIObj; eObjType: DBIOBJType; var hObj:
  hDBIObj): DBIResult stdcall;
```

Description

DbiGetObjFromObj returns an object of the specified object type associated with or derived from a given object.

Parameters

hObj Type: hDBIObj (Input)
Specifies the object.

eObjType Type: DBIOBJType (Input)
Specifies the type of object.

phObj Type: phDBIObj (Output)
Pointer to the object handle.

Usage

The following table summarizes the relationship between *eObjType* and *hObj*:

eObjType	**Type of *hObj* allowed**
objCURSOR	None
objDRIVER	objCURSOR, objDATABASE
objDATABASE	objCURSOR
objSESSION	objCURSOR, objDATABASE, NULL (active)
objCLIENT	Any or NULL
objSYSTEM	Any or NULL
objSTATEMENT	None

DbiResult return values

DBIERR_NONE	The object handle was returned successfully.
DBIERR_INVALIDPARAM	*phObj* is NULL or *hObj* is invalid.
DBIERR_NA	No associated object.

C Examples: DbiGetObjFromObj

An example for this function is under development and will be provided in an upcoming Help release.

Delphi Examples: DbiGetObjFromObj

Show the driver name associated with the given parameters.
Delphi users will rarely need to call this function because most of this information is available through methods and properties of the TTable object.

```
//  Arguments:
//     hTmpDb:        Database handle
//     pszTableName:  Name of an existing table in the specified database
//     fDbiGetObjFromObj(hTmpDb, 'Employee.DB');
procedure fDbiGetObjFromObj(hTmpDb: hDBIDb; TblName: string);
var
  hCursor: hDBICur;
  szName: array[0..DBIMAXPATHLEN] of char;
  nLen: Word;
  hObj: hDBIObj;
  rslt: DBIResult;
begin
  // Open the specified table
  Check(DbiOpenTable(hTmpDb, PChar(TblName), nil, nil, nil, 0,
    dbiREADONLY, dbiOPENSHARED, xltFIELD, True, nil, hCursor));
  // Retrieve driver handle given cursor handle
  Check(DbiGetObjFromObj(hDBIObj(hCursor), DBIOBJType(objDRIVER), hObj));
  // Display driver name associated with the object handle
  rslt := DbiGetProp(hObj, drvDRIVERTYPE, @szName, sizeof(DBIPATH),nLen);
  if (rslt <> DBIERR_NONE) then
     Check(DbiCloseCursor(hCursor))
  else
    ShowMessage('Drive type: '+szName);
  // Close table
  Check(DbiCloseCursor(hCursor));
end;
```

Return the driver name for the specified table
Return the driver name for the specified table. This example uses the following input:

```
  DriverStr := fDbiGetObjFromObj(hTmpDb, CustomerTbl);
```

The function is:

```
function fDbiGetObjFromObj(Table: TTable): string;
var
  nLen: Word;
  hObj: hDBIObj;
begin
  // Retrieve driver handle given cursor handle
  Check(DbiGetObjFromObj(hDBIObj(Table.Handle), objDRIVER, hObj));
  SetLength(Result, DBIMAXDRIVELEN);
  // Get driver name associated with the driver handle
  Check(DbiGetProp(hObj, drvDRIVERTYPE, PChar(Result), DBIMAXDRIVELEN,
  nLen));
  SetLength(Result, StrLen(PChar(Result)));
end;
```

DbiGetPriorRecord {button C Examples,JI(`>example',`exdbigetpriorrecord')} {button Delphi Examples,JI(`>example',`dexdbigetpriorrecord')}

C syntax

```
DBIResult DBIFN DbiGetPriorRecord (hCursor, [eLock], [pRecBuf],
  [precProps]);
```

Delphi syntax

```
function DbiGetPriorRecord (hCursor: hDBICur; eLock: DBILockType; pRecBuff:
  Pointer; precProps: pRECProps): DBIResult stdcall;
```

Description

DbiGetPriorRecord retrieves the previous record in the table associated with the given cursor.

Parameters

hCursor Type: hDBICur (Input)
Specifies the cursor handle.

eLock Type: DBILockType (Input)
Specifies the lock request type Optional.

pRecBuf Type: pBYTE (Output)
Pointer to the client buffer that receives the record data. Optional. If NULL, no data is returned.

precProps Type: pRECProps (Output)
Pointer to the client-allocated RECProps structure. For dBASE, FoxPro, and Paradox drivers only. Optional. If NULL, no record properties are returned.

Usage

If a record buffer is provided, DbiGetPriorRecord reads the data for the record into the record buffer. If the *precProps* argument is supplied, record properties are returned (for dBASE, FoxPro, Access, and Paradox only). If filters are active, only records that meet the filter's criteria are retrieved. The record can be locked if an explicit lock is specified (using *eLock*), and the function call fails if the requested lock cannot be acquired. (Exceptions: see the discussion of SQL-specific locking behavior that follows.)

dBASE and FoxPro: If the *precProps* argument is supplied, the record number can be retrieved for the prior record (the *iPhyRecNum* field of the RECProps structure). dBASE and FoxPro do not support the concept of sequence numbers.

Paradox: If the *precProps* argument is supplied, the sequence number can be retrieved for the prior record (via the iSeqNum field of *precProps*). Paradox does not support the concept of record numbers.

SQL: Record properties are not supported for SQL drivers (*precProps* is NULL). If *precProps* is supplied, no properties are returned.

Prerequisites

A valid cursor handle must be obtained. If a lock is requested, the call returns DBIERR_NONE only if the lock is granted. For SQL, an error is returned if the cursor is not bidirectional.

Completion state

If the cursor is currently positioned on the first record in the table and the user calls DbiGetPriorRecord, then a BOF error is returned.

DbiResult return values

DBIERR_NONE	The prior record was retrieved successfully.
DBIERR_BOF	The cursor was positioned in the crack before the beginning of the file or on the first record after the crack. The cursor is now positioned in the crack at the beginning of the file.
DBIERR_INVALIDHNDL	The specified cursor handle is invalid or NULL.
DBIERR_ALREADYLOCKED	The record is already locked by the same user in the same session.
DBIERR_FILELOCKED	The table is already locked by another user (Paradox, FoxPro, Access, and dBASE only).
DBIERR_NA	Cursor is unidirectional.

See also

DbiGetRecord, DbiGetNextRecord, DbiGetPriorRecord, DbiGetRelativeRecord, DbiGetField, DbiModifyRecord

C Examples: DbiGetPriorRecord

Retrieve the prior record for the specified cursor.

For local tables only, if *pRecNum* is not null, the corresponding record number is returned. This example uses the following input:

fDbiGetPriorRecord(hCursor, pRecBuf, &RecNum);

```
DBIResult fDbiGetPriorRecord(hDBICur hTmpCur, pBYTE pTmpRecBuf, pUINT32
  pRecNum)
{
   DBIResult      rslt;
   CURProps       CurProps;
   RECProps       RecProps;
   rslt = Chk(DbiGetPriorRecord(hTmpCur, dbiNOLOCK, pTmpRecBuf, &RecProps));
   if (rslt != DBIERR_NONE)
      return rslt;
   if (pRecNum != NULL)
   {
      rslt = Chk(DbiGetCursorProps(hTmpCur, &CurProps));
      if (rslt != DBIERR_NONE)
         return rslt;
      if (strcmp(CurProps.szTableType,  szPARADOX) == 0)
         *pRecNum = RecProps.iSeqNum;
      else
      {
         if (strcmp(CurProps.szTableType,  szDBASE) ==0)
            *pRecNum = RecProps.iPhyRecNum;
         else
            *pRecNum =0;
      }
   }
   return rslt;
}
```

Delphi Examples: DbiGetPriorRecord

Retrieve the previous record in the table associated with the cursor:

Use Delphi's TTable methods to get records from a table (Next, Prior, First, Last, MoveBy, and so on.) For information about retrieving record numbers from a Paradox, FoxPro, Access, or dBASE table, see DbiGetRecord.

DbiGetProp {button C Examples,JI(`>example',`exdbigetprop')} {button Delphi Examples,JI(`>example',`dexdbigetprop')}

C syntax

```
DBIResult DBIFN DbiGetProp (hObj, iProp, pPropValue, iMaxLen, piLen);
```

Delphi syntax

```
function DbiGetProp (hObj: hDBIObj; iProp: Longint; PropValue: Pointer;
  iMaxLen: Word; var iLen: Word): DBIResult stdcall;
```

Description

DbiGetProp retrieves the properties of an object. See Getting and Setting Properties

Parameters

hObj Type: hDBIObj (Input)
Specifies the system, session, client, driver, database, cursor, or statement object.

iProp Type: UINT32 (Input)
Specifies the property to retrieve.

pPropValue Type: pVOID (Output)
Pointer to the client variable that receives the value of the property. Optional. If NULL, validates *iProp* for retrieval.

iMaxLen Type: UINT16 (Input)
Specifies the length of the *pPropValue* buffer.

p*iLen* Type: pUINT16 (Output)
Pointer to the client variable that receives the buffer length.

Usage

The specified object does not necessarily have to match the type of property as long as the object is associated with the object type of the property. For example, the property drvDRIVERTYPE assumes an object of type objDRIVER, but because a cursor is derived from a driver, a cursor handle (objCURSOR) could also be specified. See DbiGetObjFromObj for details about associated objects.

You can access the native connection, statement, and cursor handles by using DbiGetProp with the properties: dbNATIVEHNDL, dbNATIVEPASSTHRUHNDL, stmtNATIVEHNDL, and curNATIVEHNDL. This feature for retrieving native handles is useful for making direct native API calls when the necessary functionality is not available through BDE or in order to improve performance.

To inquire whether a driver supports stored procedures, use the property dbPROCEDURES.

To retrieve the server's default transaction isolation level use the property dbDEFAULTTXNISO.

Example

```
DBIPATH filename;
result=DbiGetProp (hCursor, curFILENAME, &filename, sizeof (DBIPATH),
  &length);
```

returns the file name associated with the cursor handle *hCursor* in filename and its length in length.

DbiResult return values

DBIERR_NONE	The properties were retrieved successfully.
DBIERR_BUFFTOOSMALL	Required buffer length is bigger than iMaxLen.
DBIERR_NOTSUPPORTED	Property is not supported for this object.

See also
DbiSetProp, DbiGetCursorProps, DbiGetObjFromObj

C Examples: DbiGetProp

Example 1: Get the native database handle from a remote database:

This example uses the following input:

```
fDbiGetProp1(hDb, &hIBDb);
```

```
DBIResult fDbiGetProp1(hDBIDb hTmpDb, pUINT32 hRemoteDb)
{
   DBIResult   rslt;

   rslt = Chk(DbiGetProp((hDBIObj)hTmpDb, dbNATIVEHNDL, (pBYTE)hRemoteDb,
               sizeof(dbNATIVEHNDL), NULL));
   return rslt;
}
```

Example 2: Return a string containing information about the specified table:

Note: *pCursorInfo* must be large enough to hold data. This example uses the following input:

```
char    Buffer[500];
fDbiGetProp2(hCur, Buffer);
```

```
DBIResult fDbiGetProp2(hDBICur hTmpCur, pCHAR pCursorInfo)
{
   DBIResult   rslt;
   CHAR        Buffer[500];
   INT16       Level;

   strcpy(pCursorInfo, "\0");
   // Get the table name.
   rslt = Chk(DbiGetProp((hDBIObj)hTmpCur, curTABLENAME, (pBYTE)Buffer,
                     DBIMAXTBLNAMELEN, NULL));
   if (rslt != DBIERR_NONE)
      return rslt;
   wsprintf(pCursorInfo, "%s\r\nTable Name: %s", pCursorInfo, Buffer);

   // Get the table type.
   rslt = Chk(DbiGetProp((hDBIObj)hTmpCur, curTABLETYPE, (pBYTE)Buffer,
                     DBIMAXNAMELEN, NULL));
   if (rslt != DBIERR_NONE)
      return rslt;
   wsprintf(pCursorInfo, "%s\r\nTable Type: %s", pCursorInfo, Buffer);

   // Get the full file name.
   rslt = Chk(DbiGetProp((hDBIObj)hTmpCur, curFILENAME, (pBYTE)Buffer,
                     DBIMAXPATHLEN, NULL));
   if (rslt != DBIERR_NONE)
      return rslt;
   wsprintf(pCursorInfo, "%s\r\nFile Name: %s", pCursorInfo, Buffer);

   // Get the table level.
   rslt = Chk(DbiGetProp((hDBIObj)hTmpCur, curTABLELEVEL, (pBYTE)&Level,
                     sizeof(curTABLELEVEL), NULL));
   if (rslt != DBIERR_NONE)
      return rslt;
   wsprintf(pCursorInfo, "%s\r\nTable Level: %d", pCursorInfo, Level);

   return rslt;
```

```
}
```

Example 3: Return a string containing information about the specified SQL database:

Note: *pCursorInfo* must be large enough to hold data. This example uses the following input:

```
char    Buffer[500];
fDbiGetProp3(hCur, Buffer);
```

```
DBIResult fDbiGetProp3(hDBIDb hTmpDb, pCHAR pDBInfo)
{
   DBIResult   rslt;
   BOOL        b;
   UINT16      i;

   strcpy(pDBInfo, "\0");

   // Does the Database support Asyncronous Query Execution support?
   rslt = Chk(DbiGetProp((hDBIObj)hTmpDb, dbASYNCSUPPORT, (pBYTE)&b,
                     sizeof(b), NULL));
   if (rslt != DBIERR_NONE)
      return rslt;
   if (b == TRUE)
      wsprintf(pDBInfo, "%s\r\nAsync. query exec support: TRUE", pDBInfo);
   else
      wsprintf(pDBInfo, "%s\r\nAsync. query exec support: FALSE", pDBInfo);

   // Does the Database support Stored Procedures?
   rslt = Chk(DbiGetProp((hDBIObj)hTmpDb, dbPROCEDURES, (pBYTE)&b,
                     sizeof(b), NULL));
   if (rslt != DBIERR_NONE)
      return rslt;
   if (b == TRUE)
      wsprintf(pDBInfo, "%s\r\nStored Procedure support: TRUE", pDBInfo);
   else
      wsprintf(pDBInfo, "%s\r\nStored Procedure support: FALSE", pDBInfo);

   // What is the major server version?
   rslt = Chk(DbiGetProp((hDBIObj)hTmpDb, dbSERVERVERSION, (pBYTE)&i,
                     sizeof(i), NULL));
   if (rslt != DBIERR_NONE)
      return rslt;
   wsprintf(pDBInfo, "%s\r\nMajor server version: %d", pDBInfo, i);

   return rslt;
}
```

Delphi Examples: DbiGetProp

Example 1: Return the native database handle

This example uses the following input:

```
Size := GetNativeDBHandle(Database1.Handle, NativeDB);
```

Size is a variable of type word. *NativeDB* is a variable of type longint.

```
function GetNativeDBHandle(DBHandle: hDBIDb; var NativeHandle: longint):
  Word;
begin
  Result := 0;
  // Get the native handle to the database...
  Check(DbiGetProp(hDBIObj(DBHandle), dbNATIVEHNDL, @NativeHandle,
    sizeof(NativeHandle), Result));
end;
```

Native Handles

Native handles allow you to bypass BDE functions to use native SQL database APIs to create and manipulate tables. This approach can deliver substantial performance improvement.

For example, to get a native database handle from a SQL database, you would use this code (assuming you have already obtained a valid handle *hDb* to an existing SQL database):

```
--------------
UINT16          Size;
hDBIDb          hDb;
UINT32          hNativeDb

DbiGetProp(hDb, dbNATIVEHNDL, &hNativeDb, sizeof(hNativeDb), &Size);
```

A native handle to the SQL database is returned in *&hNativeDb* and the size in bytes in *&Size*. Now you can execute native API calls for the SQL database.

The following table shows the information that is available for each driver when using dbNATIVEHNDL, dbNATIVEPASSTHRUHNDL, stmtNATIVEHNDL, or curNATIVEHNDL with DbiGetProp.

dbNATIVEHNDL, dbNATIVEPASSTHRUHNDL

	***ppropValue**	***pilen**
InterBase	gds_db_handle	4
Sybase	DBPROCESS NEAR *	2
Oracle	LDA	64
Informix	DBIERR_NOTSUPPORTED	--
DB2	HDBC	sizeof(HDBC)
ODBC Socket	HDBC	4

stmtNATIVEHNDL, curNATIVEHNDL

	***ppropValue**	***pilen**
InterBase	gds_stmt_handle	4
Sybase	DBIERR_NOTSUPPORTED	--
Oracle	CDA	64
Informix	DBIERR_NOTSUPPORTED	--
DB2	HSTMT	sizeof(HSTMT)
ODBC Socket	HSTMT	4

When SQLPASSTHRU MODE is NOT SHARED, the native handles returned from DbiGetProp with dbNATIVEHNDL and dbNATIVEPASSTHRUHNDL will be different. Certain drivers (for example, Sybase) may open multiple connections for one call to DbiOpenDatabase Only the main native connection handle is available.

Although the native connection and statement handles are always available when there is an active connection or statement, the native cursor handle may not always be available. For example: When working with a "dead" (snapshot) cursor, SQL Links caches each record as it is fetched from the server cursor. When all the records have been fetched, the server cursor is closed and it is no longer available. An attempt to retrieve the native cursor handle by using DbiGetProp with curNATIVEHNDL will return the error, DBIERR_OBJNOTFOUND.

Additional information on the native handle and its use is available from the SQL server vendors.

DbiGetRecord {button C Examples,JI(`>example',`exdbigetrecord')} {button Delphi Examples,JI(`>example',`dexdbigetrecord')}

C syntax

```
DBIResult DBIFN DbiGetRecord (hCursor, [eLock], [pRecBuf], [precProps]);
```

Delphi syntax

```
function DbiGetRecord (hCursor: hDBICur; eLock: DBILockType; pRecBuff:
  Pointer; precProps: pRECProps): DBIResult stdcall;
```

Description

DbiGetRecord retrieves the current record, if any, in the table associated with *hCursor*.

Parameters

hCursor Type: hDBICur (Input)
Specifies the cursor handle.

eLock Type: DBILockType (Input)
Specifies the lock request type Optional.

pRecBuf Type: pBYTE (Output)
Pointer to the client buffer that receives the record data. Optional. If NULL, no data is returned.

precProps Type: pRECProps (Output)
Pointer to the client-allocated RECProps structure. For Paradox, FoxPro, and dBASE drivers only. Optional. If NULL, no record properties are returned.

Usage

If NULL pointers are supplied for *pRecBuf* and *pRecProps*, DbiGetRecord can be used to validate the current cursor position (on a current record, or on a crack).

If filters are active, the record is retrieved only if it meets the filter's criteria. The record can be locked if an explicit lock is specified (using *eLock*), and the function call fails if the requested lock cannot be acquired. (Exceptions: see the discussion of SQL-specific locking behavior that follows. Also see Locking.)

If the cursor is currently positioned on a record, and that record is subsequently deleted or the record's key value is changed, then the cursor is left on a crack between records. At this point, a call to DbiGetRecord returns the DBIERR_KEYORRECDELETED error.

dBASE and FoxPro: If *precProps* is supplied, the record number can be retrieved for the current record (via the iPhyRecNum field of *precProps*). dBASE and FoxPro do not support the concept of sequence numbers.

Paradox: If *precProps* is supplied, the sequence number can be retrieved for the current record (via the iSeqNum field of *precProps*). Paradox does not support the concept of record numbers.

SQL: Record properties are not supported for SQL drivers (*precProps* is NULL). If *precProps* is supplied, no properties are returned.

Note: DbiGetRecordCount is not supported for cursors that reference query results or reference remote tables without a unique index; an error of DBIERR_NOTABLESUPPORT is returned.

DbiResult return values

DBIERR_NONE The record was successfully retrieved.

DBIERR_BOF At beginning of file.

DBIERR_EOF	At end of file.
DBIERR_NOCURRREC	No current record.
DBIERR_KEYORRECDELETED	The cursor is positioned on a record that has been deleted, or the key value was changed.
DBIERR_INVALIDHNDL	The specified cursor handle is invalid or NULL.
DBIERR_ALREADYLOCKED	The record is already locked by the same user in the same session.
DBIERR_LOCKED	The table is already locked by another user (Paradox, FoxPro, Access, and dBASE only).
DBIERR_NOTABLESUPPORT	The cursor is referencing a query result or a remote table without a unique index.

See also

DbiGetField, DbiGetNextRecord, DbiGetPriorRecord, DbiGetRelativeRecord

C Examples: DbiGetRecord

Retrieve the current record for the specified cursor.

For local tables only, if *pRecNum* is not null, the corresponding record number is returned. This example uses the following input:

fDbiGetRecord(hCursor, pRecBuf, &RecNum);

```
DBIResult fDbiGetRecord(hDBICur hTmpCur, pBYTE pTmpRecBuf, pUINT32 pRecNum)
{
   DBIResult      rslt;
   CURProps       CurProps;
   RECProps       RecProps;
   rslt = Chk(DbiGetRecord(hTmpCur, dbiNOLOCK, pTmpRecBuf, &RecProps));
   if (rslt != DBIERR_NONE)
      return rslt;
   if (pRecNum != NULL)
   {
      rslt = Chk(DbiGetCursorProps(hTmpCur, &CurProps));
      if (rslt != DBIERR_NONE)
         return rslt;
      if (strcmp(CurProps.szTableType,  szPARADOX) == 0)
         *pRecNum = RecProps.iSeqNum;
      else
      {
         if (strcmp(CurProps.szTableType,  szDBASE) ==0)
            *pRecNum = RecProps.iPhyRecNum;
         else
            *pRecNum =0;
      }
   }
   return rslt;
}
```

Delphi Examples: DbiGetRecord

Get the record ID of the current record in the specified TTable.
This example uses the following input:

```
fDbiGetRecord(Table1, Num);
```

The procedure is:

```
procedure fDbiGetRecord(ATable: TTable; var RecID: Longint);
var
  CP: CurProps;
  RP: RecProps;
begin
  with ATable do begin
    // Make sure it is a Paradox table!
    UpdateCursorPos;                // sync BDE with Delphi
    // Find out if table support Seq nums or Physical Rec nums
    Check(DbiGetCursorProps(Handle, CP));
    Check(DbiGetRecord(Handle, dbiNOLOCK, nil, @RP));
    if (StrComp(CP.szTableType, szDBASE) = 0) then
      RecID := RP.iPhyRecNum
    else
      if (StrComp(CP.szTableType, szPARADOX) = 0) then
        RecID := RP.iSeqNum
      else
        // raise exception if it's not a Paradox or dBASE table
        raise EDatabaseError.Create('Not a Paradox or dBASE table');
  end;
end;
```

DbiGetRecordCount {button C Examples,JI(`>example',`exdbigetrecordcount')} {button Delphi Examples,JI(`>example',`dexdbigetrecordcount')}

C syntax

```
DBIResult DBIFN DbiGetRecordCount (hCursor, piRecCount);
```

Delphi syntax

```
function DbiGetRecordCount (hCursor: hDBICur; var iRecCount: Longint):
  DBIResult stdcall;
```

Description

DbiGetRecordCount is used to get the current number of records associated with the cursor.

Parameters

hCursor Type: hDBICur (Input)
Specifies the cursor handle.

piRecCount Type: pUINT32 (Output)
Pointer to the client variable which receives the number of records associated with the cursor. This number may be approximate.

Usage

This function is meant to get the number of records associated with the cursor. The count is approximate in some cases, rather than exact. (If there are any active filters associated with the cursor, or if there are any active ranges declared on it, the results are approximate; they are normally the upper limits.)

Paradox: If a range is active, the record count returned is the number of records in the range.

DbiResult return values

DBIERR_NONE	The record count was retrieved successfully.
DBIERR_INVALIDHNDL	The specified cursor handle is invalid or NULL.

See also

DbiGetExactRecordCount

C Examples: DbiGetRecordCount

Example 1: Get the record count for the specified table

This example uses the following input:
fDbiGetRecordCount1(hCursor, &RecCount);

```
DBIResult fDbiGetRecordCount1(hDBICur hTmpCur, pUINT32 piRecCount)
{
   DBIResult      rslt;
   rslt = Chk(DbiGetRecordCount(hTmpCur, piRecCount));
   return rslt;
}
```

Example 2 (for local tables only): Get the amount of records after the current location of the cursor.

This example uses the following input:
fDbiGetRecordCount2(hCursor, &RecLeft);

```
DBIResult fDbiGetRecordCount2(hDBICur hTmpCur, pUINT32 piRecLeft)
{
   DBIResult      rslt;
   UINT32         RecNum, CurrRec;
   CURProps       Props;
   RECProps       RecProps;
   rslt = Chk(DbiGetRecordCount(hTmpCur, &RecNum));
   if (rslt != DBIERR_NONE)
      return rslt;
   if (piRecLeft != NULL)
   {
      rslt = Chk(DbiGetCursorProps(hTmpCur, &Props));
      if (rslt != DBIERR_NONE)
         return rslt;
      if (strcmp(Props.szTableType, szPARADOX) == 0)
      {
         rslt = Chk(DbiGetSeqNo(hTmpCur, &CurrRec));
         if (rslt != DBIERR_NONE)
          return rslt;
      }
      else
      {
         if (strcmp(Props.szTableType, szDBASE) == 0)
         {
            rslt = Chk(DbiGetRecord(hTmpCur, dbiNOLOCK, NULL, &RecProps));
            if (rslt != DBIERR_NONE)
             return rslt;
            CurrRec = RecProps.iPhyRecNum;
         }
         else
           return DBIERR_NA;
      }
      *piRecLeft = RecNum - CurrRec;
   }
   return rslt;
}
```

Delphi Examples: DbiGetRecordCount

Return the record count of the TDataSet descendant (TTable, TQuery, TStoredProc) passed in parameter D.

This example uses the following input:

```
  ShowMessage(IntToStr(fDbiGetRecordCount(Table1)));
```

The function is:

```
function fDbiGetRecordCount(D: TDataSet): LongInt;
begin
  Check(DbiGetRecordCount(D.Handle, Result));
end;
```

DbiGetRecordForKey{button C Examples,JI(`>example',`exdbigetrecordforkey')} {button Delphi Examples,JI(`>example',`dexdbigetrecordforkey')}

C syntax

```
DBIResult DBIFN DbiGetRecordForKey (hCursor, bDirectKey, iFields, iLen,
  pKey, [pRecBuf]);
```

Delphi syntax

```
function DbiGetRecordForKey (hCursor: hDBICur; bDirectKey: Bool; iFields:
  Word; iLen: Word;  pKey: Pointer; pRecBuff: Pointer): DBIResult stdcall;
```

Description

DbiGetRecordForKey finds a record matching *pKey* and positions the cursor on that record.

Parameters

hCursor Type: hDBICur (Input)
Specifies the cursor handle.

bDirectKey Type: BOOL (Input)
Determines whether *pKey* is used to specify the key directly or not. If TRUE, the value in *pKey* is used to specify the key directly. If FALSE, *pKey* specifies the record buffer.

*iField*s Type: UINT16 (Input)
Specifies the number of fields to be used for composite keys. If *iField*s and *iLen* are both 0, the entire key is used.

iLen Type: UINT16 (Input)
Specifies the length into the last field to be used for composite keys. If not 0, the last field to be used must be a character type.

pKey Type: pBYTE (Output)
If *bDirectKey* is TRUE, the *pKey* specifies the pointer to the record key; otherwise, *pKey* specifies the pointer to the record buffer. DbiExtractKey can be used to construct the record key when *bDirectKey* is TRUE. The *iField*s and *iLen* Parameters together indicate how much of the key should be used for matching. If both are 0, the entire key is used. If a match is required on a given field of the key, all the key fields preceding it in the composite key must also be supplied for a match. Only character fields can be matched for a partial key; all other field types must be fully matched.

For partial key matches, *iField*s must be equal to the number of keyfields preceding the field being partially matched. *iLen* specifies the number of characters in the (*iFields*+1) field.

pRecBuf Type: pBYTE (Output)
Pointer to the record buffer where the new current record is returned.

Usage

SQL: For SQL tables, if the active index is not unique, DbiGetRecordForKey may return different records with the same key value.

Prerequisites

A valid cursor handle must be obtained.

Completion state

The cursor is positioned on the found record. If *pRecBuf* is supplied, the new current record is retrieved. If there is no key in the index that matches the given key, an error is returned.

DbiResult return values

DBIERR_NOCURRREC	The cursor is not positioned on a record.
DBIERR_RECNOTFOUND	No record with the specified key value was found.

See also

DbiSetToKey, DbiExtractKey

C Examples: DbiGetRecordForKey

Position the cursor and return the record buffer containing the specified key

This examples works with the STOCK.DB table and uses the following input:

fDbiGetRecordForKey(hCur, pRecBuf);

```
DBIResult fDbiGetRecordForKey(hDBICur hTmpCur, pBYTE pTmpRecBuf)
{
   DBIResult   rslt;
   DFLOAT      key = 1330.00;
   rslt = Chk(DbiInitRecord(hTmpCur, pTmpRecBuf));
   if (rslt != DBIERR_NONE)
      return rslt;
   rslt = Chk(DbiPutField(hTmpCur, 1, pTmpRecBuf, (pBYTE)&key));
   if (rslt != DBIERR_NONE)
      return rslt;
   rslt = Chk(DbiGetRecordForKey(hTmpCur, FALSE, 0, 0, pTmpRecBuf,
  pTmpRecBuf));
   return rslt;
}
```

Delphi Examples: DbiGetRecordForKey

Position the cursor and return the record buffer containing the specified key

Delphi users should use TDataSet.Locate, TTable.FindKey, or TTable.GotoKey rather than directly calling dbiGetRecordForKey. These methods are described in the Delphi online help.

DbiGetRelativeRecord {button C Examples,JI(`>example',`exdbigetrelativerecord')} {button Delphi Examples,JI(`>example',`dexdbigetrelativerecord')}

C syntax

```
DBIResult DBIFN DbiGetRelativeRecord (hCursor, iPosOffset, [eLock],
  [pRecBuf], [precProps]);
```

Delphi syntax

```
function DbiGetRelativeRecord (hCursor: hDBICur; iPosOffset: Longint; eLock:
  DBILockType; pRecBuff: Pointer; precProps: pRECProps): DBIResult stdcall;
```

Description

DbiGetRelativeRecord positions the cursor on a record in the table relative to the current position of the cursor.

Parameters

hCursor Type: hDBICur (Input)
Specifies the cursor handle.

iPosOffset Type: INT32 (Input)
Specifies the (signed) offset from current record.

eLock Type: DBILockType (Input)
Specifies the lock request type. Optional.

pRecBuf Type: pBYTE (Output)
Pointer to the client buffer that receives the record data. Optional. If NULL, no data is returned.

precProps Type: pRECProps (Output)
Pointer to the client-allocated RECProps structure.

Usage

This function positions the cursor relative to the current position. The record offset (*iPosOffset*) can be positive or negative. If the cursor is currently positioned between records, the next or prior (depending on the direction) record is counted as 1. If the filter is active, only those records that meet the filter condition are included. For dBASE or FoxPro if Soft Delete is off, only undeleted records are included.

If a record buffer is provided, DbiGetRelativeRecord reads the data for the record into the record buffer. If the *precProps* argument is supplied, record properties are returned (for Paradox, FoxPro, Access, and dBASE only). If filters are active, only records that meet the filter's criteria are retrieved. The record can be locked if an explicit lock is specified (using *eLock*), and the function call returns an error if the requested lock cannot be acquired. See the following section for SQL-specific locking behavior information.

dBASE and FoxPro: If the *precProps* argument is supplied, the record number can be retrieved for the record (the *iPhyRecNum* field of the RECProps structure). dBASE and FoxPro do not support the concept of sequence numbers.

Paradox: If the *precProps* argument is supplied, the sequence number can be retrieved for the record (via the iSeqNum field of *precProps*). Paradox does not support the concept of record numbers.

SQL: Record properties are not supported for SQL drivers (*precProps* is NULL). If *precProps* is supplied, no properties are returned.

Completion state

If not enough records exist in the result set to move to the relative record location, a

beginning of file/end of file (BOF/EOF) error is returned. An error is returned if the cursor is not bidirectional, and the cursor is moving backwards.

DbiResult return values

DBIERR_NONE	The record was retrieved successfully.
DBIERR_BOF	The beginning of the file was reached.
DBIERR_EOF	The end of the file was reached.
DBIERR_INVALIDHNDL	The specified cursor handle is invalid or NULL.
DBIERR_KEYORRECDELETED	The cursor is positioned in a crack other than BOF or EOF.
DBIERR_ALREADYLOCKED	The record is already locked by the same user in the same session.
DBIERR_FILELOCKED	The table is already locked by another user.

See also

DbiGetField, DbiGetNextRecord, DbiGetPriorRecord

C Examples: DbiGetRelativeRecord

Retrieve the relative record for the specified cursor.

For local tables only, if *pRecNum* is not null, the corresponding record number is returned. This example use the following input:

fDbiGetRelativeRecord(hPXCur, 10, pPXRecBuf, &FldNum);

```
DBIResult fDbiGetRelativeRecord(hDBICur hTmpCur, INT32 iOffset, pBYTE
  pTmpRecBuf, pUINT32 pRecNum)
{
   DBIResult      rslt;
   CURProps       CurProps;
   RECProps       RecProps;
   rslt = Chk(DbiGetRelativeRecord(hTmpCur, iOffset, dbiNOLOCK, pTmpRecBuf,
  &RecProps));
   if (rslt != DBIERR_NONE)
      return rslt;
   if (pRecNum != NULL)
   {
      rslt = Chk(DbiGetCursorProps(hTmpCur, &CurProps));
      if (rslt != DBIERR_NONE)
         return rslt;
      if (strcmp(CurProps.szTableType,  szPARADOX) == 0)
         *pRecNum = RecProps.iSeqNum;
      else
      {
         if (strcmp(CurProps.szTableType,  szDBASE) ==0)
            *pRecNum = RecProps.iPhyRecNum;
         else
            *pRecNum =0;
      }
   }
   return rslt;
}
```

Delphi Examples: DbiGetRelativeRecord

Retrieve the relative record for the specified cursor.

Use Delphi's TTable methods to get records from a table (Next, Prior, First, Last, MoveBy, and so on.) For information about retrieving record numbers from a Paradox, FoxPro, Access, or dBASE table, see DbiGetRecord.

DbiGetRintDesc {button C Examples,JI(`>example',`exdbigetrintdesc')} {button Delphi Examples,JI(`>example',`dexdbigetrintdesc')}

C syntax

```
DBIResult DBIFN DbiGetRintDesc (hCursor, iRintSeqNo, printDesc);
```

Delphi syntax

```
function DbiGetRintDesc (hCursor: hDBICur; iRintSeqNo: Word; printDesc:
  pRINTDesc): DBIResult stdcall;
```

Description

DbiGetRintDesc retrieves the referential integrity descriptor identified by the referential integrity sequence number and the cursor.

Parameters

hCursor Type: hDBICur (Input)
Specifies the cursor handle.

iRintSeqNo Type: UINT16 (Input)
The referential integrity sequence number. This number is between 1 and the value of iRefIntChecks. The value of iRefIntChecks can be obtained from the cursor properties (CURProps) structure.

printDesc Type: pRINTDesc (Output)
Pointer to the client variable that receives the referential integrity descriptor.

Usage

If a field map is associated with the cursor, the *aiThisTabFld* array in the referential integrity descriptor reflects the field map. If any of the fields are not part of the field-mapped record, a negative number is listed.

DbiResult return values

DBIERR_NONE The descriptor was returned successfully.

See also

DbiGetCursorProps, DbiOpenRintList

C Examples: DbiGetRintDesc

Get the first referential integrity constraint for the specified cursor.

If *RIDesc* is not null, a string is also formulated with RI information.This example uses the following input:

fDbiGetRintDesc(hCur, &RintDesc, Buffer);

```
DBIResult fDbiGetRintDesc(hDBICur hTmpCur, pRINTDesc pRintDesc, pCHAR RIDesc)
{
   DBIResult   rslt;
   CURProps    CurProps;
   rslt = Chk(DbiGetCursorProps(hTmpCur, &CurProps));
   if(CurProps.iRefIntChecks > 0)
   {
      rslt = Chk(DbiGetRintDesc(hTmpCur, 1, pRintDesc));
      if (rslt != DBIERR_NONE)
         return rslt;
      if (RIDesc != NULL)
        wsprintf(RIDesc, "RI Number: %i, IR Name: %s, RI Tbl Name: %s, RI
  Fields: %d",
                  pRintDesc->iRintNum, pRintDesc->szRintName, pRintDesc-
  >szTblName,
                  pRintDesc->iFldCount);
   }
   else
      MessageBox(0, "No Referential Integrity Links", "Warning", MB_OK);
   return rslt;
}
```

Delphi Examples: DbiGetRintDesc

Return and fill a TStringList with information on the referential integrity. This example uses the following input:

```
  fDbiGetRIntDesc(OrdersTbl, 1, MyList);
```

The function is:

```
function fDbiGetRIntDesc(Table: TTable; SeqNo: Word; RIntList: TStringList):
  RINTDesc;
var
  ThisTable, OtherTable: string;
  Props: CURProps;
  B: Byte;
begin
  ThisTable := '';
  OtherTable := '';
  FillChar(Result, sizeof(Result), #0);
  Check(DbiGetCursorProps(Table.Handle, Props));
  if (Props.iRefIntChecks = 0) then
    raise EDatabaseError.Create('There are no referential integrity checks
  on this table');
  Check(DbiGetRIntDesc(Table.Handle, SeqNo, @Result));
  if (RIntList <> nil) then begin
    with RIntList do begin
      Add(Format('NUMBER=%d', [Result.iRintNum]));
      Add(Format('NAME=%s', [Result.szRintName]));
      case Result.eType of
        rintMASTER: Add('TYPE=MASTER');
        rintDEPENDENT: Add('TYPE=DEPENDENT');
      else
        Add('TYPE=UNKNOWN');
      end;
      Add(Format('OTHER TABLE=%s', [Result.szTblName]));
      case Result.eModOp of
        rintRESTRICT: Add('MODIFY=RESTRICT');
        rintCASCADE: Add('MODIFY=CASCADE');
      else
        Add('MODIFY=UNKNOWN');
      end;
      case Result.eDelOp of
        rintRESTRICT: Add('DELETE=RESTRICT');
        rintCASCADE: Add('DELETE=CASCADE');
      else
        Add('DELETE=UNKNOWN');
      end;
      Add(Format('FIELD COUNT=%d', [Result.iFldCount]));
      for B := 0 to DBIMAXFLDSINKEY do begin
        if (Result.aiThisTabFld[B] <> 0) then begin
          if (B <> 0) then
            ThisTable := Format('%s, %d', [ThisTable,
  Result.aiThisTabFld[B]])
          else
            ThisTable := IntToStr(Result.aiThisTabFld[B]);
        end
        else
          Break;
```

```
      end;
      Add('FIELDS=' + ThisTable);
      for B := 0 to DBIMAXFLDSINKEY do begin
        if (Result.aiOthTabFld[B] <> 0) then begin
          if (B <> 0) then
            OtherTable := Format('%s, %d', [OtherTable,
  Result.aiOthTabFld[B]])
          else
            OtherTable := IntToStr(Result.aiOthTabFld[B]);
        end
        else
          Break;
      end;
      Add('FIELDS OTHER=' + OtherTable);
    end;
  end;
end;
```

DbiGetSeqNo {button C Examples,JI(`>example',`exdbigetrecordcount')} {button Delphi Examples,JI(`>example',`dexdbigetseqno')}

C syntax
```
DBIResult DBIFN DbiGetSeqNo (hCursor, piSeqNo);
```

Delphi syntax
```
function DbiGetSeqNo (hCursor: hDBICur; var iSeqNo: Longint): DBIResult
  stdcall;
```

Description
DbiGetSeqNo retrieves the sequence number of the current record in the table associated with the *cursor*.

Parameters
hCursor Type: hDBICur (Input)
Specifies the cursor handle.

piSeqNo Type: pUINT32 (Output)
Pointer to the client variable that receives the logical sequence number of the current record in the table associated with *hCursor*.

Usage
Paradox: This function is supported for all Paradox tables.

SQL: This function is not supported by SQL drivers.

dBASE: This function is not supported by the dBASE driver.

Access: This function is not supported by the Access driver.

Cached updates: This function is not supported with cached updates, since cached updates use an in-memory table.

Prerequisites
The cursor should be positioned on a record.

Completion state
The sequence number is the relative position of a record with respect to the beginning of the file. A sequence number for a given record therefore depends on the current index in use. An active range also affects the sequence numbers, the sequence number is relative to the beginning of the range. Filters do not affect sequence numbers, so there might seem to be gaps in the sequence numbers.

DbiResult return values
DBIERR_NOTSUPPORTED	This call is not supported for the given table.
DBIERR_NONE	The sequence number was returned successfully.
DBIERR_BOF	The cursor must be positioned on a record; it is positioned at the beginning of the file.
DBIERR_EOF	The cursor must be positioned on a record; it is positioned at the end of the file.
DBIERR_KEYORRECDELETED	The cursor is positioned on a deleted record.
DBIERR_NOCURRREC	No record is current.

See also
DbiSetToSeqNo, DbiGetCursorProps

Delphi Examples: DbiGetSeqNo

Retrieve the sequence number of the current record associated with a cursor:
The following procedure returns the record ID of the current record in ATable. If ATable is a Paradox table, it uses DbiGetSeqNo() to obtain the sequence number. If ATable is a dBASE or FoxPro table, it uses record properties provided by DbiGetRecord to obtain the physical record number. If the table is SQL or text, an exception is raised. The record ID is returned in the RecID parameter, which is passed by reference.

```
procedure GetRecordID(ATable: TTable; var RecID: Longint);
var
  CP: CURProps;
  RP: RECProps;
begin
  with ATable do begin
    { Make sure it is a Paradox table! }
    UpdateCursorPos;                // sync BDE with Delphi
    { Find out if table support Seq nums or Physical Rec nums }
    Check(dbiGetCursorProps(Handle, CP));
    case CP.iSeqNums of
      0 : begin           // dBASE tables support Phy Rec Nums
        Check(DbiGetRecord(Handle, dbiNOLOCK, nil, @RP));
        RecID := RP.iPhyRecNum;
      end;
      1 : Check(DbiGetSeqNo(Handle, RecID)); // Paradox tables support Seq
  Nums
     else
      { raise exception if it's not a Paradox or dBASE table }
      raise EDatabaseError.Create('Not a Paradox or dBASE table');
    end;
    CursorPosChanged;               // sync Delphi with BDE
  end;
end;
```

DbiGetSesInfo {button C Examples,JI(`>example',`exdbigetsesinfo')} {button Delphi Examples,JI(`>example',`dexdbigetsesinfo')}

C syntax
```
DBIResult DBIFN DbiGetSesInfo (psesInfo);
```

Delphi syntax
```
function DbiGetSesInfo (var sesInfo: SESInfo): DbiResult stdcall;
```

Description
DbiGetSesInfo retrieves the environment settings for the current session.

Parameters
psesInfo Type: pSESInfo (Output)
Pointer to the client-allocated SESInfo structure.

Usage
This function provides the client with information about the resources attached to the current session, including the number of database handles and open cursors (when the session is closed, these resources are released). This function also returns the session ID and name, the current private directory, and the lock retry time for repeated attempts to lock a table. The lock retry time is specified by DbiSetLockRetry.

Completion state
The session information is returned in the specified SESInfo structure.

DbiResult return values
DBIERR_NONE The session information was returned successfully.

DBIERR_INVALIDHNDL *psesInfo* is NULL.

See also
DbiSetLockRetry, DbiStartSession, DbiCloseSession, DbiGetCurrSession, DbiSetCurrSession

C Examples: DbiGetSesInfo

Get session information.

If *SesInfo* is not NULL, a string is also returned containing session information. This example uses the following input:

fDbiGetSesInfo(&Sesinfo, Buffer);

```
DBIResult fDbiGetSesInfo(pSESInfo pSesInfo, pCHAR SesInfo)
{
   DBIResult   rslt;
   rslt = Chk(DbiGetSesInfo(pSesInfo));
   if (rslt == DBIERR_NONE)
   {
      if (SesInfo != NULL)
         wsprintf(SesInfo, "ID: %d, Name: %s, Open DB: %d, Open Cursors: %d,
  "
            "Lock Wait: %d, Net Dir: %s, Private Dir: %s", pSesInfo-
  >iSession,
            pSesInfo->szName, pSesInfo->iDatabases, pSesInfo->iCursors,
            pSesInfo->iLockWait, pSesInfo->szNetDir, pSesInfo->szPrivDir);
   }
   return rslt;
}
```

Delphi Examples: DbiGetSesInfo

Get BDE session information. This function can return the SESInfo structure or clear and add the information to the SesInfoList TStringList. If nil is passed in, only the SESInfo structure is returned. This example uses the following input:

```
  Ses := fDbiGetSesInfo(MyList);
```

The function is:

```
function fDbiGetSesInfo(SesInfoList: TStringList): SESInfo;
begin
  Check(DbiGetSesInfo(Result));
  if (SesInfoList <> nil) then
  begin
    with SesInfoList do begin
      Clear;
      Add(Format('SESSION ID=%d', [Result.iSession]));
      Add(Format('SESSION NAME=%s', [Result.szName]));
      Add(Format('DATABASES=%d', [Result.iDatabases]));
      Add(Format('CURSORS=%d', [Result.iCursors]));
      Add(Format('LOCK WAIT=%d', [Result.iLockWait]));
      Add(Format('NET DIR=%s', [Result.szNetDir]));
      Add(Format('PRIVATE DIR=%s', [Result.szPrivDir]));
    end;
  end;
end;
```

DbiGetSysConfig {button C Examples,JI(`>example',`exdbigetsysconfig')} {button Delphi Examples,JI(`>example',`dexdbigetsysconfig')}

C syntax

```
DBIResult DBIFN DbiGetSysConfig (psysConfig);
```

Delphi syntax

```
function DbiGetSysConfig (var sysConfig: SYSConfig): DBIResult stdcall;
```

Description

DbiGetSysConfig retrieves BDE system configuration information.

Parameters

psysConfig Type: pSYSConfig (Output)
Pointer to the client-allocated SYSConfig structure.

Completion state

The SYSConfig structure pointed to by *psysConfig* contains the retrieved system configuration information.

DbiResult return values

DBIERR_NONE System configuration information was returned successfully.

See also

DbiGetSysVersion, DbiGetClientInfo, DbiGetSysInfo

C Examples: DbiGetSysConfig

Get system configuration information.

If *SysCfg* is not NULL, a string is also returned containing system configuration. This example uses the following input:

fDbiGetSysConfig(&SysConfig, Buffer);

```
DBIResult fDbiGetSysConfig(pSYSConfig pSysConfig, pCHAR SysCfg)
{
   DBIResult   rslt;
   CHAR        szLocal[] = {"False"};
   rslt = Chk(DbiGetSysConfig(pSysConfig));
   if (rslt == DBIERR_NONE)
   {
      if (SysCfg != NULL)
      {
         if (pSysConfig->bLocalShare == TRUE)
            strcpy(szLocal, "True");
         wsprintf(SysCfg, "Local Share: %s, Net Type: %s, User Name: %s, "
            ".CFG File: %s, Lang Driver: %s", szLocal, pSysConfig->szNetType,
            pSysConfig->szUserName, pSysConfig->szIniFile, pSysConfig-
  >szLangDriver);
      }
   }
   return rslt;
}
```

Delphi Examples: DbiGetSysConfig

Get system configuration information:

DbiGetSysConfig retrieves the BDE system configuration information and appends it to the TStringList passed in. This example uses the following input:

```
  fDbiGetSysConfig(MySysInfo);
```

The procedure is:

```
procedure fDbiGetSysConfig(var IdapiSysConfig: TStringList);
var
  SysConfigInfo: SYSConfig;
begin
  Check(DbiGetSysConfig(SysConfigInfo));
  if SysConfigInfo.bLocalShare then
    IdapiSysConfig.Add('Local Share: ON')
  else
    IdapiSysConfig.Add('Local Share: OFF');
  IdapiSysConfig.Add('Net Protocol: ' +
  IntToStr(SysConfigInfo.iNetProtocol));
  if SysConfigInfo.bNetShare then
    IdapiSysConfig.Add('Net Share: ON')
  else
    IdapiSysConfig.Add('Net Share: OFF');
  IdapiSysConfig.Add('Network Type: ' + StrPas(SysConfigInfo.szNetType));
  IdapiSysConfig.Add('User Name: ' + StrPas(SysConfigInfo.szUserName));
  IdapiSysConfig.Add('Ini File: ' + StrPas(SysConfigInfo.szIniFile));
  IdapiSysConfig.Add('Language Driver: ' +
  StrPas(SysConfigInfo.szLangDriver));
end;
```

DbiGetSysInfo {button C Examples,JI(`>example',`exdbigetsysinfo')} {button Delphi Examples,JI(`>example',`dexdbigetsysinfo')}

C syntax

```
DBIResult DBIFN DbiGetSysInfo (psysInfo);
```

Delphi syntax

```
function DbiGetSysInfo (var sysInfo: SYSInfo): DBIResult stdcall;
```

Description

DbiGetSysInfo retrieves system status and information.

Parameters

psysInfo Type: pSYSInfo (Output)
Pointer to the client-allocated SYSInfo structure.

Completion state

The SYSInfo structure pointed to by *psysInfo* contains the retrieved system status and information.

DbiResult return values

DBIERR_NONE System status information was returned successfully.

See also

DbiGetSysVersion, DbiGetSysConfig, DbiGetClientInfo

C Examples: DbiGetSysInfo

Get system information.

If SysCfg is not NULL, a string is also returned containing system information. This example uses the following input:

fDbiGetSysConfig(&SysInfo, Buffer);

```
DBIResult fDbiGetSysInfo(pSYSInfo pSysInfo, pCHAR SysInfo)
{
   DBIResult   rslt;
   rslt = Chk(DbiGetSysInfo(pSysInfo));
   if (rslt == DBIERR_NONE)
   {
      if (SysInfo != NULL)
        wsprintf(SysInfo, "Buffer Space: %d, Heap: %d, Drivers: %d, Clients:
  %d, "
           "Sessions: %d, Databases: %d, Cursors: %d", pSysInfo-
  >iBufferSpace,
           pSysInfo->iHeapSpace, pSysInfo->iDrivers, pSysInfo->iClients,
           pSysInfo->iSessions, pSysInfo->iDatabases, pSysInfo->iCursors);
   }
   return rslt;
}
```

Delphi Examples: DbiGetSysInfo

Get BDE system status information. This function can return the SYSInfo structure or clear and add the information to the SysInfoList TStringList. If nil is passed in, only the SYSInfo structure is returned. This example uses the following input:

```
  Sys := fDbiGetSysInfo(MyList);
```

The function is:

```
function fDbiGetSysInfo(SysInfoList: TStringList): SYSInfo;
begin
  Check(DbiGetSysInfo(Result));
  if (SysInfoList <> nil) then begin
    with SysInfoList do begin
      Clear;
      Add(Format('BUFFER SPACE=%d', [Result.iBufferSpace]));
      Add(Format('HEAP SPACE=%d',   [Result.iHeapSpace]));
      Add(Format('DRIVERS=%d',      [Result.iDrivers]));
      Add(Format('CLIENTS=%d',      [Result.iClients]));
      Add(Format('SESSIONS=%d',     [Result.iSessions]));
      Add(Format('DATABASES=%d',    [Result.iDatabases]));
      Add(Format('CURSORS=%d',      [Result.iCursors]));
    end;
  end;
end;
```

DbiGetSysVersion {button C Examples,JI(`>example',`exdbigetsysversion')}{button Delphi Examples,JI(`>example',`dexdbigetsysversion')}

C syntax

```
DBIResult DBIFN DbiGetSysVersion (psysVersion);
```

Delphi syntax

```
function DbiGetSysVersion (var sysVersion: SYSVersion): DBIResult stdcall;
```

Description

DbiGetSysVersion retrieves the system version information, including the BDE version number, date, and time; and the client interface version number.

Parameters

psysVersion Type: pSYSVersion (Output)
Pointer to the client-allocated SYSVersion structure.

Completion state

The SYSVersion structure returned in *psysVersion* contains the retrieved system version information.

DbiResult return values

DBIERR_NONE The system version information was returned successfully.

See also

DbiGetSysConfig, DbiGetClientInfo, DbiGetSysInfo

C Examples: DbiGetSysVersion

Get system version information:

If *SysVer* is not NULL, a string is also returned containing system version information. This example uses the following input:

fDbiGetSysVersion(&SysVersion, Buffer);

```
DBIResult fDbiGetSysVersion(pSYSVersion pSysVersion, pCHAR SysVer)
{
   DBIResult   rslt;
   UINT16      Mo, Da, H, M, Ms;
   INT16       Yr;
   rslt = Chk(DbiGetSysVersion(pSysVersion));
   if (rslt == DBIERR_NONE)
   {
      if (SysVer != NULL)
      {
         rslt = Chk(DbiDateDecode(pSysVersion->dateVer, &Da, &Mo, &Yr));
         if (rslt != DBIERR_NONE)
            return rslt;
         rslt = Chk(DbiTimeDecode(pSysVersion->timeVer, &H, &M, &Ms));
         if (rslt != DBIERR_NONE)
            return rslt;
         wsprintf(SysVer, "Engine: %d, Interface Level: %d, Date: %d/%d/%d, "
            "Time: %d:%d:%d", pSysVersion->iVersion, pSysVersion->iIntfLevel,
            Mo, Da, Yr, H, M, (Ms / 1000));
      }
   }
   return rslt;
}
```

Delphi Examples: DbiGetSysVersion

Get BDE system version information. This function can return the SYSVersion structure or clear and add the information to the SysVerList TStringList. If nil is passed in, only the SYSVersion structure is returned. This example uses the following input:

```
  Ver := fDbiGetSysVersion(MyList);
```

The function is:

```
function fDbiGetSysVersion(SysVerList: TStringList): SYSVersion;
var
  Month, Day, iHour, iMin, iSec: Word;
  Year: SmallInt;
begin
  Check(DbiGetSysVersion(Result));
  if (SysVerList <> nil) then begin
    with SysVerList do begin
      Clear;
      Add(Format('ENGINE VERSION=%d', [Result.iVersion]));
      Add(Format('INTERFACE LEVEL=%d', [Result.iIntfLevel]));
      Check(DbiDateDecode(Result.dateVer, Month, Day, Year));
      Add(Format('VERSION DATE=%s', [DateToStr(EncodeDate(Year, Month,
        Day))]));
      Check(DbiTimeDecode(Result.timeVer, iHour, iMin, iSec));
      Add(Format('VERSION TIME=%s', [TimeToStr(EncodeTime(iHour, iMin,
        Sec div 1000, iSec div 100))]));
    end;
  end;
end;
```

DbiGetTableOpenCount {button C Examples,JI(`>example',`exdbigettableopencount')} {button Delphi Examples,JI(`>example',`dexdbigettableopencount')}

C syntax

```
DBIResult DBIFN DbiGetTableOpenCount (hDb, pszTableName, [pszDriverType],
 piOpenCount);
```

Delphi syntax

```
function DbiGetTableOpenCount (hDb: hDBIDb; pszTableName: PChar;
  pszDriverType: PChar; var iOpenCount: Word): DBIResult stdcall;
```

Description

DbiGetTableOpenCount returns the total number of cursors that are open on the specified table.

Parameters

hDb Type: hDBIDb (Input)
Specifies the database handle.

pszTableName Type: pCHAR (Input)
Pointer to the name of the table. For Paradox, FoxPro, Access, and dBASE, if *pszTableName* is a fully qualified name of a table, the *pszDriverType* parameter need not be specified. If the path is not included, the path name is taken from the current directory of the database associated with *hDb*.

For SQL databases, this parameter can be a fully qualified name that includes the owner name.

pszDriverType Type: pCHAR (Input)
Pointer to the table type. Optional. For Paradox, FoxPro, and dBASE tables, this parameter is required if *pszTableName* has no extension. This parameter is ignored if the database associated with *hDb* is a SQL database. *pszDriverType* can be one of the following values: szDBASE, szMSACCESS, or szPARADOX.

piOpenCount Type: pUINT16 (Output)
Pointer to the client variable that receives the number of cursors opened on the table.

Usage

This function returns the total number of cursors open on this table by this instance of BDE, irrespective of database and current session.

Most of the functions that operate on tables require a cursor, which is obtained by calling DbiOpenTable. A table can be opened more than once, resulting in more than one cursor for that table. Some functions, such as DbiDoRestructure, require that no cursors be opened on the table. Use this function to check for this requirement.

The name of the table (not the cursor) is input to DbiGetTableOpenCount, which returns a count of how many cursors are opened on the table. This function is useful for determining whether a table is in use.

Paradox: For Paradox, the number of open cursors includes any cursors opened implicitly by referential integrity or look up tables.

DbiResult return values

DBIERR_NONE	The table open count was returned successfully.
DBIERR_INVALIDHNDL	The specified database handle is invalid or NULL.
DBIERR_INVALIDPARAM	The specified table name or the pointer to the table name is NULL.
DBIERR_NOSUCHTABLE	The specified table name is invalid.

DBIERR_UNKNOWNTBLTYPE	The specified driver type is invalid or NULL, or the pointer to the driver type is NULL.

See also

DbiOpenTable

C Examples: DbiGetTableOpenCount

Obtain the number of cursors open on a table.

```
DBIResult fDbiGetTableOpenCount(hDBIDb hDb, pCHAR TblName, pUINT16
  piOpenCount)
{
   DBIResult      rslt;
   rslt = Chk(DbiGetTableOpenCount(hDb, TblName, NULL, piOpenCount));
   return rslt;
}
```

Delphi Examples: DbiGetTableOpenCount

Return the number of open cursors on the specified table. This example uses the following input:

```
  NumCursors := fDbiGetTableOpenCount(InterBaseTable);
```

The function is:

```
function fDbiGetTableOpenCount(Table: TTable): Word;
var
  Props: CURProps;
begin
  Check(DbiGetCursorProps(Table.Handle, Props));
  Check(DbiGetTableOpenCount(Table.DBHandle, PChar(Table.TableName),
    Props.szTableType, Result));
end;
```

DbiGetTableTypeDesc {button C Examples,JI(`>example',`exdbigettabletypedesc')} {button Delphi Examples,JI(`>example',`dexdbigettabletypedesc')}

C syntax

```
DBIResult DBIFN DbiGetTableTypeDesc (pszDriverType, pszTableType, ptblType);
```

Delphi syntax

```
function DbiGetTableTypeDesc (pszDriverType: PChar; pszTableType: PChar; var
  tblType: TBLType): DBIResult stdcall;
```

Description

DbiGetTableTypeDesc returns a description of the capabilities of the table type given in *pszTableType* for the driver type given in *pszDriverType*.

Parameters

pszDriverType Type: pCHAR (Input)
Pointer to the driver type.

pszTableType Type: pCHAR (Input)
Pointer to the table type. Use DbiOpenTableTypesList to get a list of valid table types.

ptblType Type: pTBLType (Output)
Pointer to the client-allocated TBLType structure.

Usage

SQL: The table type distinguishes between views, queries, and tables. It does not identify the driver type.

DbiResult return values

DBIERR_NONE The table type Description was returned successfully.

DBIERR_INVALIDHNDL	The pointer to the driver type is NULL, or the pointer to the table type is NULL, or *pTblType* is NULL.
DBIERR_UNKNOWNDRVTYPE	The specified driver type is invalid or NULL, or the specified table type is invalid or NULL.

See also

DbiOpenTableTypesList

C Examples: DbiGetTableTypeDesc

An example for this function is under development and will be provided in an upcoming Help release.

Delphi Examples: DbiGetTableTypeDesc

Retrieve a description of the capabilities of the table type

The description is appended to the TStringList passed in. This example uses the following input:

```
  fDbiGetTableTypeDesc(szPARADOX, 'PDOX 7.0', MyTableInfo);
  fDbiGetTableTypeDesc(szDBASE, 'DBASE5', MyTableInfo);
  fDbiGetTableTypeDesc('INTRBASE', 'INTERBASE', MyTableInfo);
```

The procedure is:

```
procedure fDbiGetTableTypeDesc(DriverType, TableType: PChar;
  var TableTypeInfo: TStringList);


  function BoolVal(InBool: Boolean): string;
  begin
    if InBool then Result:= 'True'
    else Result:= 'False';
  end;

var
  TableTypeRec: TBLType;
begin
  Check(DbiGetTableTypeDesc(DriverType, TableType, TableTypeRec));
  TableTypeInfo.Add('Table ID: ' + IntToStr(TableTypeRec.iId));
  TableTypeInfo.Add('Name: ' + StrPas(TableTypeRec.szName));
  TableTypeInfo.Add('Text: ' + StrPas(TableTypeRec.szText));
  TableTypeInfo.Add('Format: ' + StrPas(TableTypeRec.szFormat));
  TableTypeInfo.Add('User can Read/Write: ' +
  BoolVal(TableTypeRec.bReadWrite));
  TableTypeInfo.Add('Can create new tables: ' +
  BoolVal(TableTypeRec.bCreate));
  TableTypeInfo.Add('Can restructure: ' +
  BoolVal(TableTypeRec.bRestructure));
  TableTypeInfo.Add('Val Checks can be specified: ' +
  BoolVal(TableTypeRec.bValChecks));
  TableTypeInfo.Add('Can be protected: ' + BoolVal(TableTypeRec.bSecurity));
  TableTypeInfo.Add('Can participate in ref integrity: ' +
  BoolVal(TableTypeRec.bRefIntegrity));
  TableTypeInfo.Add('Supports primary key concept: ' +
  BoolVal(TableTypeRec.bPrimaryKey));
  TableTypeInfo.Add('Can have other indexes: ' +
  BoolVal(TableTypeRec.bIndexing));
  TableTypeInfo.Add('Number of Phy Field types supported: ' +
  IntToStr(TableTypeRec.iFldTypes));
  TableTypeInfo.Add('Max record size: ' +
  IntToStr(TableTypeRec.iMaxRecSize));
  TableTypeInfo.Add('Max fields in a table: ' +
  IntToStr(TableTypeRec.iMaxFldsInTable));
  TableTypeInfo.Add('Maximum field name length: ' +
  IntToStr(TableTypeRec.iMaxFldNameLen));
  TableTypeInfo.Add('Driver dependent table level (version): ' +
  IntToStr(TableTypeRec.iTblLevel));
end;
```

DbiGetTimeFormat {button C Examples,JI(`>example',`exdbigettimeformat')} {button Delphi Examples,JI(`>example',`dexdbigettimeformat')}

C syntax

```
DBIResult DBIFN DbiGetTimeFormat (pfmtTime);
```

Delphi syntax

```
function DbiGetTimeFormat (var fmtTime: FMTTime): DBIResult stdcall;
```

Description

DbiGetTimeFormat gets the time format for the current session.

Parameters

pfmtTime Type: pFMTTime (Output)
Pointer to the client-allocated FMTTime structure.

Usage

The time format is used by QBE for input and wildcard character matching. It is also used by batch operations (such as DbiDoRestructure and DbiBatchMove) to handle data type coercion between character and datetime or time types.

DbiResult return values

DBIERR_NONE The time format was successfully retrieved.

DBIERR_INVALIDHNDL *pfmtTime* is NULL.

See also

DbiGetNumberFormat, DbiGetDateFormat, DbiSetTimeFormat

C Examples: DbiGetTimeFormat

An example for this function is under development and will be provided in an upcoming Help release.

Delphi Examples: DbiGetTimeFormat

Obtain information about the Time Format on your system.
This example uses the following input:

```
  Memo1.Lines.Add(fDbiGetTimeFormat);
```

The function is:

```
function fDbiGetTimeFormat: string;
var
   MyTimeFormat : FMTTime;
begin
  Check(DbiGetTimeFormat(MyTimeFormat));
  SetLength(Result, 100);
  with MyTimeFormat do
    Result := Format('Separator: %s, AM: %s, PM: %s',
      [cTimeSeparator, szAmString, szPmString]);
  SetLength(Result, StrLen(PChar(Result)));
end;
```

DbiGetTranInfo {button C Examples,JI(`>example',`exdbigettraninfo')} {button Delphi Examples,JI(`>example',`dexdbigettraninfo')}

C syntax
```
DBIResult DBIFN DbiGetTranInfo (hDb, hXact, pxInfo);
```

Delphi syntax
```
function DbiGetTranInfo (hDb: hDBIDb; hXact: hDBIXact; pxInfo: pXInfo):
  DBIResult stdcall;
```

Description
DbiGetTranInfo retrieves transaction information.

Parameters
hDb Type: hDBIDb (Input)
Specifies the database handle.

hXact Type: hDBIXact (Input)
Specifies the transaction handle. If NULL, *hDb* is used; if not NULL, *hDb* is ignored.

pxInfo Type: XInfo (Output)
Pointer to the client-allocated XInfo structure.

Usage
After a successful DbiBeginTran request, the transaction state is active. The state remains active until DbiEndTran is called. While the transaction is active, the actual isolation level being used can be retrieved with this function. Since transaction nesting is currently not supported, the uNests value is unused.

Prerequisites
A valid database handle must be obtained on a database.

Completion state
Information function only; does not affect transaction processing.

DbiResult return values
DBIERR_NONE

C Examples: DbiGetTranInfo

Gets the specified transaction information.

This example uses the following input:

fDbiGetTranInfo(hDb, xTran, Buffer);

```
DBIResult fDbiGetTranInfo(hDBIDb hTmpDb, hDBIXact hXact, pCHAR XactInfo)
{
   DBIResult   rslt;
   XInfo       xinfo;
   CHAR        XState[10] = {"Unknown"};
   CHAR        XType[20] = {"Unknown"};

   rslt = Chk(DbiGetTranInfo(hTmpDb, hXact, &xinfo));
   if (rslt == DBIERR_NONE)
   {
      switch (xinfo.exState)
      {
         case xsACTIVE:
         strcpy(XState, "Active");
         break;
         case xsINACTIVE:
         strcpy(XState, "Inactive");
         break;
      }
      switch (xinfo.eXIL)
      {
         case xilDIRTYREAD:
         strcpy(XType, "Dirty Read");
         break;
         case xilREADCOMMITTED:
         strcpy(XType, "Read Committed");
         break;
         case xilREPEATABLEREAD:
         strcpy(XType, "Repeatable Read");
         break;
      }
   }
   wsprintf(XactInfo, "Transaction State: %s, Isolation Level: %s, Nests:
  %d",
            XState, XType, xinfo.uNests);
   return rslt;
}
```

Delphi Examples: DbiGetTranInfo

Get transaction information on the database handle.

This example uses the following input:

```
  fDbiGetTranInfo(Database1.Handle);
```

The procedure is:

```
procedure fDbiGetTranInfo(hTmpDb: HDBIDb);
var
  xInfoVar: XInfo;
begin
  Check(DbiGetTranInfo(hTmpDb, nil, @xInfoVar));
  case XInfoVar.eXIL of
    xilDIRTYREAD: ShowMessage('Isolation level is Uncommitted changes');
    xilREADCOMMITTED: ShowMessage('Isolation level is Committed changes');
    xilREPEATABLEREAD: ShowMessage('Isolation level is Full read
  repeatablity');
  end;
end;
```

DbiGetVchkDesc {button C Examples,JI(`>example',`exdbigetvchkdesc')} {button Delphi Examples,JI(`>example',`dexdbigetvchkdesc')}

C syntax

```
DBIResult DBIFN DbiGetVchkDesc (hCursor, iValSeqNo, pvalDesc);
```

Delphi syntax

```
function DbiGetVchkDesc (hCursor: hDBICur; iValSeqNo: Word; pvalDesc:
  pVCHKDesc): DBIResult stdcall;
```

Description

DbiGetVchkDesc retrieves the validity check descriptor identified by the validity check sequence number and the cursor.

Parameters

hCursor Type: hDBICur (Input)
Specifies the cursor handle.

iValSeqNo Type: UINT16 (Input)
The validity check sequence number. This number is between 1 and the value of *iValChecks*. The value of *iValChecks* can be obtained from the cursor properties (CURProps) structure.

pvalDesc Type: pVCHKDesc (Output)
Pointer to the client-allocated VCHKDesc structure.

Usage

If a field map is active, the *iFld*Num in the validity check descriptor reflects the field map. If any of the fields are not part of the field-mapped record, a negative number is listed.

DbiResult return values

DBIERR_NONE The descriptor was returned successfully.

See also

DbiGetCursorProps, DbiOpenVChkList

C Examples: DbiGetVchkDesc

Get the first Validity Check Descriptor for the specified cursor.

This example uses the following input:

fDbiGetVchkDesc(hCur, &VchkDesc);

```
DBIResult fDbiGetVchkDesc(hDBICur hCur, pVCHKDesc pVchkDesc)
{
   DBIResult   rslt;
   CURProps    CurProps;
   rslt = Chk(DbiGetCursorProps(hCur, &CurProps));
   if(CurProps.iValChecks > 0)
      rslt = Chk(DbiGetVchkDesc(hCur, 1, pVchkDesc));
   else
      MessageBox(0, "No validity checks on this table", "Validity Check
  Warning", MB_OK);
   return rslt;
}
```

Delphi Examples: DbiGetVchkDesc

Obtain the first validity check descriptor for the specified cursor.
This example uses the following input:

```
  fDbiGetVchkDesc(hCur, VchkDesc);
```

The procedure is:

```
procedure fDbiGetVchkDesc(hCur: hDBICur; var VchkDes: VCHKDesc);
var
   CurProp: CURProps;
begin
   Check(DbiGetCursorProps(hCur, CurProp));
  if (CurProp.iValChecks > 0) then begin
    Check(DbiGetVchkDesc(hCur, 1, @VchkDes));
    ShowMessage('Field #' + IntToStr(VchkDes.iFldNum) +
      ' has Validity Checks.');
  end
  else
    MessageDlg(' Validity Check Warning: No validity checks on this table.',
      mtError,[MBOK], 0);
end;
```

DbiInit {button C Examples,JI(`>example',`exdbiinit')} {button Delphi Examples,JI(`>example',`dexdbiinit')}

C syntax
```
DBIResult DBIFN DbiInit (pEnv);
```

Delphi syntax
```
function DbiInit (pEnv: PDbiEnv): DBIResult stdcall;
```

Description
DbiInit initializes the BDE environment.

Parameters
pEnv Type: pDBIEnv (Input)
Pointer to the DBIEnv structure. Optional. Can be used to change the working directory and the location of the configuration file, to set up the language driver, and to supply BDE with the client name.

Usage
Initializes the BDE environment. Default settings can be overwritten by supplying the appropriate settings. If *pEnv* is NULL, then BDE assumes that the start-up directory is the working directory. In this case, *szClientName* is empty and *bForceLocalInit* is FALSE.

Prerequisites
DbiInit must be called once by each client application before any other calls (DbiOpenDatabase, DbiOpenTable, and so on.) are made. The client should be familiar with the environment Parameters such as working directory, BDE configuration file path, and so on.

DbiResult return values
DBIERR_NONE	The BDE environment was initialized successfully.
DBIERR_MULTIPLEINIT	Illegal attempt to initialize BDE more than once.
DBIERR_OSACCESS	Attempting to run a BDE application from a Windows NT directory without write access; create a Windows NT TEMP directory to avoid this

See also
DbiExit, DbiDllExit

C Examples: DbiInit

Example 1: Initialize BDE with default settings.

It is strongly recommended to use this method. This example uses the following input:

fDbiInit1();

```
DBIResult fDbiInit1()
{
   DBIResult      rslt;
   rslt = Chk(DbiInit(NULL));
   return rslt;
}
```

Example 2: Initialize BDE with a different working directory.

It is not recommended to use this method for most cases. This example uses the following input:

fDbiInit2("c:\\program\\tables);

```
DBIResult fDbiInit2(pCHAR WorkDir)
{
   DBIResult      rslt;
   DBIEnv         Env;
   memset(&Env, 0, sizeof(DBIEnv));
   strcpy(Env.szWorkDir, WorkDir);
   rslt = Chk(DbiInit(&Env));
   return rslt;
}
```

Example 3: Initialize BDE with a different configuration file.

It is not recommended to use this method for most cases. This example uses the following input:

fDbiInit3("c:\\myidapi\\idapi32.cfg");

```
DBIResult fDbiInit3(pCHAR ConfigDir)
{
   DBIResult      rslt;
   DBIEnv         Env;
   memset(&Env, 0, sizeof(DBIEnv));
   strcpy(Env.szIniFile, ConfigDir);
   rslt = Chk(DbiInit(&Env));
   return rslt;
}
```

Delphi Examples: DbiInit

Initialize BDE with default values.

If you have any of the "Data Access" or "Data Controls" VCL components in your project, you should not directly call DbiInit in a Delphi application Those components will automatically call DbiInit.

If you are not using VCL database components, then the following code shows how to initialize the BDE with default values:

```
Check(dbiInit(nil));
```

DbiInitRecord {button C Examples,JI(`>example',`exdbiinitrecord')} {button Delphi Examples,JI(`>example',`dexdbiinitrecord')}

C syntax

```
DBIResult DBIFN DbiInitRecord (hCursor, pRecBuf);
```

Delphi syntax

```
function DbiInitRecord (hCursor: hDBICur; pRecBuff: Pointer): DBIResult
  stdcall;
```

Description

DbiInitRecord initializes a record buffer. This operation is required before composing a new record for insertion.

Parameters

hCursor Type: hDBICur (Input)
Specifies the cursor handle.

pRecBuf Type: pBYTE (Output)
Pointer to the client buffer that receives the initialized record buffer.

Usage

DbiInitRecord initializes the record buffer to a blank record according to the data types of the fields.

Paradox: If the table has associated default values with any of the fields, the default values are used to initialize the fields.

Completion state

The record buffer contains blank fields or default values. The position of the given cursor is not affected. The client application can use the BDE field-level functions to fill the record buffer with the appropriate values.

DbiResult return values

DBIERR_NONE	The initialization was successful.
DBIERR_INVALIDHNDL	The specified cursor handle is invalid or NULL.
DBIERR_INVALIDPARAM	The specified record buffer is NULL.

See also

DbiAppendRecord, DbiGetRecord, DbiGetNextRecord, DbiGetPriorRecord, DbiModifyRecord, DbiInsertRecord, DbiPutField, DbiSetToKey, DbiGetBlob, DbiPutBlob, DbiOpenBlob, DbiFreeBlob, DbiGetField

C Examples: DbiInitRecord

Initialize the specified record buffer.

This example uses the following input:

fDbiInitRecord(hCursor, RecBuf);

```
DBIResult fDbiInitRecord(hDBICur hTmpCur, pBYTE pTmpRecBuf)
{
   DBIResult      rslt;
   rslt = Chk(DbiInitRecord(hTmpCur, pTmpRecBuf));
   return rslt;
}
```

Delphi Examples: DbiInitRecord

An example for this function is under development and will be provided in an upcoming Help release.

DbiInsertRecord {button C Examples,JI(`>example',`exdbiinsertrecord')} {button Delphi Examples,JI(`>example',`dexdbiinsertrecord')}

C syntax

```
DBIResult DBIFN DbiInsertRecord (hCursor, [eLock], pRecBuf);
```

Delphi syntax

```
function DbiInsertRecord (hCursor: hDBICur; eLock: DBILockType; pRecBuff:
  Pointer): DBIResult stdcall;
```

Description

DbiInsertRecord inserts a new record, contained in *pRecBuf*, into the table associated with the given cursor.

Parameters

hCursor Type: hDBICur (Input)
Specifies the cursor handle.

eLock Type: DBILockType (Input)
Specifies the lock request type. Optional.

pRecBuf Type: pBYTE (Input)
Pointer to the record buffer.

Usage

The client application can optionally acquire a lock on the newly inserted record by specifying the lock type in *eLock*.

dBASE, FoxPro, Access: For dBASE, FoxPro, and Access there is no difference between DbiAppendRecord and DbiInsertRecord. The record is inserted at the end of the table. The cursor is positioned at the inserted record. If an active range exists, the cursor might be positioned at the beginning or end of the file.

Paradox: Before inserting the record, the function verifies any referential integrity requirements or validity checks that may be in place. If either fails, an error is returned and the insert operation is canceled. If a primary index is in place, the record is physically placed at a location that conforms to the primary index order. With non-indexed tables, the record is inserted before the current position.

SQL: The table must be opened for write access. After the insert, the cursor is always positioned on the inserted record.

Prerequisites

Other users cannot have a write lock, or greater, on the table. The record buffer should be initialized with DbiInitRecord, and data filled in using DbiPutField or DbiOpenBlob, and DbiPutBlob.

Completion state

After successful completion, the cursor is positioned on the new record. If the function fails, the record is not inserted and the current position of the cursor is not affected.

If the cursor has a filter or a range associated with it, the cursor might be positioned on a crack or BOF/EOF and the operation will fail if a record lock was requested.

DbiResult return values

DBIERR_NONE	The record was successfully inserted.
DBIERR_INVALIDHNDL	The specified cursor handle is invalid or NULL.
DBIERR_INVALIDPARAM	The specified record buffer is NULL.

DBIERR_FOREIGNKEYERR	The target table is a detail table in a referential integrity link, and the linking value cannot be found in the master table.
DBIERR_MINVALERR	The specified data is less than the required minimum value.
DBIERR_MAXVALERR	The specified data is greater than the required maximum value.
DBIERR_REQDERR	The field cannot be blank.
DBIERR_LOOKUPTABLEERR	The specified value was not found in the assigned lookup table.
DBIERR_KEYVIOL	The table has a unique index and the inserted key value conflicts with an existing record's key value.
DBIERR_FILELOCKED	The table is locked by another user.
DBIERR_TABLEREADONLY	Table access denied; the specified cursor handle is read-only.
DBIERR_NOTSUFFTABLERIGHTS	Insufficient table rights to insert a record (Paradox only).
DBIERR_NODISKSPACE	Insert failed due to insufficient disk space.
DBIERR_RECLOCKFAILED	Insert failed because the record could not be locked due to range or filter constraint.

See also

DbiPutField, DbiGetNextRecord, DbiGetRecord, DbiGetRelativeRecord, DbiAppendRecord, DbiModifyRecord

C Examples: DbiInsertRecord

Append or insert a record on the specified table.

If the table has an index, insert the record; otherwise, append the record. This example uses the following input:

fAddRecord(hCur, pRecBuf);

```
DBIResult fAddRecord(hDBICur hTmpCur, pBYTE pTmpRecBuf)
{
   DBIResult        rslt;
   CURProps         CurProps;
   rslt = Chk(DbiGetCursorProps(hTmpCur, &CurProps));
   if (rslt != DBIERR_NONE)
      return rslt;
   if (CurProps.iIndexes == 0)
      rslt = Chk(DbiAppendRecord(hTmpCur, pTmpRecBuf));
   else
      rslt = Chk(DbiInsertRecord(hTmpCur, dbiNOLOCK, pTmpRecBuf));
   return rslt;
}
```

Delphi Examples: DbiInsertRecord

Append or insert a record on the specified table.

Use the Insert and Post methods on a dataset to insert records. This inserts a new record, contained in *pTmpRecBuf*, into the table associated with the given cursor. This example uses the following input:

```
  fDbiInsertRecord(hCursor, RecBuf);
```

The procedure is:

```
procedure fDbiInsertRecord(hTmpCur: hDBICur; pTmpRecBuf: pByte);
begin
  Check(DbiInsertRecord(hTmpCur, dbiNOLOCK, pTmpRecBuf));
end;
```

Insert or append a record to a table.

If the table has an index, insert the record. Most Delphi users should use: TTable.Insert, TTable.Append, TTable.InsertRecord, or TTable.AppendRecord. This example uses the following input:

```
  fDbiInsertRecord(Table1.Handle, pRecBuf: pBYTE);
```

The procedure is:

```
procedure fDbiInsertRecord(hTmpCur: hDBICur; pRec: pBYTE);
var
  Props: CURProps;
begin
  Check(DbiGetCursorProps(hTmpCur, Props));
  // Check to see if there are any active indexes on the table
  if (Props.iIndexes > 0) then
  // Insert the record
    Check(DbiInsertRecord(hTmpCur, dbiNOLOCK, pRec))
  else
  // Append the record
    Check(DbiAppendRecord(hTmpCur, pRec));
end;
```

DbiIsRecordLocked {button C Examples,JI(`>example',`exdbiisrecordlocked')} {button Delphi Examples,JI(`>example',`dexdbiisrecordlocked')}

C syntax

```
DBIResult DBIFN DbiIsRecordLocked (hCursor, pbLocked);
```

Delphi syntax

```
function DbiIsRecordLocked (hCursor: hDBICur; var bLocked: Bool): DBIResult
  stdcall;
```

Description

DbiIsRecordLocked is used to check if current record is locked in the current session.

Parameters

hCursor Type: hDBICur (Input)
Specifies the cursor handle.

pbLocked Type: pBOOL (Output)
Pointer to the client variable. Set to TRUE if the record is locked; otherwise, FALSE.

Usage

Record locks differ from table locks in that they only have two states: locked or not locked. Table locks have four states: no lock, read lock, write lock, or exclusive lock.

Prerequisites

The cursor must be positioned on a record.

Completion state

The lock status is returned in *pLocked*, and indicates whether the record is locked by any application in the current session.

SQL: For SQL, the lock status returned in *pLocked* indicates whether the record is locked by you.

DbiResult return values

DBIERR_NONE	The lock status was returned successfully.
DBIERR_NOCURRREC	There is no current record.
DBIERR_BOF	There is no current record at the beginning of the file.
DBIERR_EOF	There is no current record at the end of the file.
DBIERR_KEYORRECDELETED	There is no current record.
DBIERR_INVALIDHNDL	The specified cursor handle is invalid or NULL.
DBIERR_INVALIDPARAM	pbLocked is NULL.

See also

DbiGetNextRecord, DbiGetPriorRecord, DbiGetRecord, DbiGetRelativeRecord, DbiRelRecordLock, DbiIsTableLocked, DbiAcqTableLock, DbiRelTableLock

C Examples: DbiIsRecordLocked

An example for this function is under development and will be provided in an upcoming Help release.

Delphi Examples: DbiIsRecordLocked

Checks the lock status of the current record.

This example uses the following input:

```
  IsRecordLocked(Table1, True);
```

NOTE: If ByAnyone is true, then the function also checks if any other session has the record locked. If ByAnyone is false, then only the current session is checked. The function returns False if not locked, and True if locked.

The function is:

```
function IsRecordLocked(Table: TTable; ByAnyone: Boolean): Boolean;
var
  Locked: BOOL;
  hCur: hDBICur;
  rslt: DBIResult;
begin
  Table.UpdateCursorPos;
  // Is the record locked by the current session...
  Check(DbiIsRecordLocked(Table.Handle, Locked));
  Result := Locked;
  // If the current session does not have a lock and the ByAnyone varable is
  // set to check all sessions, continue check...
  if (not Result) and (ByAnyone) then begin
    // Get a new cursor to the same record...
    Check(DbiCloneCursor(Table.Handle, False, False, hCur));
    try
      // Try and get the record with a write lock...
      rslt := DbiGetRecord(hCur, dbiWRITELOCK, nil, nil);
      if (rslt <> DBIERR_NONE) then begin
        // if an error occured and it is a lock error, return true...
        if (HiByte(rslt) = ERRCAT_LOCKCONFLICT) then
          Result := True
        else
          // If some other error happened, throw an exception...
          Check(rslt);
      end
      else
        // Release the lock in this session if the function was
  successful...
        Check(DbiRelRecordLock(hCur, False));
    finally
      // Close the cloned cursor...
      Check(DbiCloseCursor(hCur));
    end;
  end;
end;
```

DbiIsTableLocked {button C Examples,JI(`>example',`exdbiistablelocked')} {button Delphi Examples,JI(`>example',`dexdbiistablelocked')}

C syntax

```
DBIResult DBIFN DbiIsTableLocked (hCursor, edbiLock, piLocks);
```

Delphi syntax

```
function DbiIsTableLocked (hCursor: hDBICur; epdxLock: DBILockType; var
  iLocks: Word): DBIResult stdcall;
```

Description

DbiIsTableLocked returns the number of locks of type *edbiLock* acquired on the table associated with the given session.

Parameters

hCursor Type: hDBICur (Input)
Specifies the cursor handle.

edbiLock Type: DBILockType (Input)
Specifies the lock type to verify.

piLocks Type: pUINT16 (Output)
Pointer to the client variable that receives the number of locks of the given lock type.

Usage

dBASE or FoxPro: For dBASE and FoxPro tables, dbiREADLOCKs are upgraded to dbiWRITELOCKs. If the value of *edbiLock* is dbiREADLOCK, then the number of write locks are returned in *piLocks*.

DbiResult return values

DBIERR_NONE	The number of locks was returned successfully.
DBIERR_INVALIDHNDL	The specified cursor handle is invalid or NULL.
DBIERR_INVALIDPARAM	*piLocks* is NULL.

See also

DbiAcqTableLock, DbiRelTableLock, DbiOpenLockList

edbiLock
edbiLock can be one of the following values:

Value	Description
dbiNOLOCK	Dirty read
dbiREADLOCK	Read lock
dbiWRITELOCK	Write lock

C Examples: DbiIsTableLocked

An example for this function is under development and will be provided in an upcoming Help release.

Delphi Examples: DbiIsTableLocked

Return the number of locks of type edbiLock aquired on the table passed to the function.

You can obtain table locks by calling the LockTable and UnLockTable methods of the TTable component. This example uses the following input:

```
  LockTotal:= fDbiIsTableLocked(Table1, dbiWRITELOCK);
```

The function is:

```
function fDbiIsTableLocked(Tbl: TTable; Lock: DBILockType): Word;
var
  NumLocks: Word;
begin
  Check(DbiIsTableLocked(Tbl.Handle, Lock, NumLocks));
  Result:= NumLocks;
end;
```

DbiIsTableShared {button C Examples,JI(`>example',`exdbiistableshared')} {button Delphi Examples,JI(`>example',`dexdbiistableshared')}

C syntax

```
DBIResult DBIFN DbiIsTableShared (hCursor, pbShared);
```

Delphi syntax

```
function DbiIsTableShared (hCursor: hDBICur; var bShared: Bool): DBIResult
  stdcall;
```

Description

DbiIsTableShared determines whether the table is physically shared or not.

Parameters

hCursor Type: hDBICur (Input)
Specifies the cursor handle.

pbShared Type: pBOOL (Output)
Pointer to the client variable. Set to TRUE if the table is physically shared.

Usage

Standard: The table is physically shared if it is placed on a shared drive (network, or local drive when LOCALSHARE in the configuration is TRUE), and the table is not opened exclusively. If a table is shared, dirty data is not buffered. The table is available to all users in the session, unless acquired table or record locks have been placed since the table was opened.

DbiResult return values

DBIERR_NONE	The table shared status was returned successfully.
DBIERR_INVALIDHNDL	The specified cursor handle is invalid or NULL.
DBIERR_INVALIDPARAM	*pbShared* is NULL.

See also

DbiOpenTable, DbiAcqTableLock, DbiAcqPersistTableLock, DbiRelTableLock, DbiRelPersistTableLock, DbiForceReread, DbiCheckRefresh

C Examples: DbiIsTableShared

An example for this function is under development and will be provided in an upcoming Help release.

Delphi Examples: DbiIsTableShared

An example for this function is under development and will be provided in an upcoming Help release.

DbiLinkDetail {button C Examples,JI(`>example',`exdbilinkdetail')} {button Delphi Examples,JI(`>example',`dexdbibeginlinkmode')}

C syntax

```
DBIResult DBIFN DbiLinkDetail (hMstrCursor, hDetlCursor, iLnkFields,
  piMstrFields, piDetlFields);
```

Delphi syntax

```
function DbiLinkDetail (hMstrCursor: hDBICur; hDetlCursor: hDBICur;
  iLnkFields: Word; piMstrFields: PWord; piDetlFields: PWord): DBIResult
  stdcall;
```

Description

DbiLinkDetail establishes a link between two cursors such that the detail cursor has its record set limited to the set of records matching the linking key values of the master cursor.

Parameters

hMstrCursor Type: hDBICur (Input)
Specifies the cursor handle associated with the master table. The cursor does not have to be opened on an index.

hDetlCursor Type: hDBICur (Input)
Specifies the cursor handle associated with the detail table. The cursor must be opened on an index corresponding to all the link fields.

iLnkFields Type: UINT16 (Input)
Specifies the number of link fields.

piMstrFields Type: pUINT16 (Input)
Pointer to the array of field numbers of link fields in the master table.

piDetlFields Type: pUINT16 (Input)
Pointer to the array of field numbers of link fields in the detail table.

Usage

This function is useful for establishing one-to-one or one-to-many relationships between tables. A master cursor can have more than one detail cursor; a detail cursor can have only one master cursor. A detail cursor can also be a master cursor. Links apply to all available driver types; they can be established between cursors of the same or different driver types. The effect is equivalent to setting a range using DbiSetRange on the detail table and using the linking fields of the master table.

Prerequisites

For the cursors to be linked, both cursors must be enabled with DbiBeginLinkMode. The data types of linked fields in master and detail records must be compatible. The detail cursor must be opened on an index corresponding to all of the linking fields. For expression links, see DbiLinkDetailToExp.

Completion state

The linked cursors are modified so that the detail cursor allows access only to the records that match the linking value of the master record. If the position of the master cursor changes so that a different linking value is obtained for the linking fields, the detail cursor is set to a new range of records and is positioned to the beginning of this range.

DbiResult return values

DBIERR_NONE The link between the detail cursor (hDetlCursor) and the master cursor (hMstrCursor)

	was successfully established.
DBIERR_INVALIDHNDL	One or more of the specified cursor handles is invalid or NULL.

See also

DbiLinkDetailToExp, DbiUnlinkDetail, DbiSetRange

C Examples: DbiLinkDetail

An example for this function is under development and will be provided in an upcoming Help release.

Delphi Examples: DbiLinkDetail

An example for this function is under development and will be provided in an upcoming Help release.

DbiLinkDetailToExp {button C Examples,JI(`>example',`exdbilinkdetailtoexp')} {button Delphi Examples,JI(`>example',`dexdbilinkdetailtoexp')}

C syntax

```
DBIResult DBIFN DbiLinkDetailToExp (hCursorMstr, hCursorDetl, iKeyLen,
  pszMstrExp);
```

Delphi syntax

```
function DbiLinkDetailToExp (hCursorMstr: hDBICur; hCursorDetl: hDBICur;
  iKeyLen: Word; pszMstrExp: PChar): DBIResult stdcall;
```

Description

DbiLinkDetailToExp links the detail cursor to the master cursor using a dBASE expression.

Parameters

hCursorMstr Type: hDBICur (Input)
Specifies the cursor handle associated with the master table. Must be a cursor on a dBASE table. The cursor does not have to be opened on an index.

hCursorDetl Type: hDBICur (Input)
Specifies the cursor handle associated with the detail table. The cursor must be ordered on an index corresponding to the provided expression, and the cursor must be open on a dBASE table.

iKeyLen Type: UINT16 (Input)
Specifies the length of the key to match.

pszMstrExp Type: pCHAR (Input)
Pointer to the expression string. Must be a valid dBASE expression whose key type is the same as the active index of the detail table.

Usage

This function is supported by the dBASE driver only.

dBASE and FoxPro: This function is used to establish one-to-many or one-to-one relationships, using expressions. This function is used to create linked cursors so that the master cursor is on a dBASE table and the link is a dBASE-style expression, not a set of fields.

Prerequisites

hCursorMstr and *hCursorDetl* must be link cursors. This is done by calling DbiBeginLinkMode for both master and detail cursor. For the tables to be linked, both cursor handles must be obtained on a dBASE or FoxPro table.

Completion state

The linked cursors are set up such that the detail cursor shows only records that match the linking value of the master record.

DbiResult return values

DBIERR_NONE	The specified detail cursor was successfully linked to the specified master cursor.
DBIERR_INVALIDHNDL	One or more of the specified cursor handles is invalid or NULL.
DBIERR_INVALIDLINKEXPR	The expression used was invalid.

See also

DbiLinkDetail, DbiUnlinkDetail

C Examples: DbiLinkDetailToExp

An example for this function is under development and will be provided in an upcoming Help release.

Delphi Examples: DbiLinkDetailToExp

An example for this function is under development and will be provided in an upcoming Help release.

DbiLoadDriver {button C Examples,JI(`>example',`exdbiloaddriver')} {button Delphi Examples,JI(`>example',`dexdbiloaddriver')}

C syntax
```
DBIResult DBIFN DbiLoadDriver (pszDriverType);
```

Delphi syntax
```
function DbiLoadDriver (pszDriverType: PChar): DBIResult stdcall;
```

Description
DbiLoadDriver loads a given driver. Use DbiOpenDriverList to get list of valid drivers.

Parameters
pszDriverType Type: pCHAR (Input)
Pointer to the driver name.

DbiResult return values
DBIERR_NONEThe driver has been loaded successfully.

See also
DbiOpenDriverList

C Examples: DbiLoadDriver

An example for this function is under development and will be provided in an upcoming Help release.

Delphi Examples: DbiLoadDriver

Load a given driver.
This example uses the following input:

```
  fDbiLoadDriver('PARADOX')
```

The procedure is:

```
procedure fDbiLoadDriver(DriverType: string);
begin
  Check(DbiLoadDriver(PChar(DriverType)));
end;
```

DbiMakePermanent {button C Examples,JI(`>example',`exdbimakepermanent')} {button Delphi Examples,JI(`>example',`dexdbimakepermanent')}

C syntax

```
DBIResult DBIFN DbiMakePermanent (hCursor, [pszName], bOverWrite);
```

Delphi syntax

```
function DbiMakePermanent (hCursor: hDBICur; pszName: PChar; bOverWrite:
  Bool): DBIResult stdcall;
```

Description

DbiMakePermanent changes a temporary table created by DbiCreateTempTable into a permanent table, optionally renaming it using *pszName*.

Parameters

hCursor Type: hDBICur (Input)
Specifies the cursor handle.

pszName Type: pCHAR (Input)
Pointer to the name of the permanent table.

bOverWrite Type: BOOL (Input)
If set to TRUE, overwrites the existing file.

Usage

This function is used to change a temporary table, created with DbiCreateTempTable, into a permanent table, that is, one that will not be deleted when the cursor is closed with DbiCloseCursor. DbiSaveChanges can also be used to make the temporary table permanent, but the table is flushed out to disk immediately. With DbiMakePermanent, buffers are flushed to disk when convenient, or when the cursor is closed. The table is renamed to *pszName* if different from NULL.

SQL: This function is not supported by SQL drivers.

Prerequisites

A temporary table must have been created with DbiCreateTempTable.

Completion state

The table is saved to disk when the cursor is closed.

DbiResult return values

DBIERR_NONE	The temporary table has been designated as a permanent table.

See also

DbiSaveChanges, DbiCreateTempTable, DbiCloseCursor, DbiQInstantiateAnswer

C Examples: DbiMakePermanent

An example for this function is under development and will be provided in an upcoming Help release.

Delphi Examples: DbiMakePermanent

An example for this function is under development and will be provided in an upcoming Help release.

DbiModifyRecord {button C Examples,JI(`>example',`exdbimodifyrecord')} {button Delphi Examples,JI(`>example',`dexdbimodifyrecord')}

C syntax

```
DBIResult DBIFN DbiModifyRecord (hCursor, pRecBuf, bFreeLock);
```

Delphi syntax

```
function DbiModifyRecord (hCursor: hDBICur; pRecBuf: Pointer; bFreeLock:
  Bool): DBIResult stdcall;
```

Description

DbiModifyRecord modifies the current record of the table associated with *hCursor* with the data supplied in *pRecBuf*.

Parameters

hCursor Type: hDBICur (Input)
Specifies the cursor handle for the table. The cursor must be positioned on a valid record.

pRecBuf Type: pBYTE (Input)
Pointer to the client buffer where the modified record is stored.

bFreeLock Type: BOOL (Input)
Specifies whether to release locks on completion. If set to TRUE, the lock is released on the updated record when DbiModifyRecord completes. If set to FALSE, the lock is not released.

Usage

Paradox: Before the table is updated, any referential integrity requirements or validity checks in place are verified. If any fail, an error is returned and the operation is canceled.

SQL: Tables must be opened with write access. If the table has no unique index or server row ID (this includes views), DbiModifyRecord can be used to modify records if the server supports it. However, if you attempt to modify a record that has a duplicate, you will receive an error.

If the record is locked (using dbiREADLOCK or dbiWRITELOCK), and the user tries to modify the record after another user has deleted the record or changed the key value for the record, DbiModifyRecord returns a DBIERR_KEYORRECDELETED error.

Prerequisites

The cursor must be positioned on a record, not on a crack, beginning of file, or end of file. The user must have read-write access to the table. The record must not be locked by another session.

Completion state

The cursor is positioned on the updated record. An error is returned if there is no current record for the cursor. If the key has changed, DbiModifyRecord is equivalent to calling first DbiDeleteRecord then DbiInsertRecord. When a record is modified in a table that has an active index, the position of the modified record may change if the key value was modified.

If the client requests to keep a lock on a modified record, and the record flies outside a current range or filter condition, the function returns DBIERR_RECLOCKFAILED and the operation fails.

DbiResult return values

DBIERR_NONE The record was modified successfully.

DBIERR_KEYVIOL The table has a unique index and the modified key value conflicts with another record's key value.

DBIERR_BOF/EOF	The cursor is not positioned on a valid record; it is positioned at the beginning or the end of the table.
DBIERR_FILELOCKED	The table is locked by another user.
DBIERR_INVALIDHNDL	The specified cursor handle is invalid or NULL.
DBIERR_INVALIDPARAM	The specified record buffer is NULL.
DBIERR_KEYORRECDELETED	The specified cursor is not positioned on a valid record.
DBIERR_FOREIEGNKEYERR	The target table is a detail table in a referential integrity link and the linking value cannot be found in the master table (Paradox only).
DBIERR_MINVALERR	The specified data is less than the required minimum value.
DBIERR_MAXVALERR	The specified data is greater than the required maximum value.
DBIERR_REQDERR	The field cannot be blank.
DBIERR_LOOKUPTABLEERR	The specified value cannot be located in the assigned lookup table.
DBIERR_NOTSUFFTABLERIGHTS	Insufficient table rights to update table.
DBIERR_TABLEREADONLY	The specified cursor is read-only.
DBIERR_RECLOCKFAILED	The record lock failed.
DBIERR_MULTIPLUNIQRECS	Attempt to modify a record that has a duplicate (SQL).

See also

DbiDeleteRecord, DbiInitRecord, DbiPutField, DbiGetNextRecord, DbiGetRecord, DbiGetField, DbiAppendRecord, DbiInsertRecord, DbiGetBlob, DbiPutBlob, DbiOpenBlob, DbiFreeBlob

C Examples: DbiModifyRecord

Modify the current record and remove the record lock (if one exists)

This example uses the following input:

fDbiModifyRecord(hCur, pRecBuf);

```
DBIResult fDbiModifyRecord(hDBICur hTmpCur, pBYTE pTmpRecBuf)
{
   DBIResult      rslt;
   rslt = Chk(DbiModifyRecord(hTmpCur, pTmpRecBuf, TRUE));
   return rslt;
}
```

Delphi Examples: DbiModifyRecord

Modify the current record and remove the record lock (if one exists)

Delphi programs should use the Post and Edit methods on a dataset to modify a record. This procedure modifies the current record with the contents of the given record buffer.

```
procedure fDbiModifyRecord(hCur: hDbiCur; pRecBufModify: pByte);
begin
  Check(DbiModifyRecord(hCur, pRecBufModify, True));
end;
```

DbiNativeToAnsi {button C Examples,JI(`>example',`exdbinativetoansi')} {button Delphi Examples,JI(`>example',`dexdbinativetoansi')}

C syntax

```
DBIResult DBIFN DbiNativeToAnsi (pLdObj, pAnsiStr, pOemStr, iLen,
  pbDataLoss);
```

Delphi syntax

```
function DbiNativeToAnsi (LdObj: Pointer; pAnsiStr: PChar; pNativeStr:
  PChar; iLen: Word; var bDataLoss: Bool): DBIResult stdcall;
```

Description

DbiNativeToAnsi translates strings from the language driver's native character set to ANSI. If the native character set is ANSI, no translation takes place.

Parameters

pLdObj Type: pVOID (Input)
Pointer to the language driver object returned from DbiGetLdObj.

pAnsiStr Type: pCHAR (Output)
Pointer to the client buffer that returns the ANSI data. If pAnsiStr equals *pOemStr*, conversion occurs in place.

pOemStr Type: pCHAR (Input)
Pointer to the buffer containing data to be translated.

iLen Type: UINT16 (Input)
If *iLen* equals 0, assumes null-terminated string; otherwise, *iLen* specifies the length of the buffer to convert.

pbDataLoss Type: pBOOL (Output)
Pointer to a client variable. Set to TRUE if a character cannot map to an ANSI character.

Usage

This function works on drivers having both ANSI and OEM native character sets, but it does not deal with multi-byte character sets such as Japanese ShiftJIS. If the native character set is ANSI, no translation takes place.

DbiResult return values

DBIERR_NONE Translation completed successfully.

See also

DbiAnsiToNative, DbiGetLdObj

C Examples: DbiNativeToAnsi

An example for this function is under development and will be provided in an upcoming Help release.

Delphi Examples: DbiNativeToAnsi

An example for this function is under development and will be provided in an upcoming Help release.

DbiOpenBlob {button C Examples,JI(`>example',`exdbigetblob')} {button Delphi Examples,JI(`>example',`dexdbigetblob')}

C syntax

```
DBIResult DBIFN DbiOpenBlob (hCursor, pRecBuf, iField, eOpenMode);
```

Delphi syntax

```
function DbiOpenBlob (hCursor: hDBICur; pRecBuf: Pointer; iField: Word;
  eOpenMode: DBIOpenMode): DBIResult stdcall;
```

Description

DbiOpenBlob prepares the cursor's record buffer to access a BLOB field. The BLOB is opened and the BLOB handle is stored in the record buffer, which can then be passed to DbiGetBlob, DbiPutBlob, and other BLOB functions.

Parameters

hCursor Type: hDBICur (Input)
Specifies the cursor handle.

pRecBuf Type: pBYTE (Input)
Pointer to the record buffer.

iField Type: UINT16 (Input)
Specifies the ordinal number of the BLOB field within the record.

eOpenMode Type: DBIOpenMode (Input)
Specifies the BLOB open mode.

If dbiREADWRITE is specified, both the database and the table must be opened in dbiREADWRITE mode.

Usage

DbiOpenBlob opens the BLOB and stores the supplied BLOB handle in *pRecBuf* so that all or portions of the BLOB field can be retrieved, modified, deleted, or inserted, and the size of the field can be determined. The BLOB field can be opened in either read-only or read-write mode, depending on the value specified in *eOpenMode*.

DbiOpenBlob must be called prior to calling the BLOB functions DbiGetBlobSize, DbiGetBlob, DbiPutBlob, DbiTruncateBlob, or DbiFreeBlob.

Standard: It is advisable to lock the record before opening the BLOB in read-write mode. This ensures that another client application does not lock the record or update the BLOB, preventing the record from being updated.

SQL: This function is supported by SQL drivers. However, for SQL servers that do not support BLOB handles for random reads and writes, full BLOB support requires uniquely identifiable rows. Most SQL servers limit a single sequential BLOB read to less than the maximum size of a BLOB. In cases with no row uniqueness and without BLOB handles, an entire BLOB may not be available.

Completion state

DbiOpenBlob fails if the client application does not have sufficient rights to access the BLOB field. To close a BLOB field after it has been opened with DbiOpenBlob, a call to DbiFreeBlob must be made.

DbiResult return values

DBIERR_NONE	The BLOB field was successfully opened.
DBIERR_INVALIDHNDL	The specified cursor handle is invalid or NULL.
DBIERR_INVALIDPARAM	The specified record buffer is NULL.

DBIERR_OUTOFRANGE	The specified field number is equal to zero, or is greater than the number of fields in the table.
DBIERR_BLOBOPENED	The specified BLOB field is already open.
DBIERR_NOTABLOB	The specified field number does not correspond to a BLOB field.
DBIERR_OPENBLOBLIMIT	The allowed number of open BLOB handles for the current driver has been exceeded.
DBIERR_TABLEREADONLY	The BLOB cannot be opened in read-write mode; the table is read-only.

See also

DbiGetBlob, DbiPutBlob, DbiTruncateBlob, DbiFreeBlob, DbiGetBlobSize

DbiOpenCfgInfoList {button C Examples,JI(`>example',`exdbiopencfginfolist')} {button Delphi Examples,JI(`>example',`dexdbiopencfginfolist')}

C syntax

```
DBIResult DBIFN DbiOpenCfgInfoList (hCfg, eOpenMode, eConfigMode,
  pszCfgPath, phCur);
```

Delphi syntax

```
function DbiOpenCfgInfoList (hCfg: hDBICfg; eOpenMode: DBIOpenMode;
  eConfigMode: CFGMode; pszCfgPath: PChar;
 var hCur: hDBICur): DBIResult stdcall;
```

Description

DbiOpenCfgInfoList returns a handle to a list of all the nodes in the BDE configuration file accessible by the specified path.

Parameters

hCfg Type: hDBICfg (Input)
Specifies the configuration file handle; must be NULL.

eOpenMode Type: DBIOpenMode (Input)
Specifies the open mode; choose dbiREADWRITE or dbiREADONLY.

eConfigMode Type: CFGMode (Input)
Specifies the configuration mode; only cfgPersistent is supported.

pszCfgPath Type: pCHAR (Input)
Pointer to the configuration file path name used to locate a piece of information within the configuration file. The path name starts at the root, denoted by a backslash (\). As many levels as necessary to locate the target piece of information may be specified. Each node specified in the path name must have at least one subnode or an error results. The path name must be NULL-terminated. See the Usage section for an example. If the path is a valid ODBC driver name and data source not stored in the configuration file, the default parameter settings for that driver and data source are returned.

phCur Type: phDBICur (Output)
Pointer to a cursor handle.

Usage

This function can be used to retrieve information from the configuration file about BDE drivers, internal buffers, and aliases by supplying a known path in *pszConfigPath*.

DbiOpenCfgInfoList accesses the same configuration file that was used when BDE was initialized. If no configuration file was used during DbiInit, an empty table is returned.

The full path name is supplied by *pszConfigPath*, starting at the root, and then subsequently specifying the name of a node, a backslash (\), one of the node's subnodes, and so on until the desired level is reached. For example, to retrieve the values used to initialize BDE, the *pszConfigPath* passed in would be:

```
\system\init
```

phCur then receives the handle to a table containing a list of records, each representing a node accessible by the specified path name. The cursor is used by subsequent record manipulation calls such as DbiGetNextRecord and DbiGetPriorRecord. DbiGetCursorProps can be used to allocate the proper record size or the client application can allocate the size of the CFGDesc structure. After the record is retrieved it can be cast with the CFGDesc type definition and used as if it is a CFGDesc C language structure.

DbiModifyRecord can also be used with the cursor with the following restrictions:

- *szValue* is the only field that can be updated.
- Only leaf nodes can be modified.

This function can also be used to build a path name to a target piece of information within the configuration file, when the path name is not known. In that case, the first call to DbiOpenCfgInfoList is passed with *pszConfigPath* set to backslash (\). The table returned lists all the nodes accessible to the root. If these nodes do not contain the target information (in *szText*[*MAXSCFLDLEN*]), subsequent calls to DbiOpenCfgInfoList can be made, each one extending the path name to access one level deeper in the configuration file.

You can use the read-only property sesCFGNAME to retrieve the name of the configuration file used by the active session.

Note: The session property sesCFGMODE2 can affect the list returned by this function.

Prerequisites

The database engine must be initialized with a configuration file.

DbiResult return values

DBIERR_NONE	The handle to the table listing configuration file information was returned successfully.
DBIERR_INVALIDHNDL	The specified cursor handle is invalid or NULL.
DBIERR_INVALIDPARAM	The specified record buffer is NULL.

See also

DbiInit, DbiOpenDatabaseList, DbiOpenDriverList

C Examples: DbiOpenCfgInfoList

An example for this function is under development and will be provided in an upcoming Help release.

Delphi Examples: DbiOpenCfgInfoList

Returns a handle to an in-memory table listing all the nodes in the configuration file accessible by the specified path.

WARNING: Be extremely careful when altering the IDAPI.CFG configuration file. Make absolutely sure that all options and parameters are correct or corruption of the configuration file can, and more than likely, occur.

Example 1: Retrieve a particular value from the IDAPI.CFG configuration file.

This example uses the following input:

```
  Edit1.Text := GetConfigParameter(PARADOXLEVEL, @Count);
```

NOTE: Param (in this case PARADOXLEVEL) must be a string that contains the path to the node and the node item separated by a semi-colon. At the bottom of this page are some of the more popular paths and items that are declared as constants for use with all these examples.

The function is:

```
function GetConfigParameter(Param: string; Count: pword): string;
var
  hCur: hDBICur;
  rslt: DBIResult;
  Config: CFGDesc;
  Path, Option: string;
  Temp: array[0..255] of char;
begin
  Result := '';
  hCur := nil;
  if (Count <> nil) then
    Count^ := 0;
  try
    if (Pos(';', Param) = 0) then
      raise EDatabaseError.Create('Invalid parameter passed to' +
        'function.  There must be a semi-colon delimited sting passed');
    Path := Copy(Param, 0, Pos(';', Param) - 1);
    Option := Copy(Param, Pos(';', Param) + 1, Length(Param) - Pos(';',
  Param));
    Check(DbiOpenCfgInfoList(nil, dbiREADONLY, cfgPERSISTENT,
      StrPCopy(Temp, Path), hCur));
    Check(DbiSetToBegin(hCur));
    repeat
      rslt := DbiGetNextRecord(hCur, dbiNOLOCK, @Config, nil);
      if (rslt = DBIERR_NONE) then begin
        if (StrPas(Config.szNodeName) = Option) then
          Result := Config.szValue;
        if (Count <> nil) then
          Inc(Count^);
      end
      else
        if (rslt <> DBIERR_EOF) then
          Check(rslt);
    until (rslt <> DBIERR_NONE);
  finally
    if (hCur <> nil) then
      Check(DbiCloseCursor(hCur));
  end;
```

```
end;
```

DbiOpenDatabase {button C Examples,JI(`>example',`exdbiopendatabase')} {button Delphi Examples,JI(`>example',`dexdbiopendatabase')}

C syntax

```
DBIResult DBIFN DbiOpenDatabase (pszDbName, pszDbType, eOpenMode,
  eShareMode, [pszPassword], iOptFlds, pOptFldDesc, pOptParams, phDb);
```

Delphi syntax

```
function DbiOpenDatabase (pszDbName: PChar; pszDbType: PChar; eOpenMode:
  DBIOpenMode; eShareMode: DBIShareMode; pszPassword: PChar; iOptFlds: Word;
  pOptFldDesc: pFLDDesc; pOptParams: Pointer; var hDb: hDBIDb): DBIResult
  stdcall;
```

Description

DbiOpenDatabase is called to open a database in the current session. On success, a database handle is returned.

Parameters

pszDbName Type: pCHAR (Input)
Pointer to the alias name string defined in the configuration file. Optional. If NULL, the standard database is opened. If *pszDbName* specifies a SQL database, *pszDbType* can be NULL. If *pszDbName* specifies an ODBC data source not in the configuration file, the BDE adds the data source and ODBC driver to the session automatically.

pszDbType Type: pCHAR (Input)
Pointer to the database type string. Optional. If both *pszDbName* and *pszDbType* are NULL, a standard database is opened.

eOpenMode Type: DBIOpenMode (Input)
Specifies the open mode.

eShareMode Type: DBIShareMode (Input)
Specifies the share mode.

pszPassword Type: pCHAR (Input)
Pointer to the password string. Optional. SQL only.

iOptFlds Type: UINT16 (Input)
Specifies the number of optional parameters. Refer to DbiCreateTable for use of optional parameters.

pOptFldDesc Type: pFLDDesc (Input)
Pointer to an array of field descriptors for the optional parameters. Refer to DbiCreateTable for use of optional parameters.

pOptParams Type: pBYTE (Input)
Pointer to the optional parameters required by the database. Refer to DbiCreateTable for use of optional parameters.

phDb Type: phDBIDb (Output)
Pointer to the database handle.

Usage

The database must be opened before a table can be opened in the database.

The database handle is passed into several functions. The values in *pszDbName* and *pszDbType* determine which database is opened. The *eOpenMode* and *eShareMode* parameters determine the access modes of the cursors within each database. For example, if *eOpenMode* is set to dbiREADONLY, its associated cursors are also READONLY.

ODBC: DbiOpenDatabase automatically adds ODBC drivers and data sources as BDE

aliases to the active session when they aren't currently stored in the configuration file. The BDE also supports ODBC 3 drivers.

SQL: SQL configuration file settings might override the *eOpenMode* setting.

OptFields, *pOptFldDesc* and *pOptParams* are the optional parameters. The optional parameters passed by this function vary depending on the driver. They can be identified by calling the DbiOpenCfgInfoList function.

Standard: Connecting to a standard database:

- If *pszDbName* and *pszDbType* are both set to NULL, the unnamed standard database is opened.
- If *pszDbName* specifies an alias for a standard database in the configuration file, this database is opened.

SQL: Connecting to a SQL database:

- If *pszDbName* specifies a SQL ALIAS from the configuration file, *pszDbType* is NULL, and iOptFlds is 0, a SQL database is opened. (Supply the password if required.)
- If *pszDbName* is NULL, and *pszDbType* is one of the SQL driver names (for example, Oracle, Sybase), a SQL database is opened. If optional parameters are not specified, driver-specific defaults are used.

Prerequisites

DbiInit must be called prior to calling DbiOpenDatabase. The database must be successfully opened before any other calls can be made to access or manipulate data. If the database requires login, a password must be supplied.

DbiResult return values

DBIERR_NONE	The database was successfully opened.
DBIERR_UNKNOWNDB	The specified database or database type is invalid.
DBIERR_NOCONFIGFILE	The configuration file was not found.
DBIERR_INVALIDDBSPEC	When using an alias from the configuration file, the specification is invalid.
DBIERR_DBLIMIT	The maximum number of databases have been opened.

See also

DbiOpenTableList, DbiGetDatabaseDesc

Database Types

Examples of database types include:

- STANDARD
- ORACLE
- SYBASE
- INTRBASE
- DB2
- INFORMIX
- MSACCESS

C Examples: DbiOpenDatabase

Example 1: Open a standard database using an alias name.

This example uses the following input:

fDbiOpenDatabase1(&hDb, "MyAlias");

```
DBIResult fDbiOpenDatabase1(phDBIDb phDb, pCHAR alias)
{
   DBIResult    rslt;
   rslt = Chk(DbiOpenDatabase(alias, NULL, dbiREADWRITE, dbiOPENSHARED,
            NULL, 0, NULL, NULL, phDb));
   return rslt;
}
```

Example 2: Open a standard database with no alias name.

To access tables in a directory other than the current, you must call DbiSetDirectory. This example uses the following input:

fDbiOpenDatabase2(&hDb, "C:\\MyDir\\Tables");

```
DBIResult fDbiOpenDatabase2(phDBIDb phDb, pCHAR Directory)
{
   DBIResult   rslt;
   rslt = Chk(DbiOpenDatabase(NULL, NULL, dbiREADWRITE, dbiOPENSHARED,
                              NULL, 0, NULL, NULL, phDb));
   if (rslt != DBIERR_NONE)
      return rslt;
   rslt = Chk(DbiSetDirectory(*phDb, Directory));
   return rslt;
}
```

Example 3: Open a remote database with alias and password..

This example uses the following input:

fDbiOpenDatabase3(&hDb, "MyAlias", "MyPswd");

```
DBIResult fDbiOpenDatabase3(phDBIDb phDb, pCHAR alias, pCHAR password)
{
   DBIResult    rslt;
   rslt = Chk(DbiOpenDatabase(alias, NULL, dbiREADWRITE, dbiOPENSHARED,
            password, 0, NULL, NULL, phDb));
   return rslt;
}
```

Example 4: Open a remote database with a different user name than what is specified on the alias.

This example uses the following input:

fDbiOpenDatabase4(&hDb, "IBPerfect", "USER1", "password1");

```
DBIResult fDbiOpenDatabase4(phDBIDb phDb, pCHAR alias, pCHAR UserName, pCHAR
  password)
{
   FLDDesc      UserNameDesc;

   memset(&UserNameDesc, 0, sizeof(UserNameDesc));
   UserNameDesc.iOffset = 0;
   UserNameDesc.iLen = (UINT16)(strlen(UserName) + 1);
   strcpy(UserNameDesc.szName, "USER NAME");
   return Chk(DbiOpenDatabase(alias, NULL, dbiREADWRITE, dbiOPENSHARED,
            password, 1, &UserNameDesc, (pBYTE)UserName, phDb));
```

```
}
```

Delphi Examples: DbiOpenDatabase

Example 1: Open a local database:

```
procedure fDbiOpenDatabase1(var hDb: hDbiDb; Alias: String);
begin
  Check(DbiOpenDatabase(PChar(Alias), nil, dbiReadWrite, dbiOpenShared,
    nil, 0, nil, nil, hdb));
end;
```

Example 2: Open a NULL database (for in-memory or temp tables)

Set the directory for the tables (temp tables only).

```
procedure fDbiOpenDatabase2(var hDb: hDbiDb; Directory: string);
begin
  Check(DbiOpenDatabase(nil, nil, dbiREADWRITE, dbiOPENSHARED,
    nil, 0, nil, nil, hDb));
Check(DbiSetDirectory(hDb, PChar(Directory)));
end;
```

Example 3: Open a remote database with alias and password.

```
procedure fDbiOpenDatabase3(var hDb: hDbiDb; Alias, Password: string);
begin
  Check(DbiOpenDatabase(PChar(Alias), nil, dbiREADWRITE, dbiOPENSHARED,
    PChar(Password), 0, nil, nil, hDb));
end;
```

Example 4: Open a database with a user name other than the one specified in the BDE configuration.

This example uses the following input:

```
  fDbiOpenDatabase3(hDb, 'IBLOCAL', 'sysdba', 'speedy');
```

The procedure is:

```
procedure fDbiOpenDatabase3(var hTmpDb: hDBIDb; Alias, UserName, Password:
  string);
var
  Options: FLDDesc;
begin
  FillChar(Options, sizeof(Options), #0);
  Options.iOffset := 0;
  Options.iLen := Length(UserName) + 1;
  StrPCopy(Options.szName, 'USER NAME');
  Check(DbiOpenDatabase(PChar(Alias), nil, dbiREADWRITE, dbiOPENSHARED,
    PChar(Password), 1, @Options, PChar(UserName), hTmpDb));
end;
```

DbiOpenDatabaseList {button C Examples,JI(`>example',`exdbiopendatabaselist')} {button Delphi Examples,JI(`>example',`dexdbiopendatabaselist')}

C syntax
```
DBIResult DBIFN DbiOpenDatabaseList (phCur);
```

Delphi syntax
```
function DbiOpenDatabaseList (var hCur: hDBICur): DBIResult stdcall;
```

Description
DbiOpenDatabaseList returns a cursor on a list of accessible databases (and all aliases) found in the configuration file.

Parameters
phCur Type: phDBIcur (Output)
Pointer to an in-memory table.

Usage
Accessible databases are those that are defined within the configuration file. The cursor should be closed after information retrieval is complete. Each of the database records can be retrieved by using DbiGetNextRecord. DbiGetCursorProps can be used to allocate the proper record size. After the record is retrieved, it can be cast with the DBDesc type definition, and used like an DBDesc C language structure.

Completion state
A cursor on a list of accessible databases is returned. The cursor is positioned before the first record.

Note: The session property sesCFGMODE2 can affect the list returned by this function.

DbiResult return values
DBIERR_NONE The table was created successfully.

DBIERR_INVALIDHNDL *phCur* is NULL.

See also
DbiGetDatabaseDesc

C Examples: DbiOpenDatabaseList

An example for this function is under development and will be provided in an upcoming Help release.

Delphi Examples: DbiOpenDatabaseList

Return a list of of accessible databases and all aliases found in the configuration file.

This example uses the following input:

```
  fDbiOpenDatabaseList(DatabaseList);
```

The procedure is:

```
procedure fDbiOpenDatabaseList(DatabaseList: TStringList);
var
  TmpCursor: hDbiCur;
  Database: DBDesc;
  rslt: DbiResult;
begin
  DatabaseList.Clear;
  Check(DbiOpenDatabaseList(TmpCursor));
  repeat
    rslt:= DbiGetNextRecord(TmpCursor, dbiNOLOCK, @Database, nil);
    if (rslt <> DBIERR_EOF) then begin
      DatabaseList.Add(StrPas(Database.szName)
        + ' - ' + StrPas(Database.szPhyName)
        +  ' - ' + StrPas(Database.szDbType))
    end;
  until (rslt <> DBIERR_NONE);
  Check(DbiCloseCursor(TmpCursor));
end;
```

DbiOpenDriverList {button C Examples,JI(`>example',`exdbiopendriverlist')} {button Delphi Examples,JI(`>example',`dexdbiopendriverlist')}

C syntax
```
DBIResult DBIFN DbiOpenDriverList (phCur);
```

Delphi syntax
```
function DbiOpenDriverList (var hCur: hDBICur): DBIResult stdcal;
```

Description
DbiOpenDriverList creates a list of driver names available to the client application.

Parameters
phCur Type: phDBICur (Output)
Pointer to the cursor handle.

Usage
The list of drivers is obtained from the Registry and can be used as input to other functions. If no drivers were configured, an error is returned. The table contains only one CHAR field (of size DBINAME), the name of the driver. It does *not* contain all the information contained in DRVType structure.

Note: The session property sesCFGMODE2 can affect the list returned by this function.

DbiResult return values

DBIERR_NONE	The table containing a list of the available drivers was successfully created.
DBIERR_INVALIDHNDL	*phCur* is NULL.
DBIERR_NOCONFIGFILE	No configuration file was available at initialization time.
DBIERR_OBJNOTFOUND	No drivers were configured at initialization time.

See also
DbiGetDriverDesc

sesCFGMODE2

Replaces sesCFGMODE, letting you specify how aliases and drivers appear in the list returned by DbiOpenCfgInfoList, DbiOpenDatabaseList and DbiOpenDriverList. You can combine the following settings using a binary AND:

Setting	Description
cfgmNone	Use sesCFGMODE (for backward compatibility).
cfgmVirtual	Show only ODBC drivers and data sources, whether they're in the configuration file or not.
cfgmPersistent	Show only aliases and drivers saved in the configuration file.
cfgmSession	Show only aliases and drivers added to the current session.
cfgmAll	Show all aliases and drivers (shortcut for combining all settings).

For example, set sesCFGMODE2 to `cfgmVirtual & cfgmSession` to display drivers and aliases added to the current session in addition to ODBC drivers, data sources.

C Examples: DbiOpenDriverList

An example for this function is under development and will be provided in an upcoming Help release.

Delphi Examples: DbiOpenDriverList

Return a list of driver names available to the client application.
This example uses the following input:

```
  fDbiOpenDriverList(DriverList);
```

The procedure is:

```
procedure fDbiOpenDriverList(var DriverList: TStringList);
var
  TmpCursor: hdbicur;
  Driver: DRVType;
  rslt: dbiResult;
begin
  Check(DbiOpenDriverList(TmpCursor));
  DriverList.Clear;
  repeat
    rslt:= DbiGetNextRecord(TmpCursor, dbiNOLOCK, @Driver, nil);
    if (rslt <> DBIERR_EOF) then begin
      DriverList.Add(StrPas(Driver.szType))
    end;
  until rslt <> DBIERR_NONE;
  Check(DbiCloseCursor(TmpCursor));
end;
```

DbiOpenFamilyList {button C Examples,JI(`>example',`exdbiopenfamilylist')} {button Delphi Examples,JI(`>example',`dexdbiopenfamilylist')}

C syntax

```
DBIResult DBIFN DbiOpenFamilyList (hDb, pszTableName, [pszDriverType],
  phFmlCur);
```

Delphi syntax

```
function DbiOpenFamilyList (hDb: hDBIDb; pszTableName: PChar; pszDriverType:
  PChar; var hFmlCur: hDBICur): DBIResult stdcall;
```

Description

DbiOpenFamilyList creates a table listing the family members associated with a specified table. Each of the family records can be retrieved by using DbiGetNextRecord. DbiGetCursorProps can be used to allocate the proper record size. After the record is retrieved, it can be cast with the FMLDesc type definition, and used like an FMLDesc C language structure.

Parameters

hDb Type: hDBIDb (Input)
Specifies the database handle.

pszTableName Type: pCHAR (Input)
Pointer to the table name. If *pszTableName* is a fully qualified name of a table, the *pszTableType* parameter need not be specified. If the path is not included, the path name is taken from the current directory of the database associated with *hDb*.

pszDriverType Type: pCHAR (Input)
Pointer to the table type. Optional. This parameter is required if *pszTableName* has no extension. *pszDriverType* can be one of the following values: szDBASE, szMSACCESS, or szPARADOX.

phFmlCur Type: phDBICur (Output)
Pointer to the family list table.

Usage

Family members include default members, as specified by the driver, and registered family members.

dBASE or FoxPro: For dBASE or FoxPro tables, the table can include maintained index files (.MDX files), compressed index files (.CDX files), BLOBs (.DBT or .FPT files), and tables (.DBF files).

Paradox: For Paradox tables, the table can include index files (.PX, .X??, .Y?? files), BLOBs (.MB files), and validity check and referential integrity files (.VAL files).

SQL, Access: This function is not supported with SQL and Access tables. With SQL and Access databases, this function returns an empty table.

Prerequisites

The user must have full password rights to the table; that is, any required passwords to get prvFULL rights must have been added to the current session prior to calling this function.

DbiResult return values

DBIERR_NONE	The table of family members was successfully created.
DBIERR_INVALIDHNDL	The specified database handle is invalid or NULL, or phFmlCur is NULL.
DBIERR_INVALIDPARAM	The specified table name or the pointer to the table name is NULL.

DBIERR_NOSUCHTABLE	The specified table is invalid.
DBIERR_UNKNOWNDRIVER	The table type or the pointer to the table type is NULL, or the table type is invalid.

See also

DbiOpenFileList, DbiOpenFieldList, DbiOpenIndexList, DbiOpenRintList, DbiOpenSecurityList

C Examples: DbiOpenFamilyList

Create a table listing all family members associated with a table.

```
DBIResult fDbiOpenFamilyList(hDBIDb hDb, pCHAR TblName)
{
   DBIResult      rslt;
   hDBICur        hFmlCur;
   FMLDesc        FmlDesc;
   rslt = Chk(DbiOpenFamilyList(hDb, TblName, NULL, &hFmlCur));
   rslt = Chk(DbiGetNextRecord(hFmlCur, dbiNOLOCK,
                               (pBYTE)&FmlDesc, NULL));
   return rslt;
}
```

Delphi Examples: DbiOpenFamilyList

An example for this function is under development and will be provided in an upcoming Help release.

DbiOpenFieldList {button C Examples,JI(`>example',`exdbiopenfieldlist')} {button Delphi Examples,JI(`>example',`dexdbiopenfieldlist')}

C syntax

```
DBIResult DBIFN DbiOpenFieldList (hDb, pszTableName, [pszDriverType],
  bPhyTypes, phCur);
```

Delphi syntax

```
function DbiOpenFieldList (hDb: hDBIDb; pszTableName: PChar; pszDriverType:
  PChar; bPhyTypes: Bool; var hCur: hDBICur): DBIResult stdcall;
```

Description

DbiOpenFieldList creates a table listing of fields in a specified table and their descriptions.

Parameters

hDb Type: hDBIDb (Input)
Specifies the database handle.

pszTableName Type: pCHAR (Input)
Pointer to the table name. For Paradox, FoxPro, and dBASE, if *pszTableName* is a fully qualified name of a table, the *pszDriverType* parameter need not be specified. If the path is not included, the path name is taken from the current directory of the database associated with *hDb*.

For SQL databases, this parameter can be a fully qualified name that includes the owner name.

pszDriverType Type: pCHAR (Input)
Pointer to the table type. Optional. For Paradox, FoxPro, and dBASE tables, this parameter is required if *pszTableName* has no extension. This parameter is ignored if the database associated with *hDb* is a SQL database. *pszDriverType* can be one of the following values: szDBASE, szMSACCESS, or szPARADOX.

bPhyTypes Type: BOOL (Input)
Specifies whether physical or logical field types are returned. Physical types represent the data in its native state, specific to each driver. Logical types are the generic, derived BDE translations of the native data types. *bPhyTypes* can be set to TRUE or FALSE. TRUE indicates that native physical types are returned; FALSE indicates that BDE logical types are returned.

phCur Type: phDBICur (Output)
Pointer to the field list table.

Usage

This function retrieves field information from a closed table, as opposed to DbiGetFldDescs which uses an opened table. After the record is retrieved, it can be cast with the FLDDesc type definition, and used like a FLDDesc C language structure.

DbiResult return values

DBIERR_NONE	The cursor to the table was returned successfully.
DBIERR_INVALIDHNDL	The specified database handle is invalid or NULL.
DBIERR_INVALIDPARAM	The specified table name or the pointer to the table name is NULL.
DBIERR_UNKNOWNTBLTYPE	The specified driver type is not known.
DBIERR_NOSUCHTABLE	The specified table is invalid.

See also

DbiOpenFileList, DbiOpenTableList, DbiGetNextRecord, DbiGetPriorRecord,

DbiOpenFamilyList, DbiSetDirectory, DbiGetCursorProps, DbiGetFieldDescs

C Examples: DbiOpenFieldList

Get the field names for the specified table:

For standard tables, the extension must be specified. The Fields variable must have sufficient space allocated for function to use. This example uses the following input:

fDbiOpenFieldList(hDb, "customer.db", FieldNames);

```
DBIResult fDbiOpenFieldList(hDBIDb hTmpDb, pCHAR TblName, pCHAR Fields)
{
   FLDDesc       FldDesc;
   DBIResult     rslt;
   hDBICur       hTmpCur = 0;
   Fields[0] = '\0';
   rslt = Chk(DbiOpenFieldList(hTmpDb, TblName, NULL, FALSE, &hTmpCur));
   while (DbiGetNextRecord(hTmpCur, dbiNOLOCK, (pBYTE)&FldDesc, NULL) ==
  DBIERR_NONE)
   {
      strcat(Fields, FldDesc.szName);
      strcat(Fields, "  ");
   }
   if (hTmpCur != 0)
      DbiCloseCursor(&hTmpCur);
   return rslt;
}
```

Delphi Examples: DbiOpenFieldList

Display logical or physical field types for a table

This example uses the following input:

```
fDbiOpenFieldList(Table1, True, Memo1.Lines);
```

The procedure is:

```
procedure fDbiOpenFieldList(Table: TTable; Physical: Boolean; List:
  TStrings);

function BDEFieldIntToStr(FieldType: Word): string;
begin
  case FieldType of
    fldUNKNOWN: result := 'unknown';
    fldZSTRING: result := 'string';               { Null terminated string }
    fldDATE: result := 'date';                    { Date     (32 bit) }
    fldBLOB: result := 'BLOb';                    { Blob }
    fldBOOL: result := 'boolean';                 { Boolean  (16 bit) }
    fldINT16: result := 'integer';                { 16 bit signed number }
    fldINT32: result := 'long integer';           { 32 bit signed number }
    fldFLOAT: result := 'float';                  { 64 bit floating point }
    fldBCD: result := 'BCD';                      { BCD }
    fldBYTES: result := 'bytes';                  { Fixed number of bytes }
    fldTIME: result := 'time';                    { Time        (32 bit) }
    fldTIMESTAMP: result := 'timestamp';          { Time-stamp  (64 bit) }
    fldUINT16: result := 'unsigned int';          { Unsigned 16 bit
  integer }
    fldUINT32: result := 'unsigned long int';     { Unsigned 32 bit
  integer }
    fldFLOATIEEE: result := 'float IEEE';         { 80-bit IEEE float }
    fldVARBYTES: result := 'varbytes';            { Length prefixed var
  bytes }
    fldLOCKINFO: result := 'lockinfo';            { Look for LOCKINFO
  typedef }
    fldCURSOR: result := 'Oracle cursor';         { For Oracle Cursor type }

    { Paradox types (Physical) }
    fldPDXCHAR: result := 'alpha';                { Alpha    (string) }
    fldPDXNUM: result := 'numeric';               { Numeric }
    fldPDXMONEY: result := 'money';               { Money }
    fldPDXDATE: result := 'date';                 { Date }
    fldPDXSHORT: result := 'smallint';            { Short }
    fldPDXMEMO: result := 'Memo BLOb';            { Text Memo       (blob) }
    fldPDXBINARYBLOB: result := 'Binary BLOb';    { Binary data     (blob) }
    fldPDXFMTMEMO: result := 'formatted BLOb';    { Formatted text  (blob) }
    fldPDXOLEBLOB: result := 'OLE BLOb';          { OLE object      (blob) }
    fldPDXGRAPHIC: result := 'Graphic BLOb';      { Graphics object (blob) }
    fldPDXLONG: result := 'long integer';         { Long }
    fldPDXTIME: result := 'time';                 { Time }
    fldPDXDATETIME: result := 'date time';        { Time Stamp }
    fldPDXBOOL: result := 'boolean';              { Logical }
    fldPDXAUTOINC: result := 'auto increment';    { Auto increment (long) }
    fldPDXBYTES: result := 'bytes';               { Fixed number of bytes }
    fldPDXBCD: result := 'BCD';                   { BCD (32 digits) }
```

```
    { xBASE types (Physical) }
    fldDBCHAR: result := 'character';             { Char string }
    fldDBNUM: result := 'number';                 { Number }
    fldDBMEMO: result := 'Memo BLOb';             { Memo          (blob) }
    fldDBBOOL: result := 'logical';               { Logical }
    fldDBDATE: result := 'date';                  { Date }
    fldDBFLOAT: result := 'float';                { Float }
    fldDBLOCK: result := 'LOCKINFO';              { Logical type is LOCKINFO
  }
    fldDBOLEBLOB: result := 'OLE BLOb';           { OLE object    (blob) }
    fldDBBINARY: result := 'Binary BLOb';         { Binary data   (blob) }
    fldDBBYTES: result := 'bytes';                { Only for TEMPORARY
  tables }
    fldDBLONG: result := 'long integer';          { Long (Integer) }
    fldDBDATETIME: result := 'date time';         { Time Stamp }
    fldDBDOUBLE: result := 'double';              { Double }
    fldDBAUTOINC: result := 'aut increment';      { Auto increment (long) }
  else
    Result := 'not found';
  end;
end;

var
  hFieldCur: hDBICur;
  rslt: DBIResult;
  Field: FLDDesc;
begin
  List.Clear;
  Check(DbiOpenFieldList(Table.DBHandle, PChar(Table.TableName), nil,
    Physical, hFieldCur));
  repeat
    rslt := DbiGetNextRecord(hFieldCur, dbiNOLOCK, @Field, nil);
    if (rslt = DBIERR_NONE) then begin
      List.Add(Format('Field #%d) Name:%s Type:%s', [Field.iFldNum,
        Field.szName, BDEFieldIntToStr(Field.iFldType)]));
    end;
  until (rslt <> DBIERR_NONE);
end;
```

DbiOpenFieldTypesList {button C Examples,JI(`>example',`exdbiopenfieldtypeslist')} {button Delphi Examples,JI(`>example',`dexdbiopenfieldtypeslist')}

C syntax

```
DBIResult DBIFN DbiOpenFieldTypesList (pszDriverType, [pszTblType], phCur);
```

Delphi syntax

```
function DbiOpenFieldTypesList (pszDriverType: PChar; pszTblType: PChar;
  hCur: hDBICur): DBIResult stdcall;
```

Description

DbiOpenFieldTypesList creates a table containing a list of field types supported by the table type for the driver type.

Parameters

pszDriverType Type: pCHAR (Input)
Pointer to the driver type. Driver type must be found in the Registry.

pszTblType Type: pCHAR (Input)
Pointer to the table type. Use DbiOpenTableTypesList to retrieve table type information. If *pszTblType* is not specified, the default table type is used. Optional.

phCur Type: phDbiCur (Output)
Pointer to the cursor handle.

Usage

This function can be used to determine the legal field types, sizes, and other field-level attributes for a particular driver and table type. This allows configurable table creation UIs and allows for validation of field types (FLDType) without creating a table.

DbiResult return values

DBIERR_NONE	The table with the list of field types was created successfully.
DBIERR_UNKNOWNDRIVER	Invalid *pszDriverType* passed as input.

See also

DbiGetFieldTypeDesc

C Examples: DbiOpenFieldTypesList

Get the field types for the specified driver.

The Types variable must have sufficient space allocated for function to use. This example uses the following input:

fDbiOpenFieldTypesList(szDBASE, Buffer);

```
DBIResult fDbiOpenFieldTypesList(pCHAR Driver, pCHAR Types)
{
   DBIResult      rslt;
   hDBICur        hTmpCur = 0;
   FLDType        FldType;
   Types[0] = '\0';
   rslt = Chk(DbiOpenFieldTypesList(Driver, NULL, &hTmpCur));
   while (DbiGetNextRecord(hTmpCur, dbiNOLOCK, (pBYTE)&FldType, NULL) ==
  DBIERR_NONE)
   {
      strcat(Types, FldType.szName);
      strcat(Types, "  ");
   }
   if (hTmpCur != 0)
      DbiCloseCursor(&hTmpCur);
   return rslt;
}
```

Delphi Examples: DbiOpenFieldTypesList

Obtain field types for the driver and append the information to the TStringList passed in.

This example uses the following input:

```
  fDbiOpenFieldTypesList(TmpList, szParadox, 'PDOX 7.0');
```

The procedure is:

```
procedure fDbiOpenFieldTypesList(FieldTypeList: TStringList; DrvType,
  TblType: string);
var
  TmpCursor: hDbiCur;
  FieldType: FLDType;
  result: dbiResult;
begin
  Check(DbiOpenFieldTypesList(PChar(DrvType), PChar(TblType), TmpCursor));
  FieldTypeList.Clear;
  repeat
    result:= DbiGetNextRecord(TmpCursor, dbiNOLOCK, @FieldType, nil);
    if (result <> DBIERR_EOF) then begin
      FieldTypeList.Add('Field Name: ' + FieldType.szName);
    end;
  until (Result <> DBIERR_NONE);
  Check(DbiCloseCursor(TmpCursor));
end;
```

DbiOpenFieldXlt {button C Examples,JI(`>example',`exdbiopenfieldxlt')} {button Delphi Examples,JI(`>example',`dexdbiopenfieldxlt')}

C syntax

```
DBIResult DBIFN DbiOpenFieldXlt (pszSrcDriverType, pszSrcLangDrv, pfldSrc,
  pszDesDriverType, pszDstLangDrv, pfldDest, pbDataLoss, phXlt);
```

Delphi syntax

```
function DbiOpenFieldXlt (pszSrcTblType: PChar; pszSrcLangDrv: PChar;
  pfldSrc: pFLDDesc; pszDestTblType : PChar; pszDstLangDrv: PChar; pfldDest:
  pFLDDesc; var bDataLoss: Bool; var hXlt: hDBIXlt): DBIResult stdcall;
```

Description

DbiOpenFieldXlt builds a field translation object that can be used to translate a logical or physical field type into any other compatible logical or physical field type.

Parameters

pszSrcDriverType Type: pCHAR (Input)
Pointer to the source driver type. Set to NULL for logical.

pszSrcLangDrv Type: pCHAR (Input)
Pointer to the language driver name of the source. Set to NULL if no character set transliteration is desired. Ignored if both source and destination are not character types.

pfldSrc Type: pFLDDesc (Input)
Pointer to the source field descriptor.

pszDesDriverType Type: pCHAR (Input)
Pointer to the destination driver type. Set to NULL for logical.

pszDstLangDrv Type: pCHAR (Input)
Pointer to the language driver name of the destination. Set to NULL if no character set transliteration is desired. Ignored if both source and destination are not character types.

pfldDest Type: pFLDDesc (Input)
Pointer to the destination field descriptor.

pbDataLoss Type: pBOOL (Output)
Pointer to a client variable used to indicate both the possibility of data loss and actual data loss for each field translated when DbiTranslateField is called. If NULL, no data loss detection is done.

phXlt Type: phDBIXlt (Output)
Pointer to the translation object handle.

Usage

This function used in conjunction with DbiTranslateField allows clients to convert any logical or physical field data to any compatible logical or physical field data. The client supplies a pair of logical or physical field descriptors. These descriptors can be obtained from a call to DbiGetFieldDescs or DbiOpenFieldList.

If *pbDataLoss* is supplied, this client indicator variable is set to TRUE when the translation object is built if there is the potential for data loss when converting between the source and destination field types. For example, if the user requests a translation object to convert a dBASE character field to a BDE logical TIMESTAMP field, the data loss indicator is set to TRUE, because the character field may not contain a legal TIMESTAMP string according to the current session's DBIDATE and TIME conventions. Additionally, each time DbiTranslateField is called this client flag is set to TRUE if that particular field conversion caused data loss. If supplied, this client variable must remain addressable until the

translation object is closed with DbiCloseFieldXlt. For BLOB fields, this function provides a translation object that does nothing.

DbiResult return values

DBIERR_NONE	The translation object was successfully built.
DBIERR_NOTSUPPORTED	The requested field conversion is not considered legal.
DBIERR_INVALIDPARAM	One of the required pointers is NULL.
DBIERR_INVALID_FIELDDESC	One of the field descriptors is invalid.
DBIERR_NO_MEMORY	More space is needed in *pbDataLoss*.

See also

DbiTranslateField, DbiCloseFieldXlt

C Examples: DbiOpenFieldXlt

Build a field descriptor object for field translation (physical/logical):

```
DBIResult fDbiOpenFieldXlt(hDBICur hSrcCur, hDBICur hDesCur, phDBIXlt pXlt)
{
   DBIResult      rslt;
   pFLDDesc       psrcFldDesc;
   pFLDDesc       pdesFldDesc;
   CURProps       sCurProps;
   CURProps       dCurProps;
   BOOL           bDataLoss;

   Chk(DbiGetCursorProps(hSrcCur, &sCurProps));
   Chk(DbiGetCursorProps(hDesCur, &dCurProps));
   psrcFldDesc =(pFLDDesc)malloc((sizeof(FLDDesc))*(sCurProps.iFields));
   pdesFldDesc =(pFLDDesc)malloc((sizeof(FLDDesc))*(dCurProps.iFields));
   Chk(DbiGetFieldDescs(hSrcCur, psrcFldDesc));
   Chk(DbiGetFieldDescs(hDesCur, pdesFldDesc));
   rslt = Chk(DbiOpenFieldXlt(szPARADOX, "hebrew", &psrcFldDesc[1],
                              szDBASE, "hebrew", &pdesFldDesc[1],
                              &bDataLoss, pXlt));
   if(bDataLoss == TRUE)
      //Data Loss Possible
   return rslt;
}
```

Delphi Examples: DbiOpenFieldXlt

An example for this function is under development and will be provided in an upcoming Help release.

DbiOpenFileList {button C Examples,JI(`>example',`exdbiopenfilelist')} {button Delphi Examples,JI(`>example',`dexdbiopenfilelist')}

C syntax

```
DBIResult DBIFN DbiOpenFileList (hDb, [pszWild], phCur);
```

Delphi syntax

```
function DbiOpenFileList (hDb: hDBIDb; pszWild: PChar; var hCur: hDBICur):
  DBIResult stdcall;
```

Description

DbiOpenFileList opens a cursor on a list of files contained within the database.

Parameters

hDb Type: hDBIDb (Input)
Specifies the database handle.

pszWild Type: pCHAR (Input)
Pointer to the search string for retrieving a selective list of tables. Two wildcard characters can be used: the asterisk (*) and the question mark (?). The asterisk expands to any number of characters; the question mark expands to a single character.

phCur Type: phDBICur (Output)
Pointer to the file list table.

Usage

Each of the file records can be retrieved by using DbiGetNextRecord. DbiGetCursorProps can be used to allocate the proper record size. After the record is retrieved, it can be cast with the FILEDesc type definition, and used like an FILEDesc C language structure.

Access: This function is not supported.

Standard: DbiOpenFileList provides an efficient way to retrieve all the names of files in a database directory. This function returns a list of all files that match the wildcard criteria, if any.

SQL: This function returns information similar to that returned by DbiOpenTableList. Some fields, such as *szExt*, *bDir,* and *iSize*, are not applicable for SQL databases.

Synonyms: Many server vendors (including Oracle) provide objects called *synonyms*. Synonyms are alternate names for other objects, such as tables or views. SQL Links provides the option to include synonyms in the table lists returned from DbiOpenTableList and DbiOpenFileList. See SQL Links Guide

DbiResult return values

DBIERR_NONE	The cursor on the table was opened successfully.
DBIERR_INVALIDHNDL	The specified database handle is invalid or NULL, or *phCur* is NULL.
DBIERR_INVALIDPARAM	The specified record buffer is NULL.

See also

DbiOpenDatabase, DbiOpenTableList

pszWild

SQL: The search string has the following format: <*ownername*>.<*objectname*>. If no period is embedded in the wildcard string, it is assumed that *pszWild* represents a search for the object name only, and that the requested tables are for the current owner.

The following table provides examples of wildcard use for SQL databases:

Setting	Retrieves
NULL	All tables.
.	All tables for all owners. The default if NULL is passed.
*	All tables for the current owner.
*.EMP	All tables named EMP for all owners.
*CUST	All tables for the current owner ending in CUST.

Standard: For standard databases, search conventions are those used by DOS.

C Examples: DbiOpenFileList

An example for this function is under development and will be provided in an upcoming Help release.

Delphi Examples: DbiOpenFileList

Return a list of files contained within the database.

This example uses the following input:

```
fDbiOpenFileList(Database1.handle, '*.*', MyFileList);
```

The procedure is:

```
procedure fDbiOpenFileList(hDB: hDbiDb; Wild: string; var FileList:
  TStringList);
var
  TmpCursor: hdbicur;
  TmpFileDesc: FileDesc;
  rslt: dbiResult;
begin
  Check(DbiOpenFileList(hDB, PChar(Wild), TmpCursor));
  FileList.Clear;
  repeat
    rslt:= DbiGetNextRecord(TmpCursor, dbiNOLOCK, @TmpFileDesc, nil);
    if (rslt <> DBIERR_EOF) then begin
      FileList.Add(StrPas(TmpFileDesc.szfilename) + '.' +
        StrPas(TmpFileDesc.szext))
    end;
  until (rslt <> DBIERR_NONE);
  Check(DbiCloseCursor(TmpCursor));
end;
```

DbiOpenFunctionArgList {button C Examples,JI(`>example',`exdbiopenfunctionarglist')} {button Delphi Examples,JI(`>example',`dexdbiopenfunctionarglist')}

C syntax

```
DBIResult DBIFN DbiOpenFunctionArgList (hDb, pszFuncName, uOverload, phCur);
```

Delphi syntax

```
function DbiOpenFunctionArgList (hDb: hDBIDb; pszFuncName: PChar; uOverload:
  Word; var hCur: hDBICur): DBIResult stdcall;
```

Description

DbiOpenFunctionArgList opens a cursor to a schema table for the data source function defined by *pszFunctionName* for the driver associated with the *hDb*. Record description is of type DBIFUNCArgDesc.

Parameters

hDb Type: hDBIDb (Input)
Specifies the universal database handle.

pszFuncName Type: pCHAR (Input)
Name of data source function.

uOverload Type: UINT16 (Input)
Overload number, used with functions that take different sets of arguments.

phCur Type: phDBICur (Output)
Specifies returned cursor on the schema table "Arguments"

Usage

Use DbiOpenFunctionArgList to retrieve the arguments to functions supported by the data source. This information can be useful for clients developing a visual query builder.

When you call DbiOpenFunctionList, you get the number of overloads it can have. By passing the corresponding overload number (*uOverload*) to DbiOpenFunctionArgList you get the list of arguments that function takes with that particular overload number.

```
typedef struct {
      DBINAME         szName;           // Function name
      CHAR            szDesc[255];      // Short description
      UINT16          uOverload;        // Number of function overloads
      DBISTDFuncs     eStdFn;           // Corresponding to DBI standard
  function
    } DBIFUNCDesc;
typedef DBIFUNCDesc far *pDBIFUNCDesc;

typedef struct {
      UINT16          uArgNum;       // Argument position num; 0 for fn
  return
      UINT16          uFldType;      // Field type
      UINT16          uSubType;      // Field subtype (if applicable)
      UNIT16          ufuncFlags;    // Functions Flags
    } DBIFUNCArgDesc;
typedef DBIFUNCArgDesc far *pDBIFUNCArgDesc;
```

Prerequisites

DbiInit must be called, and *hDB* must reference a SQL data source.

Completion state

Returns a cursor with the function argument data, which must be closed by using DbiCloseCursor. If *pszFunctionName* is not a valid function associated for the driver asssociated with *hDb*, the cursor points to an empty table. It returns a 'not applicable' error if the *hDb* is associated with a local table.

DbiResult return values

DBIERR_NONE	The cursor on the table was opened successfully.
DBIERR_INVALIDHNDL	The specified database handle is invalid or NULL, or *phCur* is NULL.

See also

DbiOpenFunctionList

C Examples: DbiOpenFunctionArgList

An example for this function is under development and will be provided in an upcoming Help release.

Delphi Examples: DbiOpenFunctionArgList

An example for this function is under development and will be provided in an upcoming Help release.

DbiOpenFunctionList {button C Examples,JI(`>example',`exdbiopenfunctionlist')} {button Delphi Examples,JI(`>example',`dexdbiopenfunctionlist')}

C syntax

```
DBIResult DBIFN DbiOpenFunctionList (hDb, eUserDefined, phCur);
```

Delphi syntax

```
function DbiOpenFunctionList (hDb: hDBIDb; eoptBits: DBIFUNCOpts; var hCur:
  hDBICur): DBIResult stdcall;
```

Description

DbiOpenFunctionList opens a cursor to a schema table containing a list of all the functions for the driver associated with *hDb*. Record description is of type DBIFUNCDesc. For SQL data source only.

Parameters

hDb Type: hDBIDb (Input)
Specifies the universal database handle.

eUserDefined Type: DBIFUNCOpts (Input)
Set to fnListINCL_USER_DEF (the only setting included in the enumeration) to include user-defined functions. For InterBase only.

phCur Type: phDBICur (Output)
Specifies returned cursor on functions

Usage

Use DbiOpenFunctionList to retrieve the list of functions supported by the data source. This information can be useful for clients developing a visual query builder.

Prerequisites

DbiInit must be called, and *hDb* must reference a SQL data source.

Completion state

Returns a cursor to a schema table that lists all functions supported by the database. When building a SQL query this cursor must be closed with a call to DbiCloseCursor.

DbiResult return values

DBIERR_NOTSUPPORTED DbiOpenFunctionList returns an error if *hDb* is associated with a local table.

See also

DbiOpenFunctionArgList

C Examples: DbiOpenFunctionList

An example for this function is under development and will be provided in an upcoming Help release.

Delphi Examples: DbiOpenFunctionList

An example for this function is under development and will be provided in an upcoming Help release.

DbiOpenIndex {button C Examples,JI(`>example',`exdbiopenindex')} {button Delphi Examples,JI(`>example',`dexdbiopenindex')}

C syntax

```
DBIResult DBIFN DbiOpenIndex (hCursor, pszIndexName, iIndexId);
```

Delphi syntax

```
function DbiOpenIndex (hCursor: hDBICur; pszIndexName: PChar; iIndexId:
  Word): DBIResult stdcall;
```

Description

DbiOpenIndex opens the specified index or indexes for the table associated with the cursor.

Parameters

hCursor Type: hDBICur (Input)
Specifies the cursor handle.

pszIndexName Type: pCHAR (Input)
Pointer to the index name.

iIndexId Type: UINT16 (Input)
Specifies the index number. Used only with Paradox and Access tables.

Usage

dBASE and FoxPro: This function is used to open non-production dBASE and FoxPro indexes. The open index is maintained, but only in the context of this cursor. That is, only updates applied during the use of this cursor maintain the index. If the index is an .MDX or .CDX index, all tags in that index are opened and maintained.

Paradox: This function can be used only to verify that the specified index exists; it does not open the index. If the index does not exist, an error is returned. With Paradox tables, indexes are automatically opened when the table is opened.

Prerequisites

A valid cursor must be obtained, and the index must exist.

Completion state

DbiOpenIndex does not alter the current record order of the result set or the currency of the cursor. To change the current index order, use DbiSwitchToIndex.

DbiResult return values

DBIERR_NONE	The index was successfully opened on a dBASE table; the index exists on a Paradox table.
DBIERR_INVALIDHNDL	The specified handle is invalid or NULL.
DBIERR_INDEXOPEN	The index is already opened, either implicitly or explicitly.
DBIERR_NOSUCHINDEX	No such index exists for the table.

See also

DbiAddIndex, DbiCloseIndex, DbiSwitchToIndex

C Examples: DbiOpenIndex

An example for this function is under development and will be provided in an upcoming Help release.

Delphi Examples: DbiOpenIndex

An example for this function is under development and will be provided in an upcoming Help release.

DbiOpenIndexList {button C Examples,JI(`>example',`exdbiopenindexlist')} {button Delphi Examples,JI(`>example',`dexdbiopenindexlist')}

C syntax

```
DBIResult DBIFN DbiOpenIndexList (hDb, pszTableName, [pszDriverType],
  phCur);
```

Delphi syntax

```
function DbiOpenIndexList (hDb: hDBIDb; pszTableName: PChar; pszDriverType:
  PChar; var hCur: hDBICur): DBIResult stdcall;
```

Description

DbiOpenIndexList opens a cursor on a table listing the indexes on a specified table, along with their descriptions.

Parameters

hDb Type: hDBIDb (Input)
Specifies the database handle.

pszTableName Type: pCHAR (Input)
Pointer to the table name for which indexes are to be listed. For Paradox, FoxPro, and dBASE, if *pszTableName* is a fully qualified name of a table, the *pszDriverType* parameter need not be specified. If the path is not included, the path name is taken from the current directory of the database associated with *hDb*.

For SQL databases, this parameter can be a fully qualified name that includes the owner name.

pszDriverType Type: pCHAR (Input)
Pointer to the driver type. Optional. For Paradox, FoxPro, and dBASE tables, this parameter is required if *pszTableName* has no extension. This parameter is ignored if the database associated with *hDb* is a SQL database. *pszTableType* can be one of the following values: szDBASE, szMSACCESS, or szPARADOX.

phCur Type: phDBICur (Output)
Pointer to the cursor handle.

Usage

If there are no indexes, a cursor to an empty table is returned.

Oracle: For performance reasons, bPrimary is not set in the FLDDesc structure. To determine if a primary index exists on a table, use DbiSetProp with the curGETEXTENDEDINFO property before calling DbiGetIndexDescs.

Completion state

Each of the index description records can be retrieved using DbiGetNextRecord. DbiGetCursorProps can be used to allocate the proper record size. After the record is retrieved, it can be cast with the IDXDesc type definition, and used like an IDXDesc C language structure. This function retrieves index information from a closed table, as opposed to DbiGetIndexDescs and DbiGetIndexDesc that use an open table.

DbiResult return values

DBIERR_NONE	The table listing indexes for the table has been created.
DBIERR_INVALIDHNDL	The specified database handle is invalid or NULL, or *phCur* is NULL.
DBIERR_INVALIDPARAM	The specified table name or the pointer to the table name is NULL.
DBIERR_NOSUCHTABLE	The specified table name is invalid.
DBIERR_UNKNOWNTBLTYPE	The specified driver type is invalid.

See also
DbiGetNextRecord, DbiGetCursorProps, DbiGetIndexDesc, DbiGetIndexDescs

C Examples: DbiOpenIndexList

An example for this function is under development and will be provided in an upcoming Help release.

Delphi Examples: DbiOpenIndexList

Return a list of indexes on a specific table.

This example uses the following input:

```
fDbiOpenIndexList(DBASEANIMALS, IndexList);
```

The procedure is:

```
procedure fDbiOpenIndexList(Tbl: TTable; var IndexList: TStringList);
var
  TmpCursor: hdbicur;
  rslt: dbiResult;
  IndexDesc: IDXDesc;
begin
  Check(DbiOpenIndexList(Tbl.dbhandle, PChar(Tbl.TableName), nil,
  TmpCursor));
  IndexList.Clear;
  repeat
    rslt:= DbiGetNextRecord(TmpCursor, dbiNOLOCK, @IndexDesc, nil);
    if (rslt <> DBIERR_EOF) then begin
      IndexList.Add(StrPas(IndexDesc.szName))
    end;
  until (rslt <> DBIERR_NONE);
  Check(DbiCloseCursor(TmpCursor));
end;
```

DbiOpenIndexTypesList {button C Examples,JI(`>example',`exdbiopenindextypeslist')} {button Delphi Examples,JI(`>example',`dexdbiopenindextypeslist')}

C syntax

```
DBIResult DBIFN DbiOpenIndexTypesList (pszDriverType, phCur);
```

Delphi syntax

```
function DbiOpenIndexTypesList (pszDriverType: PChar; var hCur: hDBICur):
  DBIResult stdcall;
```

Description

DbiOpenIndexTypesList creates a table containing a list of all supported index types for the driver type.

Parameters

pszDriverType Type: pCHAR (Input)
Pointer to the driver type.

phCur Type: phDBICur (Output)
Pointer to the cursor handle.

Completion state

Each of the index type description records can be retrieved using DbiGetNextRecord. DbiGetCursorProps can be used to allocate the proper record size. After the record is retrieved, it can be cast with the IDXType type definition, and used like an IDXType C language structure.

DbiResult return values

DBIERR_NONE	The list of all supported index types was returned successfully.
DBIERR_UNKNOWNTBLTYPE	The specified driver type is unknown.
DBIERR_INVALIDHNDL	The specified handle is invalid.

See also

DbiGetIndexDesc

C Examples: DbiOpenIndexTypesList

An example for this function is under development and will be provided in an upcoming Help release.

Delphi Examples: DbiOpenIndexTypesList

An example for this function is under development and will be provided in an upcoming Help release.

DbiOpenLdList {button C Examples,JI(`>example',`exdbiopenldlist')} {button Delphi Examples,JI(`>example',`dexdbiopenldlist')}

C syntax
```
DBIResult DBIFN DbiOpenLdList (phCur);
```

Delphi syntax
```
function DbiOpenLdList (var hCur: hDBICur): DBIResult stdcall;
```

Description
DbiOpenLdList creates a table containing a list of available language drivers.

Parameters
phCur Type: phDBICur (Output)
Pointer to the cursor handle.

Usage
Each of the language driver records can be retrieved by using DbiGetNextRecord. DbiGetCursorProps can be used to allocate the proper record size. After the record is retrieved, it can be cast with the LDDesc type definition, and used like an LDDesc C language structure.

DbiResult return values
DBIERR_NONE The list of available language drivers was returned successfully.

DBIERR_INVALIDHNDL *phCur* is NULL.

C Examples: DbiOpenLdList

An example for this function is under development and will be provided in an upcoming Help release.

Delphi Examples: DbiOpenLdList

Example 1: Return all language driver information for the system

This example uses the following input:

```
  GetLdList(Memo1.Lines);
```

The function is defined as follows:

```
procedure GetLdList(Lines: TStrings);
var
  hCur: hDBICur;
  LD: LDDesc;
begin
  // get a cursor to the in-mem table containing language driver
  information...
  Check(DbiOpenLdList(hCur));
  try
    while (DbiGetNextRecord(hCur, dbiNOLOCK, @LD, nil) = DBIERR_NONE) do
  begin
      // add the name, code page, and description to the result...
      Lines.Add('Name: ' + LD.szName + '  Code Page: ' +
  IntToStr(LD.iCodePage));
      Lines.Add('    Description: ' + LD.szDesc);
    end;
  finally
    Check(DbiCloseCursor(hCur));
  end;
end;
```

DbiOpenLockList {button C Examples,JI(`>example',`exdbiopenlocklist')} {button Delphi Examples,JI(`>example',`dexdbiopenlocklist')}

C syntax

```
DBIResult DBIFN DbiOpenLockList (hCursor, bAllUsers, bAllLockTypes,
  phLocks);
```

Delphi syntax

```
function DbiOpenLockList (hCursor: hDBICur; bAllUsers: Bool; bAllLockTypes:
  Bool; var hLocks: hDBICur): DBIResult stdcall;
```

Description

DbiOpenLockList creates an in-memory table containing a list of locks acquired on the table associated with *hCursor*.

Parameters

hCursor Type: hDBICur (Input)
Specifies the cursor handle.

bAllUsers Type: BOOL (Input)
Specifies whether to list locks acquired in the current session only, or to list locks acquired by all sessions. For Paradox tables, *bAllUsers* can be either TRUE or FALSE. If *bAllUsers* is set to TRUE, users for all sessions are listed; if it is set to FALSE, only users for the current session are listed. For dBASE, FoxPro, and SQL tables, *bAllUsers* must be set to FALSE. For dBASE and FoxPro, only users for the current session are listed. For SQL, only locks for the current database connection are listed. Access does not support *bAllUsers*.

bAllLockTypes Type: BOOL (Input)
Specifies whether to include all locks of all types, or record locks only. If set to FALSE, only record locks are listed. If set to TRUE, locks of all types are listed.

phLocks Type: phDBICur (Output)
Pointer to the cursor handle of a table containting the list of locks.

Usage

Retrieve each table lock by using DbiGetNextRecord. After the record is retrieved, it can be cast with the LOCKDesc type definition.

Paradox: For Paradox tables, the locks on the table are returned, including those placed by the current session and those placed by other users, depending on the value of *bAllUsers*.

dBASE or FoxPro: For dBASE and FoxPro tables, only the locks placed by the current session are returned.

SQL: For SQL tables, only the locks placed by the current database connection are returned.

Prerequisites

A valid cursor handle must be obtained on a base table; this function is not applicable to query cursors or in-memory or temporary table cursors.

Completion state

The cursor is returned in *phLocks*. Lock types returned can include both table and record locks or only record locks, as specified in *bAllLockTypes*.

DbiResult return values

DBIERR_NONE	The requested lock list was returned successfully.
DBIERR_INVALIDHNDL	The specified cursor handle is invalid or NULL, or phLocks is NULL.

See also
DbiOpenTable, DbiAcqTableLock, DbiAcqPersistTableLock

C Examples: DbiOpenLockList

Retrieve all users on a particular table:

Note: pLockInfo must be large enough to hold data. This example using the following input:

```
char     Buffer[500];
fDbiOpenLockList(hCur, Buffer);
```

```
DBIResult fDbiOpenLockList(hDBICur hTmpCur, pCHAR pLockInfo)
{
   DBIResult   rslt;
   LOCKDesc    LDesc;
   hDBICur     hLockCur = 0;

   strcpy(pLockInfo, "\0");
   rslt = Chk(DbiOpenLockList(hTmpCur, TRUE, TRUE, &hLockCur));
   if (rslt != DBIERR_NONE)
      return rslt;
   while (rslt == DBIERR_NONE)
   {
      rslt = DbiGetNextRecord(hLockCur, dbiNOLOCK, (pBYTE)&LDesc, NULL);
      if (rslt == DBIERR_NONE)
         wsprintf(pLockInfo, "%s\r\nUSER: %s", pLockInfo, LDesc.szUserName);
   }
   if (rslt == DBIERR_EOF)
     rslt = DBIERR_NONE;
   else
     Chk(rslt);
   return rslt;
}
```

Delphi Examples: DbiOpenLockList

Return a list of locks acquired on a specific table.

This example uses the following input:

```
fDbiOpenLockList(Table1, LockList);
```

The procedure is:

```
procedure fDbiOpenLockList(Tbl: TTable; var LockList: TStringList);
var
  TmpCursor: hdbicur;
  Lock: LOCKDesc;
  rslt: dbiResult;
begin
  Check(DbiOpenLockList(Tbl.handle, True, True, TmpCursor));
  Check(DbiSetToBegin(TmpCursor));
  LockList.Clear;
  repeat
    rslt:= DbiGetNextRecord(TmpCursor, dbiNOLOCK, @Lock, nil);
    if (rslt <> DBIERR_EOF) then begin
      LockList.Add('Lock Type: ' + IntToStr(Lock.iType));
      LockList.Add('User Name: ' + StrPas(Lock.szUserName));
      LockList.Add('Net Session: ' + IntToStr (Lock.iNetSession));
      LockList.Add('Session: ' + IntToStr (Lock.iSession));
      LockList.Add('Record Number: ' + IntToStr (Lock.iRecNum));
    end;
  until (rslt <> DBIERR_NONE);
  Check(DbiCloseCursor(TmpCursor));
end;
```

DbiOpenRef

C syntax

```
DBIResult DBIFN DbiOpenRef (hDBICur hCursor, UINT16 iField, BOOL bReadOnly,
  BOOL bUniDirectional, phDBICur phCursor);
```

Delphi syntax

```
function DbiOpenRef(hCursor: hDBICur; iFieldNo: Word; bReadOnly: Bool;
  bUniDirectional: Bool; var hRefCursor: hDBICur): DBIResult; stdcall;
```

Description

DbiOpenRef creates a new cursor for the REF field.

Parameters

hCursor Type: hDBICur (Input)
Specifies the cursor handle of the parent cursor.

iField Type: UINT16 (Input)
Specifies the ordinal number of the REF field within the record buffer.

bReadOnly BOOL (Input)
If True, the REF field is opened in dbiREADONLY mode and cannot be modified.

bUniDirectional BOOL (Input)
TRUE if this cursor is unidirectional (SQL only).

phCursor phDBICur (Output)
Pointer to the cursor handle for the REF.

Prerequisites

Parent cursor must be opened by a DbiOpenTable or by executing a live query.

DbiResult return values

DBIERR_NONE	The REF was opened successfully.
DBIERR_NA	The iField specified is not a REF field.

See also

DbiCloseCursor, DbiDatabaseFlush

DbiOpenNestedTable

C syntax

```
DBIResult DBIFN DbiOpenNestedTable(hDBICur hCursor UINT16 iFieldNo BOOL
  bReadOnly BOOL bUniDirectional, phDBICur );
```

Delphi syntax

```
function DbiOpenNestedTable(hCursor: hDBICur; iFieldNo: Word; bReadOnly:
Bool; bUniDirectional: Bool; var hCursorNested: hDBICur): DBIResult;
stdcall;
```

Description

DbiOpenNestedTable creates a new cursor for the nested table field.

Parameters

hCursor Type: hDBICur (Input)
Specifies the cursor handle of the table.

iFieldNo Type: UNINT16 (Input)
Specifies the ordinal number of the nested table field within the record buffer.

bReadOnly BOOL (Input)
If True, the nested table is opened in dbiREADONLY mode and cannot be modified.

bUniDirectional BOOL (Input)
TRUE if this cursor is unidirectional (SQL only).

hCursorNested hDBICur (Output)
Cursor handle for the nested table.

Prerequisites

Parent cursor must be opened by a DbiOpenTable or by executing a live query.

DbiResult return values

DBIERR_NONE	The nested table was opened successfully.
DBIERR_NA	The iFieldNo specified is not a nested table column.

See also

DbiOpenRef

DbiDatabaseFlush

C syntax
```
DBIResult DBIFN DbiDatabaseFlush (hDBIDb hDb );
```

Delphi syntax
```
function DbiDatabaseFlush(hDb: hDBIDb): DBIResult;
```

Description
Flushes all record changes (insert/modify/delete) from the client to the Oracle server. Only meaningful for Oracle8.

Parameters
hDb Type: hDBIDb (Input)
Specifies the database handle.

Usage
Use when curAUTOFLUSHREF is set to False.

DbiResult return values
DBIERR_NONE The database specified by phDb was flushed successfully.

DBIERR_INVALIDHNDL The specified database handle is invalid or NULL.

See also
DbiOpenRef

DbiOpenRintList {button C Examples,JI(`>example',`exdbiopenrintlist')} {button Delphi Examples,JI(`>example',`dexdbiopenrintlist')}

C syntax

```
DBIResult DBIFN DbiOpenRintList (hDb, pszTableName, [pszDriverType],
  phChkCur);
```

Delphi syntax

```
function DbiOpenRintList (hDb: hDBIDb; pszTableName: PChar; pszDriverType:
  PChar; var hChkCur: hDBICur): DBIResult stdcall;
```

Description

DbiOpenRintList creates a table listing the referential integrity links for a specified table, along with their descriptions.

Parameters

hDb Type: hDBIDb (Input)
Specifies the database handle.

pszTableName Type: pCHAR (Input)
Pointer to the table name. If *pszTableName* is a fully qualified name of a table, the *pszDriverType* parameter need not be specified. If the path is not included, the path name is taken from the current directory of the database associated with *hDb*.

pszDriverType Type: pCHAR (Input)
Pointer to the driver type; required only if no extension is specified by *pszTableName*. Currently, the only valid type is szPARADOX.

phChkCur Type: phDBICur (Output)
Pointer to the cursor handle.

Usage

Retrieve each of the referential integrity records by using DbiGetNextRecord. Use DbiGetCursorProps to allocate the proper record size. After the record is retrieved, it can be cast with the RINTDesc type definition, and used like a RINTDesc C language structure.

Currently, this function is supported only with Paradox, dBASE, and Access tables.

DbiResult return values

DBIERR_NONE	The cursor to the table was successfully returned.
DBIERR_INVALIDPARAM	The specified table name or pointer to the table name is NULL.
DBIERR_INVALIDHNDL	The specified database handle is invalid or NULL, or phChkCur is NULL.
DBIERR_UNKNOWNTBLTYPE	The specified table type is invalid.
DBIERR_NOSUCHTABLE	The specified table does not exist.

See also

DbiOpenVchkList, DbiCreateTable, DbiGetRintDesc

C Examples: DbiOpenRintList

Creates a string containing all the names of referential integrity constraints on the specified table.

This example uses the following inputs:

fDbiOpenRintList(hDb, "ORDERS.DB", Buffer);

```
DBIResult fDbiOpenRintList(hDBIDb hDb, pCHAR TblName, pCHAR RIntList)
{
   DBIResult      rslt;
   hDBICur        hTmpCur = 0;
   RINTDesc       RIDesc;
   RIntList[0] = '\0';
   rslt = Chk(DbiOpenRintList(hDb, TblName, szPARADOX, &hTmpCur));
   if (rslt == DBIERR_NONE)
   {
      while (DbiGetNextRecord(hTmpCur, dbiNOLOCK, (pBYTE)&RIDesc, NULL) ==
  DBIERR_NONE)
      {
         strcat(RIntList, RIDesc.szRintName);
         strcat(RIntList, "  ");
      }
   }
   return rslt;
}
```

Delphi Examples: DbiOpenRintList

Example 1: Return all Referential Integrity information in a list for the specified table.

This example uses the following input:

```
  procedure GetRintDesc(Table1, Memo1.Lines)
```

```
procedure GetRintDesc(Table: TTable; Lines: TStrings);
var
  hCur: hDBICur;
  RIDesc: RINTDesc;
  rslt: DBIResult;
  B: Byte;
  Temp: string;
begin
  // Get a cursor to the RI information...
  Check(DbiOpenRIntList(Table.DBHandle, PChar(Table.TableName), nil, hCur));
  try
    Lines.Clear;
    Check(DbiSetToBegin(hCur));
    rslt := DBIERR_NONE;
    // While there are no errors, get RI information...
    while (rslt = DBIERR_NONE) do begin
      // Get the next RI record...
      rslt := DbiGetNextRecord(hCur, dbiNOLOCK, @RIDesc, nil);
      if (rslt <> DBIERR_EOF) then begin
        // Make sure nothing out of the ordinary happened...
        Check(rslt);
        // Display information...
        Lines.Add('RI Number: ' + IntToStr(RIDesc.iRintNum));
        Lines.Add('RI Name: ' + RIDesc.szRintName);
        case RIDesc.eType of
          rintMASTER: Lines.Add('RI Type: MASTER');
          rintDEPENDENT: Lines.Add('RI Type: DEPENDENT');
        else
          Lines.Add('RI Type: UNKNOWN');
        end;
        Lines.Add('RI Other Table Name: ' + RIDesc.szTblName);
        case RIDesc.eModOp of
          rintRESTRICT: Lines.Add('RI Modify Qualifier: RESTRICT');
          rintCASCADE: Lines.Add('RI Modify Qualifier: CASCADE');
        else
          Lines.Add('RI Modify Qualifier: UNKNOWN');
        end;
        case RIDesc.eDelOp of
          rintRESTRICT: Lines.Add('RI Delete Qualifier: RESTRICT');
          rintCASCADE: Lines.Add('RI Delete Qualifier: CASCADE');
        else
          Lines.Add('RI Delete Qualifier: UNKNOWN');
        end;
        Lines.Add('RI Fields in Linking Key: ' +
  IntToStr(RIDesc.iFldCount));
        Temp := '';
        for B := 0 to (RIDesc.iFldCount - 1) do
```

```
          Temp := Temp + IntToStr(RIDesc.aiThisTabFld[B]) + ', ';
        SetLength(Temp, Length(Temp) - 2);
        Lines.Add('RI Key Field Numbers in Table: ' + Temp);
        Temp := '';
        for B := 0 to RIDesc.iFldCount - 1 do
          Temp := Temp + IntToStr(RIDesc.aiOthTabFld[B]) + ', ';
        SetLength(Temp, Length(Temp) - 2);
        Lines.Add('RI Key Field Numbers in Other Table: ' + Temp);
        Lines.Add('');
      end;
    end;
  finally
    // All information was retrieved, close the in-memory table...
    Check(DbiCloseCursor(hCur));
  end;
end;
```

DbiOpenSecurityList {button C Examples,JI(`>example',`exdbiopensecuritylist')} {button Delphi Examples,JI(`>example',`dexdbiopensecuritylist')}

C syntax

```
DBIResult DBIFN DbiOpenSecurityList (hDb, pszTableName, [pszDriverType],
  phSecCur);
```

Delphi syntax

```
function DbiOpenSecurityList (hDb: hDBIDb; pszTableName: PChar;
  pszDriverType: PChar; var hSecCur: hDBICur): DBIResult stdcall;
```

Description

DbiOpenSecurityList creates a table listing record-level security information about a specified table.

Parameters

hDb Type: hDBIDb (Input)
Specifies the database handle.

pszTableName Type: pCHAR (Input)
Pointer to the table name. If *pszTableName* is a fully qualified name of a table, the *pszDriverType* parameter need not be specified. If the path is not included, the path name is taken from the current directory of the database associated with *hDb*.

pszDriverType Type: pCHAR (Input)
Pointer to the driver type. Required only if *pszTableName* did not specify an extension. Currently, the only valid driver type is szPARADOX.

phSecCur Type: phDBICur (Output)
Pointer to the cursor handle.

Usage

Table- and field-level security is applied with the functions DbiDoRestructure and DbiCreateTable. Currently, supported only with Paradox tables.

Completion state

Each of the security information records can be retrieved via DbiGetNextRecord. DbiGetCursorProps can be used to allocate the proper record size. After the record is retrieved, it can be cast with the SECDesc type definition, and used like an SECDesc C language structure.

DbiResult return values

DBIERR_NONE	The cursor was returned successfully.
DBIERR_INVALIDHNDL	The specified database handle is invalid or NULL, or phSecCur is NULL.
DBIERR_INVALIDPARAM	The specified table name or the pointer to the table name is NULL.
DBIERR_NOSUCHTABLE	The specified table name does not exist.
DBIERR_UNKNOWNTBLTYPE	The specified table type is invalid.

See also

DbiCreateTable, DbiDoRestructure

C Examples: DbiOpenSecurityList

Get security information for the specified table.

SecInfo must be allocated large enough to hold security information. This example uses the following input:

fDbiOpenSecurityList(hDb, "MYSECTBL.DB", Buffer);

```
DBIResult fDbiOpenSecurityList(hDBIDb hDb, pCHAR TblName, pCHAR SecInfo)
{
   DBIResult      rslt;
   SECDesc        SecDesc;
   hDBICur        hTmpCur;
   CHAR           Buffer[200], Priv[20];
   SecInfo[0] = '\0';
   rslt = Chk(DbiOpenSecurityList(hDb, TblName, szPARADOX, &hTmpCur));
   if (rslt == DBIERR_NONE)
   {
      while (DbiGetNextRecord(hTmpCur, dbiNOLOCK, (pBYTE)&SecDesc, NULL) ==
  DBIERR_NONE)
      {
         switch (SecDesc.eprvTable)
         {
            case prvNONE: strcpy(Priv, "None"); break;
            case prvREADONLY: strcpy(Priv, "Read Only"); break;
            case prvMODIFY: strcpy(Priv, "Modify"); break;
            case prvINSERT: strcpy(Priv, "Insert"); break;
            case prvINSDEL: strcpy(Priv, "Insert/Delete"); break;
            case prvFULL: strcpy(Priv, "Full"); break;
            case prvUNKNOWN: strcpy(Priv, "Unknown"); break;
         }
         wsprintf(Buffer, "\r\nID: %d, Privileges: %s, Password: %s",
               SecDesc.iSecNum, Priv, SecDesc.szPassword);
         strcat(SecInfo, Buffer);
      }
   }
   return rslt;
}
```

Delphi Examples: DbiOpenSecurityList

Return the record-level security (password) information about a specified Paradox table and append it to the TStringList passed in.

This example uses the following input:

```
fDbiOpenSecurityList(SecurityTable, SecurityList);
```

The procedure is:

```
procedure fDbiOpenSecurityList(Tbl: TTable; SecurityList: TStringList);
var
  TmpCursor: hdbicur;
  Security: SECDesc;
  result: dbiResult;
begin
  Check(DbiOpenSecurityList(Tbl.dbhandle, PChar(Tbl.TableName), nil,
  TmpCursor));
  repeat
    result:= DbiGetNextRecord(TmpCursor, dbiNOLOCK, @Security, nil);
    if (result <> DBIERR_EOF) then begin
      SecurityList.Add('Security Descriptor: ' +
  IntToStr(Security.iSecNum));
      case Security.eprvTable of
        prvNone:     SecurityList.Add('No privilege');
        prvREADONLY: SecurityList.Add('Read only Table or Field');
        prvMODIFY:   SecurityList.Add('Read and Modify fields (non-key)');
        prvINSERT:   SecurityList.Add('Insert + All of above');
        prvINSDEL:   SecurityList.Add('Delete + All of above');
        prvFULL:     SecurityList.Add('Full Writes');
        prvUNKNOWN:  SecurityList.Add('Unknown');
      end;
      SecurityList.Add('Family Rights: ' + IntToStr (Security.iFamRights));
      SecurityList.Add('Session: ' + Security.szPassword);
    end;
  until (Result <> DBIERR_NONE);
  Check(DbiCloseCursor(TmpCursor));
end;
```

DbiOpenSPList {button C Examples,JI(`>example',`exdbiopensplist')} {button Delphi Examples,JI(`>example',`dexdbiopensplist')}

C syntax

DBIResult DBIFN DbiOpenSPList (*hdb*, *bExtended*, *bSystem*, *pszQual*, *phCur*);

Delphi syntax

```
function DbiOpenSPList (hDb: hDBIDb; bExtended: Bool; bSystem: Bool;
  pszQual: PChar; var hCur: hDBICur): DBIResult stdcall;
```

Description

The function DbiOpenSPList creates a table containing information about the stored procedures associated with the database. Records in the table are described by SPDesc.

Parameters

hDb Type: hDBIDb (Input)
Specifies the database handle associated with the database where the stored procedure exists.

bExtended Type: BOOL (Input)
Not currently used.

bSystem Type: BOOL (Input)
True to include system procedures

pszQual Type: pCHAR (Input)
Must be null.

phCur Type: phDBICur (Output)
Pointer to the cursor handle

Completion state

The parameter *phCur* points to the returned cursor handle. The table contains information about all stored procedures in the database associated with the specified database handle. If the associated database is a standard database, only the stored procedures in the current directory of the database are listed in the table. The record description for the table is SPDesc.

DbiResult return values

DBIERR_NONE	The cursor to the table was successfully returned.
DBIERR_INVALIDHNDL	The specified database handle is invalid or NULL.
DBIERR_NOTSUPPORTED	The driver does not support stored procedures.

C Examples: DbiOpenSPList

An example for this function is under development and will be provided in an upcoming Help release.

Delphi Examples: DbiOpenSPList

An example for this function is under development and will be provided in an upcoming Help release.

DbiOpenSPParamList {button C Examples,JI(`>example',`exdbiopenspparamlist')} {button Delphi Examples,JI(`>example',`dexdbiopenspparamlist')}

C syntax

```
DBIResult DBIFN DbiOpenSPParamList (hdb, pszSPName, bPhyTypes, uOverload,
  phCur );
```

Delphi syntax

```
function DbiOpenSPParamList (hDb: hDBIDb; pszSPName: PChar; bPhyTypes: Bool;
  uOverload: Word; var hCur: hDBICur): DBIResult stdcall;
```

Description

The function DbiOpenSPParamList creates a table listing the parameters associated with a specified stored procedure. Records in the table are described by SPParamDesc.

Parameters

hDb Type: hDBIDb (Input)
Specifies the database handle associated with the database where the stored procedure exists.

pszSPName Type: pCHAR (Input)
Pointer to the stored procedure name.

bPhyTypes Type: BOOL (Input)
Specifies whether parameter field types are returned in physical or logical datatypes.

uOverload Type: UINT16 (Input)
Overload number. Not available for all drivers. This value is 0 unless the driver supports it and has overloaded functions. For an example, see uOverload

phCur Type: phDBICur (Output)
Pointer to the cursor handle.

Usage

Standard: Not Supported.

Access: The *SPParamDesc* structure fields *iUnits1*, *uOffset*, *uLen* and *uNullOffset* are not used. When *bPhySypes* is False, *uFldType* and *uSubType* are not used.

SQL: Supported.

Sybase: DbiOpenSPParamList returns the parameters, but *eParamType* is always equal to *paramUNKNOWN*.

Oracle: For full stored procedure support, your server must be a production Oracle7 server set up with the Procedural option. If it has not been set up properly, you might get the following error from DbiOpenSPParamList: "DBMS_DESCRIBE is not defined ...".

Completion state

Returns list of the parameters associated with a specified stored procedure. The record description for the table is *SPParamDesc*.

DbiResult return values

DBIERR_NONE	The cursor to the table was successfully returned.
DBIERR_INVALIDHNDL	The specified database handle is invalid or NULL.
DBIERR_NOTSUPPORTED	The driver does not support stored procedures.

See also

DbiOpenSPList

uOverload

The *uOverload* param in DbiOpenSPParamList allows specification of an overload number if the server supports overloading of procedure and function names. For example, using Oracle 7, you might have this package specification:

```
create package EMP_RECS as
     procedure get_sal_info (
            name      in      emp.ename%type,
            salary    out     emp.sal%type);
     procedure get_sal_info (
            ID_num    in      emp.empno%type,
            salary    out     emp.sal%type);
     function get_sal_info (
            name     emp.ename%type)   return emp.sal%type;
end EMP_RECS;
```

DbiOpenSPParamList with *uOverload*=1 would return the *name* and *salary* parameters for procedure 1. If *uOverload* = 2, then *ID_num* and *salary* would be returned.

If a procedure is not overloaded, then *uOverload* should be set to 0. Otherwise *uOverload* should be set to 1..n for *n* overloadings of the name.

C Examples: DbiOpenSPParamList

An example for this function is under development and will be provided in an upcoming Help release.

Delphi Examples: DbiOpenSPParamList

An example for this function is under development and will be provided in an upcoming Help release.

DbiOpenTable {button C Examples,JI(`>example',`exdbiopentable')} {button Delphi Examples,JI(`>example',`dexdbiopentable')}

C syntax

```
DBIResult DBIFN DbiOpenTable (hDb, pszTableName, [pszDriverType],
  pszIndexName, pszIndexTagName, iIndexId, eOpenMode, eShareMode, exltMode,
  [bUniDirectional], [pOptParams], phCursor);
```

Delphi syntax

```
function DbiOpenTable (hDb: hDBIDb; pszTableName: PChar; pszDriverType:
  PChar; pszIndexName: PChar; pszIndexTagName: PChar; iIndexId: Word;
  eOpenMode: DBIOpenMode; eShareMode: DBIShareMode; exltMode: XLTMode;
  bUniDirectional: Bool; pOptParams: Pointer; var hCursor: hDBICur):
  DBIResult stdcall;
```

Description

DbiOpenTable opens the given table for access and associates a cursor handle with the opened table.

Parameters

hDb Type: hDBIDb (Input)
Specifies the database handle associated with the database where the table exists.

pszTableName Type: pCHAR (Input)
Pointer to the table name. For Paradox, FoxPro, and dBASE, if *pszTableName* is a fully qualified name of a table, the *pszDriverType* parameter need not be specified. If the path is not included, the path name is taken from the current directory of the database associated with *hDb*.

For SQL databases, *pszTableName* can be a fully qualified name that includes the owner name, in the form *<owner>.<tablename>*.

If not specified, *<owner>* is supplied from the database handle. Extensions are not valid for SQL table names.

pszDriverType Type: pCHAR (Input)
Pointer to the driver type. Optional. *pszDriverType* can be one of the following values: szDBASE, szMSACCESS, szPARADOX, or szASCII (for importing or exporting data to/from text files; see the Usage section).

For Paradox, FoxPro, and dBASE tables, this parameter is required if *pszTableName* has no extension, or if the client application wants to overwrite the default file extension, including the situation where *pszTableName* is terminated with a period(.). If *pszTableName* does not supply the default extension, and *pszDriverType* is NULL, DbiOpenTable tries to open the table with the default file extension of all file-based drivers listed in the configuration file in the order that the drivers are listed.

This parameter is ignored if the database associated with *hDb* is a SQL or Access database.

pszIndexName Type: pCHAR (Input)
Pointer to the name of the index or pseudo-index to be used to order the records in the result set. Optional. For SQL tables, the index name does not have to be qualified with the owner for servers supporting naming conventions with owner qualification. The *pszIndexName* string is limited to 127 bytes in length.

pszIndexTagName Type: pCHAR (Input)
Pointer to the tag name of the index in a .MDX or .CDX file used to order the records in the result set. Optional; used for dBASE and FoxPro tables only. This parameter is ignored if the index given by *pszIndexName* is a .NDX index.

iIndexId Type: UINT16 (Input)
Specifies the index identifier, which is the number of the index to be used to order the records in the result set. Optional; used for Paradox, Access, and SQL tables only.

Paradox: For Paradox tables, the range for the index identifier is 1 to 511. This parameter is ignored if *pszIndexName* is specified.

SQL: For SQL tables, this field is used only to specify that the table should be opened with no default index. This is done by setting iIndexId to NODEFAULTINDEX and is useful when opening a table read-only to speed up record access time.

Access: For Access tables, the range for the index identifier is 1 to the number of valid indexes. This parameter is ignored if *pszIndexName* is specified.

eOpenMode Type: DBIOpenMode (Input)
Specifies the table open mode. If the mode is read-only, updates to the table are not permitted.

eShareMode Type: DBIShareMode (Input)
Specifies the table share mode, and determines whether other users or other cursors are able to open the table.

exltMode Type: XLTMode (Input)
Specifies the data translation mode.

bUniDirectional Type: BOOL (Input)
Specifies the scan mode of the cursor for SQL only.

pOptParams Type: pBYTE (Input)
Not currently used.

phCursor Type: phDBICur (Output)
Pointer to the cursor handle for the opened table.

Usage

Text: The DbiOpenTable call can be used to open a text file for import/export of data. The *pszDriverType* argument is used differently to indicate whether the fields in the text file are fixed length or delimited. The field separator and delimiter are passed through the *pszDriverType* argument.

dBASE: If no index is specified, the table is opened in physical order. If *pszIndexTagName* specifies an index tag, the table is opened with that tag active. The index name and the tag name are specified to open the index.

FoxPro: If no index is specified, the table is opened in physical order. If *pszIndexTagName* specifies an index tag, the table is opened with that tag active. The index name and the tag name are specified to open the index. To see if a cursor is referencing a FoxPro table, retrieve CURProps using DbiGetCursorProps.

Access: If a cursor is opened exclusively in the active session, no other cursors may be opened in that session.

Paradox: If all index parameters are NULL, the table is opened in primary key order, if a primary key exists. If a secondary key is specified, the table is opened in that key. Either *pszIndexName* or *iIndexId* can be used to specify a composite or non-composite secondary index. A single-field index that is case-insensitive is classified as a composite index. See also: szName

SQL: An index can be specified only in *pszIndexName*. The index name can be qualified or unqualified. SQL provides limited support for exclusive opens, depending on the level of server explicit lock support.

Pseudo-indexes: To describe a pseudo-index rather than an existing physical index, replace the *pszIndexName* parameter with a string composed of field names. The marker

character @ denotes the use of a pseudo-index. For example, "@Customer Number@Order Number" describes a pseudo-index on a key formed by concatenating the Customer Number field with the Order Number field.

Each field identifier in the pseudo-index name must be preceded by the @ character. This character is illegal in "true" index names. No new index is generated at the server; the behavior of the pseudo-index is simulated entirely by use of the proper ORDER BY clauses on the query populating the local BDE record cache.

Fields can be identified by field numbers as well as by field names. For example, the string "@2@3@11" describes a pseudo-index consisting of the second, third, and eleventh field of the table, concatenated to make up a single key.

Each of the component fields within a *pszIndexName* is assumed to be in ASCENDING order. Ordering is case-sensitive (unless case-sensitivity is not supported on the specific server). If the fields in the *pszIndexName* represent a real unique index on the server, the pseudo-index becomes unique; otherwise, it is non-unique.

Note: Access does not support pseudo-indexes.

Referential integrity descriptions: By default, DbiOpenTable does not retrieve referential integrity information (primary keys, foreign keys, and so on) when opening a table. This improves performance. To retrieve referential integrity information, use DbiSetProp to set curGETEXTENDEDINFO to True, then call DbiGetCursorProps.

Prerequisites

If the database is opened read-only, the table cannot be opened read-write.

Completion state

After the table has been successfully opened, the cursor is opened and positioned on the crack at the beginning of the file. A valid cursor is returned.

DbiResult return values

DBIERR_NONE	The table was successfully opened.
DBIERR_INVALIDFILENAME	The specified file name is not valid.
DBIERR_NOSUCHFILE	The specified file could not be found.
DBIERR_TABLEREADONLY	This table cannot be opened for read-write access.
DBIERR_NOTSUFFTABLERIGHTS	The client application does not have sufficient rights to open this table.
DBIERR_INVALIDINDEXNAME	The specified index name is invalid.
DBIERR_INVALIDHNDL	The specified database handle is invalid or NULL.
DBIERR_INVALIDPARAM	The specified table name or the pointer to the table name is NULL, or *phCursor* is NULL.
DBIERR_UNKNOWNTBLTYPE	The specified table type is invalid.
DBIERR_NOSUCHTABLE	The specified table name is invalid.
DBIERR_NOSUCHINDEX	The specified index is not available.
DBIERR_LOCKED	The table is locked by another user.
DBIERR_DIRBUSY	Invalid attempt to open a table in private directory (Paradox only).
DBIERR_OPENTBLLIMIT	The maximum number of tables is already opened.

See also

DbiCloseCursor

C Examples: DbiOpenTable

Open the specified table.

If the table is local (Paradox, FoxPro, Access, or dBASE), the table name must also have the extension. This function uses the following input:

fDbiOpenTable(hDb, "CUSTOMER.DB", &hCur, &CurProps);

```
DBIResult fDbiOpenTable(hDBIDb hTmpDb, pCHAR pszTableName, phDBICur phTmpCur,
  pCURProps pCurProps)
{
   DBIResult      rslt;
   rslt = Chk(DbiOpenTable (hTmpDb, pszTableName, NULL, NULL, NULL, 0,
  dbiREADWRITE,
                   dbiOPENSHARED, xltFIELD, TRUE, NULL, phTmpCur));
   if (rslt != DBIERR_NONE)
      return rslt;
   if (pCurProps != NULL)
      rslt = Chk(DbiGetCursorProps(*phTmpCur, pCurProps));
   return rslt;
}
```

Delphi Examples: DbiOpenTable

Open a table:

Delphi users should use the Open method associated with the TTable component rather than directly calling DbiOpenTable. This method is defined as:

```
procedure Open;
```

The Open method opens the dataset, putting it in Browse state.

```
Table1.Open;
```

pseudo-index

For SQL data sources, a current index can be defined as any group of fields from a specific table, whether or not a corresponding index exists on the server. BDE creates a pseudo-index by using one or more user-specified SQL fields to define the requested order.

You can specify the pseudo-index even if there is a real index matching the behavior of the pseudo-index. When specifying the pseudo-index, BDE behavior is the same as it would be if the physical index existed on the server. In particular, DbiSetRange and DbiGetRecordForKey are allowed on a pseudo-index. DbiSetToBegin, DbiGetNextRecord, and so on, walk through records in the order implied by a pseudo-index.

For information on implementing pseudo-indexes, see DbiOpenTable or DbiSwitchToIndex.

Database Open Mode

The following table shows the interaction between the database open mode and *eOpenMode*:

Database	***eOpenMode***	**Result**
Read-only	Read-only	Read-only
Read-only	Read-write	Error
Read-write	Read-only	Read-only
Read-write	Read-write	Read-write

Database Share Mode

For Paradox, FoxPro, Access, and dBASE tables, if *eShareMode* is set to dbiOPENEXCL, then only this session can open the table. If the table is already opened (shared or exclusive) by another session, an attempt to open the table exclusively results in an error. The following table shows the results of different combinations of the database share mode and *eShareMode*:

Database	***eShareMode***	**Result**
Exclusive	Exclusive	Exclusive
Exclusive	Share	Exclusive
Share	Exclusive	Exclusive
Share	Share	Share

Scan Mode

This parameter can be one of the following values:

***bUniDirectional* value**	**Scan mode of SQL table cursor**
TRUE	Unidirectional. The cursor can only be advanced forward.
FALSE	Bidirectional. The cursor can be advanced forward and backward.

DbiOpenTableList {button C Examples,JI(`>example',`exdbiopentablelist')} {button Delphi Examples,JI(`>example',`dexdbiopentablelist')}

C syntax

```
DBIResult DBIFN DbiOpenTableList (hDb, bExtended, bSystem, pszWild, phCur);
```

Delphi syntax

```
function DbiOpenTableList (hDb: hDBIDb; bExtended: Bool; bSystem: Bool;
  pszWild: PChar; var hCur: hDBICur): DBIResult stdcall;
```

Description

DbiOpenTableList creates a table with information about all the tables associated with the database.

Parameters

hDb Type: hDBIDb (Input)
Specifies the database handle.

bExtended Type: BOOL (Input)
The bExtended parameter specifies whether to return only the standard table information, or to return extended table information as well. (The default is standard information only).

bSystem Type: BOOL (Input)
The bSystem parameter specifies whether to include system tables or not. SQL only.

pszWild Type: pCHAR (Input)
Pointer to the search string for retrieving a selective list of tables. Two wildcard characters can be used: the asterisk (*) and the question mark (?). The asterisk expands to any number of characters; the question mark expands to a single character.

phCur Type: phDBICur (Output)
Pointer to the cursor handle.

Usage

The client application can request either standard or extended information for the table. The *bExtended* parameter must be set to TRUE to request extended information.

Standard: The table includes tables in the directory associated with *hDb*.

SQL: For SQL servers, *bSystem* must be set to TRUE to include system tables.

Synonyms: Many server vendors (including Oracle) provide objects called *synonyms*. Synonyms are alternate names for other objects, such as tables or views. SQL Links provides the option to include synonyms in the table lists returned from DbiOpenTableList and DbiOpenFileList. See SQL Links Guide

Completion state

phCur points to the returned cursor handle. The table contains information about all the tables in the database associated with the specified database handle. If the associated database is a standard database, only the tables in the current directory of the database are listed in the table. The record description for the table is TBLBaseDesc or TBLFullDesc.

DbiResult return values

DBIERR_NONE The cursor to the table was returned successfully.

DBIERR_INVALIDHNDL The specified database handle is invalid or NULL.

See also

DbiOpenCfgInfoList, DbiOpenDriverList, DbiOpenFieldTypesList, DbiOpenIndexTypesList, DbiOpenLdList, DbiOpenTableTypesList, DbiOpenUserList

C Examples: DbiOpenTableList

Return a string containing all tables meeting the search criteria (in WildCard).

This example uses the following input:

fDbiOpenTableList(hDb, &hCursor);

```
DBIResult fDbiOpenTableList(hDBIDb hTmpDb, pCHAR TblList, pCHAR WildCard)
{
   DBIResult      rslt;
   TBLBaseDesc    ListDesc;    // structure to hold information about the
  table list.
   hDBICur        hCur;
   CHAR           Buffer[DBIMAXTBLNAMELEN + 1];
   UINT16         Count = 1;
   TblList[0] = '\0';
   rslt = Chk(DbiOpenTableList(hTmpDb, FALSE, FALSE, WildCard, &hCur));
   if (rslt != DBIERR_NONE)
      return rslt;
   while ((DbiGetNextRecord(hCur, dbiNOLOCK, (pBYTE)&ListDesc, NULL)) !=
  DBIERR_EOF)
   {
      wsprintf(Buffer, "\r\nTable %d: %s", Count++, ListDesc.szName);
      strcat(TblList, Buffer);
   }
   return rslt;
}
```

Delphi Examples: DbiOpenTableList

Create a table with information about all tables associated with database.

This example retrieves all tables in the databse with the .DB extension and puts the tablenames into a string list object, such as the Lines property of a TMemo. The example uses the following input:

```
  fDbiOpenTableList(Table1.DBHandle, Memo1.Lines);
```

The procedure is:

```
procedure fDbiOpenTableList(hTmpDb: hDBIDb; TableList: TStrings);
var
  hCursor : hDBICur;
  ListDesc : TBLBaseDesc;
begin
  Check(DbiOpenTableList(hTmpDb, False, False, '*.DB', hCursor));
  TableList.Clear;
  while (DbiGetNextRecord(hCursor, dbiNOLOCK, @ListDesc, nil) = dbiErr_None)
  do
    TableList.Add(ListDesc.szName);
end;
```

bExtended

***bExtended* value**	**Type of table info returned**
TRUE	Extended
FALSE	Standard

bSystem

***bSystem* value**	**System table included?**
TRUE	Yes
FALSE	No

DbiOpenTableTypesList {button C Examples,JI(`>example',`exdbiopentabletypeslist')} {button Delphi Examples,JI(`>example',`dexdbiopentabletypeslist')}

C syntax

```
DBIResult DBIFN DbiOpenTableTypesList (pszDriverType, phCur);
```

Delphi syntax

```
function DbiOpenTableTypesList (pszDriverType: PChar; var hCur: hDBICur):
  DBIResult stdcall;
```

Description

DbiOpenTableTypesList creates a table listing table type names for the given driver.

Parameters

pszDriverType Type: pCHAR (Input)
Pointer to the driver type.

phCur Type: phDBICur (Output)
Pointer to the cursor handle.

Completion state

Each of the table type records can be retrieved via DbiGetNextRecord. DbiGetCursorProps can be used to allocate the proper record size. After the record is retrieved, it can be cast with the TBLType type definition, and used like a TBLType C language structure.

DbiResult return values

DBIERR_NONE	The list of table type names was returned successfully.
DBIERR_INVALIDHNDL	The specified handle is invalid.
DBIERR_DRIVERNOTLOADED	The driver was not initialized.

See also

DbiGetTableTypeDesc

C Examples: DbiOpenTableTypesList

Display in a MessageBox all table types supported by a driver:

The following input is used in this example:
fDbiOpenTableTypesList(szPARADOX);

```
DBIResult fDbiOpenTableTypesList(pCHAR Driver)
{
   DBIResult   rslt;
   hDBICur     hTypeCur = 0;
   TBLType     Tbltype;
   CHAR        Buffer[500] = {'\0'};

   rslt = Chk(DbiOpenTableTypesList(Driver, &hTypeCur));
   if (rslt != DBIERR_NONE)
      return rslt;
   while (DbiGetNextRecord(hTypeCur, dbiNOLOCK, (pBYTE)&Tbltype, NULL) ==
  DBIERR_NONE)
      wsprintf(Buffer, "%sTable Name: %s, Maximum Record Size: %d\n",
               Buffer, Tbltype.szName, Tbltype.iMaxRecSize);

   MessageBox(0, Buffer, Driver, MB_OK);
   return rslt;
}
```

Delphi Examples: DbiOpenTableTypesList

Display in a MessageBox all table types supported by a driver.
This example uses the following input:

```
  fDbiOpenTableTypesList(szDBASE);
```

The procedure is:

```
procedure fDbiOpenTableTypesList(Driver: string);
var
  hTypeCur: hDBICur;
  TblTypes: TBLType;
  BufStr: string;
begin
  hTypeCur:= nil;
  Check(DbiOpenTableTypesList(PChar(Driver), hTypeCur));
  while (DbiGetNextRecord(hTypeCur, dbiNOLOCK, @TblTypes, nil) =
  DBIERR_NONE) do
  begin
    BufStr:= format('Name: %s, TableLevel: %d',
  [Tbltypes.szName,Tbltypes.iTblLevel]);
    MessageBox(0, PChar(BufStr), PChar(Driver), MB_OK);
  end;
end;
```

DbiOpenUserList {button C Examples,JI(`>example',`exdbiopenuserlist')} {button Delphi Examples,JI(`>example',`dexdbiopenuserlist')}

C syntax
```
DBIResult DBIFN DbiOpenUserList (phUsers);
```

Delphi syntax
```
function DbiOpenUserList (var hUsers: hDBICur): DBIResult stdcall;
```

Description
DbiOpenUserList creates a table containing a list of users sharing the same network file.

Parameters
phUsers Type: phDBICur (Output)
Pointer to the cursor handle.

Usage
DbiOpenUserList is supported for Paradox only.

Completion state
Each of the user records can be retrieved using DbiGetNextRecord. DbiGetCursorProps can be used to allocate the proper record size. After the record is retrieved, it can be cast with the USERDesc type definition, and used like a USERDesc C language structure.

DbiResult return values
DBIERR_NONE The user list was returned successfully.

DBIERR_INVALIDHNDL phUsers is NULL.

C Examples: DbiOpenUserList

Get information on users using the current network file.

UserInfo must be allocated large enough to hold security information. This example uses the following input:

fDbiOpenUserList(Buffer);

```
DBIResult fDbiOpenUserList(pCHAR UserInfo)
{
   DBIResult      rslt;
   USERDesc       UserDesc;
   hDBICur       hTmpCur;
   CHAR           Buffer[500];
   UserInfo[0] = '\0';
   rslt = Chk(DbiOpenUserList(&hTmpCur));
   if (rslt == DBIERR_NONE)
   {
      while (DbiGetNextRecord(hTmpCur, dbiNOLOCK, (pBYTE)&UserDesc, NULL) ==
  DBIERR_NONE)
      {
         wsprintf(Buffer, "\r\nName: %s, Session: %d, Class: %d, SerialNum:
  %s",
            UserDesc.szUserName, UserDesc.iNetSession,
  UserDesc.iProductClass,
            UserDesc.szSerialNum);
         strcat(UserInfo, Buffer);
      }
   }
   return rslt;
}
```

Delphi Examples: DbiOpenUserList

Return a list of users sharing the same network file

The returned list is appended it to the string list object specified in the UserList parameter. This example uses the following input:

```
  fDbiOpenUserList(ListBox1.Items);
```

The procedure is:

```
procedure fDbiOpenUserList(UserList: TStrings);
var
  TmpCursor: hDbiCur;
  rslt: dbiResult;
  UsrDesc: USERDesc;
begin
  Check(DbiOpenUserList(TmpCursor));
  repeat
    rslt:= DbiGetNextRecord(TmpCursor, dbiNOLOCK, @UsrDesc, nil);
    if (rslt <> DBIERR_EOF) then begin
      UserList.Add('User name: ' + UsrDesc.szUserName);
      UserList.Add('Net Session: ' + IntToStr(UsrDesc.iNetSession));
      UserList.Add('Product Class: ' + IntToStr (UsrDesc.iProductClass));
    end;
  until (rslt <> DBIERR_NONE);
  Check(DbiCloseCursor(TmpCursor));
end;
```

DbiOpenVchkList {button C Examples,JI(`>example',`exdbiopenvchklist')} {button Delphi Examples,JI(`>example',`dexdbiopenvchklist')}

C syntax

```
DBIResult DBIFN DbiOpenVchkList (hDb, pszTableName, [pszDriverType],
  phChkCur);
```

Delphi syntax

```
function DbiOpenVchkList (hDb: hDBIDb; pszTableName: PChar; pszDriverType:
  PChar; var hChkCur: hDBICur): DBIResult stdcall;
```

Description

DbiOpenVchkList creates a table containing records with information about validity checks for fields within the specified table.

Parameters

hDb Type: hDBIDb (Input)
Specifies the database handle.

pszTableName Type: pCHAR (Input)
Pointer to the table name. If *pszTableName* is a fully qualified name of a table, the *pszDriverType* parameter need not be specified. If the path is not included, the path name is taken from the current directory of the database associated with *hDb*.

For SQL databases, this parameter can be a fully qualified name that includes the owner name.

pszDriverType Type: pCHAR (Input)
Pointer to the driver type. For Paradox, required only if no extension is specified by *pszTableName*. The only valid type is *szPARADOX*. This parameter is ignored if the database associated with *hDb* is a SQL database.

phChkCur Type: phDBICur (Output)
Pointer to the cursor handle.

Usage

Paradox: This function returns information about validity checks including required fields, minimum/maximum settings for fields, lookup tables, picture specifications, and default values.

SQL: The only validity check that can be created for SQL tables is *bRequired* (required fields). However, some drivers support reporting of fields with default values.

dBASE, FoxPro, Access: This function is not supported.

Prerequisites

A valid database handle must be obtained.

Completion state

phChkCur points to the returned cursor handle on the table. Once the cursor is returned, the client application can retrieve information about validity checks from the table. The cursor is read-only.

DbiResult return values

DBIERR_NONE	The cursor to the table was returned successfully.
DBIERR_INVALIDHNDL	The specified database handle is invalid or NULL, or phChkCur is NULL.
DBIERR_INVALIDPARAM	The specified table name or the pointer to the table name is NULL.

DBIERR_NOSUCHTABLE	The specified table name does not exist.
DBIERR_UNKNOWNTBLTYPE	The specified driver type is invalid.

See also

DbiOpenRintList, DbiCreateTable, DbiGetVchkDesc

C Examples: DbiOpenVchkList

Get validity check information for the specified table.

VchkList must be allocated large enough to hold security information. This example uses the following input:

fDbiOpenVchkList(hDb, "ORDERS.DB", ValidityChecks);

```
DBIResult fDbiOpenVchkList(hDBIDb hTmpDb, pCHAR TblName, pCHAR VchkList)
{
   DBIResult      rslt;
   hDBICur        hVchkCur = 0;
   VCHKDesc       Vchk;
   CHAR           Buffer[200], Required[4];
   VchkList[0] = '\0';
   rslt = Chk(DbiOpenVchkList(hTmpDb, TblName, NULL, &hVchkCur));
   if (rslt == DBIERR_NONE)
   {
      while (DbiGetNextRecord(hVchkCur, dbiNOLOCK, (pBYTE)&Vchk, NULL) ==
  DBIERR_NONE)
      {
         if (Vchk.bRequired == FALSE)
            strcpy(Required, "False");
         else
            strcpy(Required, "True");
         wsprintf(Buffer, "\r\nField Number: %d, Required: %s",
  Vchk.iFldNum, Required);
         strcat(VchkList, Buffer);
      }
   }
   if (hVchkCur != 0)
      DbiCloseCursor(&hVchkCur);
   return rslt;
}
```

Delphi Examples: DbiOpenVchkList

Create a table containing information about validity checks for fields within the specified table:

Returns information about validity checks for fields in the dataset specified in the Tbl parameter. The information is appended to the string list object specified in the VchkList parameter.

This example uses the following input:

```
  fDbiOpenVchkList(OrdersTable, ListBox1.Items);
```

The procedure is:

```
procedure fDbiOpenVchkList(Tbl: TTable; var VCheckList: TStrings);
var
  TmpCursor: hdbicur;
  VCheck: VCHKDesc;
  rslt: dbiResult;
begin
  Check(DbiOpenVchkList(Tbl.DbHandle, PChar(Tbl.TableName), nil,
  TmpCursor));
  Check(DbiSetToBegin(TmpCursor));
  VCheckList.Clear;
  repeat
    rslt := DbiGetNextRecord(TmpCursor, dbiNOLOCK, @VCheck, nil);
    if (rslt <> DBIERR_EOF) then begin
      VCheckList.Add('Field Number: ' + IntToStr(VCheck.ifldNum));
      If VCheck.bRequired = True then
        VCheckList.Add('Field is required: TRUE')
      else
        VCheckList.Add('Field is required: FALSE');
      If VCheck.bHasMinVal = True then
        VCheckList.Add('Has Minimum Value: TRUE')
      else
        VCheckList.Add('Has Minimum Value: FALSE');
      If VCheck.bHasMaxVal = True then
        VCheckList.Add('Has Maximum Value: TRUE')
      else
        VCheckList.Add('Has Maximum Value: FALSE');
      If VCheck.bHasDefVal = True then
        VCheckList.Add('Has Default Value: TRUE')
      else
        VCheckList.Add('Has Default Value: FALSE');
    end;
  until rslt <> DBIERR_NONE;
  Check(DbiCloseCursor(TmpCursor));
end;
```

DbiPackTable {button C Examples,JI(`>example',`exdbideleterecord')} {button Delphi Examples,JI(`>example',`dexdbipacktable')}

C syntax

```
DBIResult DBIFN DbiPackTable (hDb, hCursor, pszTableName, [pszDriverType],
  bRegenIdxs);
```

Delphi syntax

```
function DbiPackTable (hDb: hDBIDb; hCursor: hDBICur; pszTableName: PChar;
  pszDriverType: PChar; bRegenIdxs: Bool): DBIResult stdcall;
```

Description

DbiPackTable optimizes table space by rebuilding the table associated with *hCursor* and releasing any free space.

Parameters

hDb Type: hDBIDb (Input)
Specifies the valid database handle.

hCursor Type: hDBICur (Input)
Specifies the cursor on the table to be packed. Optional. If *hCursor* is specified, the operation is performed on the table associated with the cursor. If *hCursor* is NULL, *pszTableName* and *pszDriverType* determine the table to be used.

pszTableName Type: pCHAR (Input)
Pointer to the table name. Optional. If *hCursor* is NULL, *pszTblName* and *pszTblType* determine the table to be used. (If both *pszTableName* and *hCursor* are specified, *pszTableName* is ignored.) If *pszTableName* is a fully qualified name of a table, the *pszDriverType* parameter need not be specified. If the path is not included, the path name is taken from the current directory of the database associated with *hDb*.

pszDriverType Type: pCHAR (Input)
Pointer to the driver type. Optional. This parameter is required if *pszTableName* has no extension. The only valid *pszDriverType* is szDBASE.

bRegenIdxs Type: BOOL (Input)
Specifies whether or not to regenerate out-of-date table indexes. If TRUE, all out-of-date table indexes are regenerated (applies to maintained indexes only). Otherwise, out-of-date indexes are not regenerated.

Usage

dBASE or FoxPro: dBASE and FoxPro let users mark a record for deletion (as opposed to actually removing it from the table). The only way to permanently remove marked records is with DbiPackTable.

Paradox: This function is not valid for Paradox tables. Use DbiDoRestructure with the bPack option, instead.

SQL, Access: This function is not valid for SQL or Access tables.

Prerequisites

Exclusive access to the table is required.

DbiResult return values

DBIERR_NONE	The table was successfully rebuilt.
DBIERR_INVALIDPARAM	The specified table name or the pointer to the table name is NULL.
DBIERR_INVALIDHNDL	The specified database handle or cursor handle is invalid or NULL.

DBIERR_NOSUCHTABLE	Table name does not exist.
DBIERR_UNKNOWNTBLTYPE	Table type is unknown.
DBIERR_NEEDEXCLACCESS	The table is not open in exclusive mode.

See also

DbiOpenTable, DbiDeleteRecord, DbiDoRestructure

Delphi Examples: DbiPackTable

Example 1: Pack a Paradox or dBASE table.

This example will pack a Paradox or dBASE table therfore removing already deleted rows in a table. This function will also regenerate all out-of-date indexes (maintained indexes). This example uses the following input:

```
PackTable(Table1)
```

The function is defined as follows:

```
// Pack a Paradox or dBASE table
// The table must be opened execlusively before calling this function...
procedure PackTable(Table: TTable);
var
  Props: CURProps;
  hDb: hDBIDb;
  TableDesc: CRTblDesc;
begin
  // Make sure the table is open exclusively so we can get the db handle...
  if not Table.Active then
    raise EDatabaseError.Create('Table must be opened to pack');
  if not Table.Exclusive then
    raise EDatabaseError.Create('Table must be opened exclusively to pack');

  // Get the table properties to determine table type...
  Check(DbiGetCursorProps(Table.Handle, Props));

  // If the table is a Paradox table, you must call DbiDoRestructure...
  if Props.szTableType = szPARADOX then begin
    // Blank out the structure...
    FillChar(TableDesc, sizeof(TableDesc), 0);
    // Get the database handle from the table's cursor handle...
    Check(DbiGetObjFromObj(hDBIObj(Table.Handle), objDATABASE,
  hDBIObj(hDb)));
    // Put the table name in the table descriptor...
    StrPCopy(TableDesc.szTblName, Table.TableName);
    // Put the table type in the table descriptor...
    StrPCopy(TableDesc.szTblType, Props.szTableType);
    // Set the Pack option in the table descriptor to TRUE...
    TableDesc.bPack := True;
    // Close the table so the restructure can complete...
    Table.Close;
    // Call DbiDoRestructure...
    Check(DbiDoRestructure(hDb, 1, @TableDesc, nil, nil, nil, False));
  end
  else
    // If the table is a dBASE table, simply call DbiPackTable...
    if (Props.szTableType = szDBASE) then
      Check(DbiPackTable(Table.DBHandle, Table.Handle, nil, szDBASE, True))
    else
      // Pack only works on PAradox or dBASE; nothing else...
      raise EDatabaseError.Create('Table must be either of Paradox or dBASE
  ' +
```

```
        'type to pack');

  Table.Open;
end;
```

DbiPutBlob {button C Examples,JI(`>example',`exdbiputblob')} {button Delphi Examples,JI(`>example',`dexdbiputblob')}

C syntax

```
DBIResult DBIFN DbiPutBlob (hCursor, pRecBuf, iField, iOffSet, iLen, pSrc);
```

Delphi syntax

```
function DbiPutBlob (hCursor: hDBICur; pRecBuf: Pointer; iField: Word;
  iOffSet: Longint; iLen: Longint; pSrc: Pointer): DBIResult stdcall;
```

Description

DbiPutBlob writes data into an open BLOB field.

Parameters

hCursor Type: hDBICur (Input)
Specifies the cursor handle.

pRecBuf Type: pBYTE (Input)
Pointer to the record buffer.

iField Type: UINT16 (Input)
Specifies the ordinal number of a BLOB field within the record buffer.

iOffSet Type: UINT32 (Input)
Specifies the starting position, offset from the beginning of the BLOB, where the data is to be written. This value must not exceed the length of the BLOB. Valid values of *iOffset* range from 0 to the BLOB field's length. If *iOffset* is less than the BLOB field's length, part of the existing BLOB field is overwritten. If *iOffset* is equal to the length of the BLOB field, the data is appended to the existing BLOB field.

If the BLOB field also has a BLOB header (BLOB tuple area), and *iOffset* falls within that header area, the information in the tuple is also updated when DbiModifyRecord, DbiAppendRecord, or DbiInsertRecord is called.

iLen Type: UINT32 (Input)
Specifies the number of bytes to write to the BLOB field. *iLen* can be greater than 64K.

pSrc Type: pBYTE (Input)
Pointer to the data to be written to the BLOB field.

Usage

The block of data supplied in *pSrc* is transferred to the BLOB field, based on the values specified in *iOffset* and *iLen*. DbiPutBlob can access data in blocks larger than 64Kb, depending on the size you allocate for the buffer.

Note: This does not update the underlying table. The client application must call DbiAppendRecord, DbiModifyRecord, or DbiInsertRecord, using this record buffer, to update the table with the BLOB field.

Prerequisites

The BLOB field must be opened in read-write mode.

Completion state

Performs the equivalent of DbiPutField, for a BLOB field.

DbiResult return values

DBIERR_NONE	The data was successfully written to the BLOB field.
DBIERR_BLOBNOTOPENED	The specified BLOB field was not opened via a call to DbiOpenBlob.
DBIERR_INVALIDBLOBHANDLE	The record buffer supplied contains an invalid BLOB handle.

DBIERR_NOTABLOB	The specified field number does not correspond to a BLOB field.
DBIERR_INVALIDBLOBOFFSET	The specified iOffSet is greater than the length of the BLOB field.
DBIERR_READONLYFLD	The BLOB field was opened in dbiREADONLY mode and cannot be modified.

See also

DbiAppendRecord, DbiModifyRecord, DbiInsertRecord, DbiGetBlob, DbiOpenBlob, DbiTruncateBlob, DbiFreeBlob, DbiGetBlobSize

C Examples: DbiPutBlob

Modify the current record and Blob:

The field specified **must** be a valid memo blob. *pTmpRecBuf* **must** have valid record information. This example uses the following input:

fBlobExample1(hCur, pRecBuf, 7, "Blob text goes here!!");

```
DBIResult fBlobExample2 (hDBICur hTmpCur, pBYTE pTmpRecBuf, UINT16 uFldNum,
                     char *Text)
{
   DBIResult rslt;
   rslt = Chk(DbiOpenBlob(hTmpCur, pTmpRecBuf, uFldNum, dbiREADWRITE));
   if (rslt != DBIERR_NONE)
      return rslt;
   if (Chk(DbiPutBlob(hTmpCur, pTmpRecBuf, uFldNum, 0, strlen(Text) + 1,
                     (pBYTE)Text)) == DBIERR_NONE)
      rslt = Chk(DbiModifyRecord(hTmpCur, pTmpRecBuf, TRUE));
   Chk(DbiFreeBlob(hTmpCur, pTmpRecBuf, uFldNum));
   return rslt;
}
```

Delphi Examples: DbiPutBlob

Modify the current record and blob.

The field specified must be a valid memo blob. The pointer *pTmpRecBuf* must have valid record information. This example uses the following input:

```
  fBlobExample1(hCur, pRecBuf, 7, "Blob text goes here!!");
```

The procedure is:

```
procedure fBlobExample2(hTmpCur: hDBICur; pTmpRecBuf: pBYTE;uFldNum:
  LongInt;NewText: string);
begin
   Check(DbiOpenBlob(hTmpCur, pTmpRecBuf, uFldNum, dbiREADWRITE));
   Check(DbiPutBlob(hTmpCur, pTmpRecBuf, uFldNum, 0, StrLen(PChar(NewText))
  + 1, PChar(NewText)));
   Check(DbiModifyRecord(hTmpCur, pTmpRecBuf, True));
   Check(DbiFreeBlob(hTmpCur, pTmpRecBuf, uFldNum));
end;
```

DbiPutField {button C Examples,JI(`>example',`exdbiputfield')} {button Delphi Examples,JI(`>example',`dexdbiputfield')}

C syntax

```
DBIResult DBIFN DbiPutField (hCursor, iField, pRecBuf, pSrc);
```

Delphi syntax

```
function DbiPutField (hCursor: hDBICur; iField: Word; pRecBuff: Pointer;
  pSrc: Pointer): DBIResult stdcall;
```

Description

DbiPutField writes the field value to the correct location in the supplied record buffer.

Parameters

hCursor Type: hDBICur (Input)
Specifies the cursor handle.

iField Type: UINT16 (Input)
Specifies the ordinal number of the field to be updated.

pRecBuf Type: pBYTE (Input)
Pointer to the record buffer, which is updated upon success.

pSrc Type: pBYTE (Input)
Pointer to the new field value.

Usage

This function is used to update a record one field at a time. If a NULL pointer is supplied, the field is set to NULL or blank.

If the xltMODE for the cursor is xltFIELD, *pSrc* is assumed to contain field data in BDE logical format. This data is translated to the driver's physical type by this function. If xltMODE is xltNONE, *pSrc* is assumed to contain field data in physical format.

DbiPutField is not supported with BLOB fields.

Oracle8: DbiPutField on an ADT field is only supported if the value put is NULL. This will have the effect of nulling out all of the child fields of the ADT. DbiPutField of a non-NULL value on an ADT member sets all parent ADT fields to not NULL. DbiPutField with NULL on an ADT member does not set ADT parents to NULL, even if all siblings are NULL.

Prerequisites

DbiVerifyField may be called to test for field level integrity violations.

Completion state

After using DbiPutField one or more times, the client application must call DbiInsertRecord, DbiAppendRecord, or DbiModifyRecord to update the table with the record buffer. If the function fails, the record buffer is not affected.

DbiResult return values

DBIERR_NONE	The field was updated successfully.
DBIERR_INVALIDHNDL	The specified cursor handle is invalid or NULL.
DBIERR_OUTOFRANGE	iField is equal to zero, or is greater than the number of fields in the table.
DBIERR_INVALIDXLATION	A translation error has occurred.

See also

DbiVerifyField, DbiAppendRecord, DbiInsertRecord, DbiModifyRecord, DbiSetToKey, DbiGetField, DbiPutBlob

C Examples: DbiPutField

Example 1: Put the field value by field number.

This example uses the following input:

fDbiPutField1(hPXCur, pPXRecBuf, 1, (pBYTE)&DFloat);

```
DBIResult fDbiPutField1(hDBICur hTmpCur, pBYTE pTmpRecBuf, INT16 FldNum,
  pBYTE Info)
{
   DBIResult      rslt;
   rslt = Chk(DbiPutField(hTmpCur, FldNum, pTmpRecBuf, Info));
   return rslt;
}
```

Example 2: Put the field value specified by a field name..

If an invalid field name is given, an error is returned. This example uses the following input:

fDbiPutField2(hPXCur, pPXRecBuf, "STOCK NO", (pBYTE)&DFloat);

```
DBIResult fDbiPutField2(hDBICur hTmpCur, pBYTE pTmpRecBuf, pCHAR FldName,
  pBYTE Info)
{
   DBIResult      rslt;
   CURProps       CurProps;
   pFLDDesc       pFldDesc;
   UINT16         Field;
   BOOL           Found = FALSE;
   rslt = Chk(DbiGetCursorProps(hTmpCur, &CurProps));
   if (rslt != DBIERR_NONE)
      return rslt;
   pFldDesc = (pFLDDesc)malloc(CurProps.iFields * sizeof(FLDDesc));
   if (pFldDesc == NULL)
      return DBIERR_NOMEMORY;
   rslt = Chk(DbiGetFieldDescs(hTmpCur, pFldDesc));
   if (rslt != DBIERR_NONE)
   {
      free(pFldDesc);
      return rslt;
   }
   for(Field = 0; Field < CurProps.iFields; Field++)
   {
      if (strcmpi(pFldDesc[Field].szName, FldName) == 0)
      {
         Found = TRUE;
         if (Info != NULL)
            rslt = Chk(DbiPutField(hTmpCur, pFldDesc[Field].iFldNum,
  pTmpRecBuf, Info));
      }
   }
   if (Found == FALSE)
     rslt = DBIERR_INVALIDFIELDNAME;
   free(pFldDesc);
   return rslt;
}
```

Delphi Examples: DbiPutField

An example for this function is under development and will be provided in an upcoming Help release.

DbiQAlloc {button C Examples,JI(`>example',`exdbiqsetparams')} {button Delphi Examples,JI(`>example',`dexdbiqexec')}

C syntax

```
DBIResult DBIFN DbiQAlloc (hDb, eQryLang, phStmt);
```

Delphi syntax

```
function DbiQAlloc (hDb: hDBIDb; eQryLang: DBIQryLang; var hStmt: hDBIStmt):
  DBIResult stdcall;
```

Description

DbiQAlloc allocates a statement handle required by query prepare functions.

Parameters

hDb Type: hDBIDb (Input
Specifies the database handle

eQryLang Type: DBIQryLang (Input)
Specifies the query language, "qrylangSQL" or "qrylangQBE".

phStmt Type: phDBIStmt (Output)
Provides the pointer to the statement handle.

Usage

This function must be called before calling DbiQPrepare in order to obtain a new statement handle. It allows you to set properties to control the query execution process before calling the next function. All query process procedures must follow this pattern:

```
DbiQAlloc
...  // Other query preparation and execution functions.
DbiQFree
```

This is the only way to do query process procedures.

If you want the query to return an updateable record set, use the following sequence of calls:

```
DbiQAlloc( hDb, eQryLang, &phStmt );
DbiSetProp( hStmt, stmtLIVENESS, wantLIVE (or wantCANNED) );
DbiQPrepare( hStmt, pszQuery );
```

In every case, after you have prepared and executed the query, call DbiQFree to free the resources allocated to this query (right after DbiQExec).

DbiResult return values

DBIERR_NONE The statement handle was returned.

See also

DbiQPrepare, DbiQExec, DbiQFree, DbiQSetParams

DbiQExec {button C Examples,JI(`>example',`exdbiqsetparams')} {button Delphi Examples,JI(`>example',`dexdbiqexec')}

C syntax

```
DBIResult DBIFN DbiQExec (hStmt, phCur);
```

Delphi syntax

```
function DbiQExec (hStmt: hDBIStmt; phCur: phDBICur): DBIResult stdcall;
```

Description

DbiQExec executes the previously prepared query identified by the supplied statement handle and returns a cursor to the result set, if one is generated.

Parameters

hStmt Type: hDBIStmt (Input)
Specifies the statement handle.

phCur Type: phDBICur (Output)
Pointer to the cursor handle.

Usage

This function is used to execute a prepared query. If the query returns a result set, the cursor handle to the result set is returned into the address given by *phCur*. If the query does not generate a result set, the returned cursor handle is zero. If no cursor handle address is given and a result set would be returned, the result set is discarded.

The same prepared query can be executed several times, but only after the returned cursor has been closed.

DbiResult return values

DBIERR_NONE The prepared query was executed successfully.

DBIERR_MULTIRESULTS Query returned multiple result sets.

See also

DbiQAlloc, DbiQPrepare, DbiQExecDirect, DbiQFree, DbiQSetParams

Delphi Examples: DbiQExec

Create a table on disk by using a given SQL statement.

The filename is also passed as the parameter TblName. The function returns the number of rows in the result table. This example uses the following input:

```
fDbiQExec(Database1.Handle, 'QUERY.DB', 'SELECT * FROM TEST;');
fDbiQExec(Table1.DBHandle, 'QUERY2.DB', 'SELECT * FROM CUSTOMER');
```

The function is:

```
function fDbiQExec(hTmpDb: hDBIDB; TblName, SQL: string): Longint;
var
  hStmt: hDBIStmt;
  hQryCur, hNewCur: hDBICur;
  iRecCount: LongInt;
begin
  hQryCur := nil;
  hNewCur := nil;
  hStmt := nil;
  try
    Check(DbiQAlloc(hTmpDb, qrylangSQL, hStmt));
    Check(DbiQPrepare(hStmt, PChar(SQL)));
    Check(DbiQExec(hStmt, @hQryCur));
    Check(DbiQInstantiateAnswer(hStmt, hQryCur, PChar(TblName), szPARADOX,
      True, @hNewCur));
    Check(DbiGetRecordCount(hNewCur, iRecCount));
    Result := iRecCount;
  finally
    if (hStmt <> nil) then
      Check(DbiQFree(hStmt));
    if (hNewCur <> nil) then
      Check(DbiCloseCursor(hNewCur));
  end;
end;
```

DbiQExecDirect {button C Examples,JI(`>example',`exdbiqexecdirect')} {button Delphi Examples,JI(`>example',`dexdbiqexecdirect')}

C syntax

```
DBIResult DBIFN DbiQExecDirect (hDb, eQryLang, pszQuery, phCur);
```

Delphi syntax

```
function DbiQExecDirect (hDb: hDBIDb; eQryLang: DBIQryLang; pszQuery: PChar;
  phCur: phDBICur): DBIResult stdcall;
```

Description

DbiQExecDirect executes a SQL or QBE query and returns a cursor to the result set, if one is generated.

Parameters

hDb Type: hDBIDb (Input)
Specifies the database handle.

eQryLang Type: DBIQryLang (Input)
Specifies the query language, QBE or SQL.

pszQuery Type: pCHAR (Input)
Pointer to the query, formulated in the appropriate language.

phCur Type: phDBICur (Output)
Pointer to the cursor handle.

Usage

This function is used to immediately prepare and execute a query. If the query returns a result set, the cursor handle to the result set is returned into the address given by *phCur*. If the query does not generate a result set, the returned cursor handle is zero. If no cursor handle address is given and a result set would be returned, the result set is discarded.

SQL: For SQL language queries, if the database handle given does not refer to a server database, the BDE SQL dialect is recognized. Otherwise, the appropriate server dialect is expected. Heterogeneous data access and cross-server data access can be achieved by using the BDE SQL dialect and referencing tables qualified with database alias names.

QBE: For QBE language queries, the BDE QBE Syntax is expected. Heterogeneous data access and cross-server data access can be achieved.

DbiResult return values

DBIERR_NONE The query was successfully prepared and executed.

DBIERR_MULTIRESULTS Query returned multiple result sets.

See also

DbiQAlloc, DbiQExec, DbiQFree, DbiQPrepare, DbiQSetParams

C Examples: DbiQExecDirect

Execute a SQL Statement and return the numbers in the result set if applicable:
Note that Count will be 0 if a result set is not created. The following input is used in this example: fDbiQExecDirect("Select * from 'CUST.DBF'", hDb, &hTmpCur, &Count);

```
DBIResult fDbiQExecDirect(pCHAR QryStr, hDBIDb hTmpDb, phDBICur phTmpCur,
  pUINT32 Count)
{
   DBIResult   rslt;

   *Count = 0;
   rslt = Chk(DbiQExecDirect(hTmpDb, qrylangSQL, QryStr, phTmpCur));
   if (rslt == DBIERR_NONE)
   {
      if (*phTmpCur != 0)
         Chk(DbiGetRecordCount(*phTmpCur, Count));
   }
   return rslt;
}
```

Delphi Examples: DbiQExecDirect

Execute a SQL statement and return the numbers in the result set if applicable.
Count will be 0 if a result set is not created. The function also returns the number of rows in the result table. This example uses the following input:

```
fDbiQExecDirect('Select * from CUSTOMER', Database1.Handle, hTmpCur);
```

The function is:

```
function fDbiQExecDirect(QryStr: string; hTmpDb: hDBIDb; var hTmpCur:
  hDBICur): Longint;
var
  Count: Longint;
begin
   Check(DbiQExecDirect(hTmpDb, qrylangSQL, PChar(QryStr), @hTmpCur));
   if (hTmpCur <> nil) then begin
     Check(DbiGetRecordCount(hTmpCur, Count));
     Result := Count;
   end
   else
     Result := 0;
end;
```

DbiQExecProcDirect {button C Examples,JI(`>example',`exdbiqexecprocdirect')} {button Delphi Examples,JI(`>example',`dexdbiqexecprocdirect')}

C syntax

```
DBIResult DBIFN DbiQExecProcDirect (hDb, pszProc, uParamDescs, paParamDescs,
  pRecBuf, phCur);
```

Delphi syntax

```
function DbiQExecProcDirect (hDb: hDBIDb; pszProc: PChar; uParamDescs: Word;
  paParamDescs: pSPParamDesc; pRecBuff: Pointer; var hCur: hDBICur):
  DBIResult stdcall;
```

Description

DbiQExecProcDirect executes a stored procedure and returns a cursor to the result set, if one is generated.

Usage

You must set **all** parameters (including output parameters) before statement execution. After execution, output parameter values are placed in the specified offset of the client-supplied *pRecBuf*. If the output parameter value is NULL or TRUNCATED, then *indNULL* or *indTRUNC* is placed in the *iNulloffset* of the client-supplied *pRecBuf*. Note that *indNULL* and *indTRUNC* are enums defined by *eINDValues*.

Sybase: Output parameter values are not available until **after** all rows have been fetched from the result set.

InterBase: When calling DbiQExecProcDirect, all input parameters must be specified **before** output parameters.

Parameters

hDb Type: hDBIDb (Input)
Specifies the database handle.

pszProc Type: pCHAR (Input)
Stored procedure name.

uParamDescs Type: UINT16 (Input)
Number of parameter descriptors.

paParamDescs Type: pSPParamDesc (Input)
Array of parameter descriptors.

pRecBuf Type: pBYTE (Input)
Record buffer.

phCur Type: phDBICur (Output)
Pointer to the cursor handle.

DbiResult return values

DBIERR_NONE The stored procedure was successfully prepared and executed.

DBIERR_MULTIRESULTS Query returned multiple result sets.

See also

DbiQAlloc, DbiQPrepareProc, DbiQSetProcParams, DbiOpenSPList, DbiOpenSPParamList

C Examples: DbiQExecProcDirect

An example for this function is under development and will be provided in an upcoming Help release.

Delphi Examples: DbiQExecProcDirect

An example for this function is under development and will be provided in an upcoming Help release.

DbiQGetBaseDescs {button C Examples,JI(`>example',`exdbiqgetbasedescs')} {button Delphi Examples,JI(`>example',`dexdbiqgetbasedescs')}

C syntax

```
DBIResult DBIFN DbiQGetBaseDescs (hStmt, phCur);
```

Delphi syntax

```
function DbiQGetBaseDescs (hStmt: hDBIStmt; phCur: phDBICur): DBIResult
  stdcall;
```

Description

DbiQGetBaseDescs returns the original database, table, and field names of the fields that make up the result set of a query.

Parameters

hStmt Type: hDBIStmt (Input)
Statement handle.

phCur Type: phDBICur (Input)
Cursor of type STMTBaseDesc.

Usage

This function gives the client the original columns upon which the result set is based; in other words, the original columns from the SQL select list along with their table and database names. By associating the base or original field attributes with the result set, Delphi users can obtain a complete picture.

As with other BDE functions that return a cursor, the cursor must be closed by the client. The normal calling sequence to use is: DbiQAlloc, DbiQPrepare, DbiQGetBaseDescs, DbiQExec, DbiQFree, DbiCloseCursor(pStmtBaseCur).

DbiResult return values

DBIERR_NONE The query's resources were released successfully.

DBIERR_NOTSUPPORTED A QBE language query could processed.

See also

DbiQAlloc, DbiQPrepare, DbiQExec, DbiQFree, DbiCloseCursor

C Examples: DbiQGetBaseDescs

An example for this function is under development and will be provided in an upcoming Help release.

Delphi Examples: DbiQGetBaseDescs

Return the original database, table, and field names for a query.

The query from which the base descriptions come is specified in the Query parameter. Descriptions are added to the string list object specified in the List parameter. This example uses the following input:

```
  GetBaseDescs(Query2, Memo1.Lines);
```

```
procedure GetBaseDescs(Query: TQuery; List: TStrings);
var
  hCur: hDBICur;
  rslt: DBIResult;
  Descs: STMTBaseDesc;
begin
  hCur := nil;
  try
    // Look at DbiQGetBaseDescs in the BDE32.HLP for more information...
    Check(DbiQGetBaseDescs(Query.STMTHandle, hCur));
    repeat
      rslt := DbiGetNextRecord(hCur, dbiNOLOCK, @Descs, nil);
      if (rslt = DBIERR_NONE) then
        // Look at STMTBaseDescs in the BDE32.HLP for more information...
        List.Add(Format('DB Name: %s  Table Name: %s  Field Name: %s',
          [Descs.szDatabase, Descs.szTableName, Descs.szFieldName]))
      else
        if (rslt <> DBIERR_EOF) then
          Check(rslt);
    until (rslt <> DBIERR_NONE);
  finally
    if (hCur <> nil) then
      check(DbiCloseCursor(hCur));
  end;
end;
```

DbiQFree {button C Examples,JI(`>example',`exdbiqsetparams')} {button Delphi Examples,JI(`>example',`dexdbiqexec')}

C syntax
```
DBIResult DBIFN DbiQFree (phStmt);
```

Delphi syntax
```
function DbiQFree (var hStmt: hDBIStmt): DBIResult stdcall;
```

Description
DbiQFree frees the resources associated with a previously allocated query identified by the supplied statement handle.

Parameters
phStmt Type: phDBIStmt (Input)
Pointer to the statement handle.

Usage
This function is used to release the resources acquired during the query execution process. If cursors are associated with an outstanding result set produced by execution of the statement, the cursors remain valid and the dependent statement resources are not released until the last cursor has been closed or the result set is read to completion, whichever happens first.

DbiResult return values
DBIERR_NONE The query's resources were released successfully.

See also
DbiQAlloc, DbiQExec, DbiQExecDirect, DbiQPrepare,

DbiQInstantiateAnswer {button C Examples,JI(`>example',`exdbiqsetparams')} {button Delphi Examples,JI(`>example',`dexdbiqexec')}

C syntax

```
DBIResult DBIFN DbiQInstantiateAnswer ( hStmt, [hCursor],
  pszAnswerName,pszAnswerType, bOverWrite, phDstCursor )
```

Delphi syntax

```
function DbiQInstantiateAnswer (hStmt: hDBIStmt; hCur: hDBICur;
  pszAnswerName: PChar; pszAnswerType : PChar; bOverWrite: Bool; var phCur:
  phDBICur): DBIResult stdcall;
```

Description

DbiQInstantiateAnswer creates an ANSWER table of type PARADOX or DBASE. The flags *pszAnswerName* and *pszAnswerType* may be used in renaming and changing the type respectively. If the flag *bOverWrite* is set to TRUE, then it will overwrite the existing *pszAnswerTable*.

Parameters

hStmt Type: hDBIStmt (Input)
Specifies the statement handle

hCursor Type: hDBICur (Input)
Specifies the cursor handle. Optional.

pszAnswerName Type: pCHAR (Input)
Pointer to the name of the permanent table.

pszAnswerType Type: pCHAR (Input)
Pointer to the name of the driver type.

bOverWrite Type: BOOL (Input)
If set to TRUE, overwrites the existing file.

phDstCursor Type: phDBICur (Output)
Pointer to the cursor handle.

Usage

DbiQInstantiateAnswer is used to create a permanent table from a cursor handle. The table name is ANSWER.DB by default or it will create *pszAnswerName* with *pszAnswerType*. You can use the *bOverWrite* flag to overwrite the existing *pszAnswerTable*.

Prerequisites

A statement handle must be allocated with DbiQAlloc.

Completion state

The table is saved to disk when the cursor is closed.

DbiResult return values

DBIERR_NONE The temporary table has been designated as a permanent table.

See also

DbiSaveChanges, DbiCreateTempTable, DbiCloseCursor

DbiQPrepare {button C Examples,JI(`>example',`exdbiqsetparams')} {button Delphi Examples,JI(`>example',`dexdbiqexec')}

C syntax
```
DBIResult DBIFN DbiQPrepare (hStmt, pszQuery);
```

Delphi syntax
```
function DbiQPrepare (hStmt: hDBIStmt; pszQuery: PChar): DBIResult stdcall;
```

Description
DbiQPrepare prepares a SQL or QBE query for execution, and accepts a handle to a statement containing the prepared query.

Parameters
hStmt Type: hDBIStmt (Input)
Specifies the statement handle

pszQuery Type: pCHAR (Input)
Pointer to the query, formulated in the appropriate language.

Usage
This function is used to prepare a query for subsequent execution.

SQL: For SQL language queries, if the database handle given does not refer to a server database, the BDE SQL dialect is recognized. Otherwise, the appropriate server dialect is expected. Heterogeneous data access and cross-server data access can be achieved by using the BDE SQL dialect and referencing tables qualified with database alias names.

QBE: For QBE language queries, the BDE QBE Syntax is expected. Heterogeneous data access and cross-server data access can be achieved.

DbiResult return values
DBIERR_NONE The query was successfully prepared for execution.

DBIERR_ALIASNOTOPEN One of the aliases used in the query was not opened prior to preparing the query. The alias name can be found on the error context stack.

See also
DbiQAlloc, DbiQExec, DbiQExecDirect, DbiQFree, DbiQSetParams

DbiQPrepareProc {button C Examples,JI(`>example',`exdbiqprepareproc')} {button Delphi Examples,JI(`>example',`dexdbiqprepareproc')}

C syntax

```
DBIResult DBIFN DbiQPrepareProc (hDb, pszProc, uParamDescs, paParamDescs,
  pRecBuf, phStmt);
```

Delphi syntax

```
function DbiQPrepareProc (hDb: hDBIDb; pszProc: PChar; uParamDescs: Word;
  paParamDescs: pSPParamDesc; pRecBuff: Pointer; var hStmt: hDBIStmt):
  DBIResult stdcall;
```

Description

DbiQPrepareProc prepares and optionally binds parameters for a stored procedure.

Parameters

hDb Type: hDBIDb (Input)
Specifies the database handle.

pszProc Type: pCHAR (Input)
Stored procedure name.

uParamDescs Type: UINT16 (Input)
Specifies the number of parameter descriptors.

paParamDescs Type: pSPParamDesc (Input)
Pointer to the array of parameter descriptors.

pRecBuf Type: pBYTE (Input)
Pointer to the record buffer (or NULL if parameters are not to be bound.)

phStmt Type: phDBIStmt (Output)
Specifies the returned statement handle.

Usage

Use with the existing functions DbiQExec and DbiQFree. If *pRecBuf* is NULL, then the parameters are not bound.

DbiResult return values

DBIERR_NONE The stored procedure was successfully prepared for execution.

See also

DbiQExecProcDirect, DbiQSetProcParams, DbiQExec, DbiQExecDirect, DbiQFree, DbiOpenSPList, DbiOpenSPParamList

C Examples: DbiQPrepareProc

An example for this function is under development and will be provided in an upcoming Help release.

Delphi Examples: DbiQPrepareProc

An example for this function is under development and will be provided in an upcoming Help release.

DbiQSetParams {button C Examples,JI(`>example',`exdbiqsetparams')} {button Delphi Examples,JI(`>example',`dexdbiqsetparams')}

C syntax

```
DBIResult DBIFN DbiQSetParams (hStmt, uFldDescs, paFldDescs, pRecBuf);
```

Delphi syntax

```
function DbiQSetParams (hStmt: hDBIStmt; uFldDescs: Word; paFldDescs:
  pFLDDesc; pRecBuff: Pointer): DBIResult stdcall;
```

Description

DbiQSetParams associates data with parameter markers embedded within a prepared query.

Parameters

hStmt Type: hDBIStmt (Input)
Specifies the statement handle.

uFldDescs Type: UINT16 (Input)
Specifies the number of parameter field descriptors given.

paFldDescs Type: pFLDDesc (Input)
Pointer to the array of parameter field descriptors.

pRecBuf Type: pBYTE (Input)
Pointer to the client buffer containing data for the specified fields.

Usage

This function is used to set the value of parameter markers in a prepared query before the query execution.

The field descriptor array and record buffer is constructed by the client and passed to BDE, which uses each specified field, along with the record buffer, to locate the data and set the specified parameter. Each field may be either a BDE type or a driver type for the database that the query is prepared for.

Parameter markers are "?", ":name", or "~name" (tilde, used only with QBE queries). The field descriptor for a "?" parameter marker must contain no name, and must contain a field number that matches the position of the "?" marker within the query, beginning with marker number one. The field descriptor for a ":name" marker must contain the name of the marker, and a field number of zero.

Parameter settings are retained from statement execution to statement execution. However, all parameters must be set before execution can occur.

To bind BLOBs and strings longer than 255 characters you must use BLOBParamDesc to describe the parameter in a FLDDesc structure. Clients need to allocate and set up a BLOBParamDesc structure for each blob or long string that will be passed as a parameter. For BLOBs, *iFldType* in FLDDesc is set to fldBLOB and *iSubType* is set to fldstMEMO or fldstBINARY. For long strings, *iFldType* should be set to fldZSTRING and the length should specified in *iUnits1* as normal. The BLOBParamDesc structure is copied into the parameter buffer at the specifed *iOffset*, then DbiQSetParams and DbiQExec are called as normal. You can determine if a field is a BLOB or long string by checking *iUnits1*. If *iUnits1* is 1, it is a BLOB; if it is greater than one, it is a long string. This test does not work on servers that don't maintain a distinction between BLOBs and long strings, like Sybase.

DbiResult return values

DBIERR_NONE The value of parameter markers was successfully set.

DBIERR_OBJNOTFOUND A field descriptor references a parameter marker that does not exist.

DBIERR_INVALIDHNDL	DbiQSetParams was called without first having called DbiQPrepare.

See also

DbiQExec, DbiQFree, DbiQPrepare, DbiQAlloc

C Examples: DbiQSetParams

Create a Query result set table.

This example uses the following input:

```
fQFunction1(hDb, "Result");
```

```
DBIResult fQFunction1(hDBIDb hTmpDb, pCHAR TblName)
{
   DBIResult   rslt;
   hDBIStmt    hStmt;
   hDBICur     hQryCur = 0, hNewCur = 0;
   FLDDesc     FldDesc;
   CHAR        SQL[] =  "select c.cust_no, c.name, c.state_prov, "
                        "  o.'Sale Date', o.'Month' "
                       "from 'cust.dbf' c, orders o "
                        "where (c.cust_no between ? and 4999) "
                        "  and (c.cust_no = o.'customer no') "
                        "  and c.state_prov is not null "
                        "order by c.name desc, o.'Month' asc; ";
   DFLOAT      Cust = 2000.00;

   // Create a Field Descriptor for the parameter
   memset(&FldDesc, 0, sizeof(FLDDesc));
   FldDesc.iFldNum = 1;
   FldDesc.iFldType = fldFLOAT;
   FldDesc.iUnits1 = 1;
   FldDesc.iLen = sizeof(Cust);
   // Allocate a statment handle
   rslt = Chk(DbiQAlloc(hTmpDb, qrylangSQL, &hStmt));
   if  (rslt != DBIERR_NONE)
      return rslt;
   // Prepare the SQL statment
   rslt = Chk(DbiQPrepare(hStmt, SQL));
   if  (rslt != DBIERR_NONE)
   {
      DbiQFree(&hStmt);
      return rslt;
   }
   // Since only one parameter is used, there is no need for a record
  buffer.
   // Use the Cust variable itsself.
   rslt = Chk(DbiQSetParams(hStmt, 1, &FldDesc, (pBYTE)&Cust));
   if  (rslt != DBIERR_NONE)
   {
      DbiQFree(&hStmt);
      return rslt;
   }
   // Execute the SQL statment
   rslt = Chk(DbiQExec(hStmt, &hQryCur));
   if  (rslt != DBIERR_NONE)
   {
      DbiQFree(&hStmt);
      return rslt;
   }
   // Save the Result set to disk and close the result query(hQryCur)
   // Now there is an open cursor on the result set on disk.
   rslt = Chk(DbiQInstantiateAnswer(hStmt, hQryCur, TblName, szPARADOX,
```

```
                                        TRUE, &hNewCur));
   if  (rslt != DBIERR_NONE)
   {
      DbiQFree(&hStmt);
      return rslt;
   }
   // Close the cursor
   if (hNewCur != 0)
      Chk(DbiCloseCursor(&hNewCur));

   return rslt;
}
DBIResult fQFunction1(hDBIDb hTmpDb, pCHAR TblName)
{
   DBIResult   rslt;
   hDBIStmt    hStmt;
   hDBICur     hQryCur = 0, hNewCur = 0;
   FLDDesc     FldDesc;
   CHAR        SQL[] =  "select c.cust_no, c.name, c.state_prov, "
                        "  o.'Sale Date', o.'Month' "
                       "from 'cust.dbf' c, orders o "
                        "where (c.cust_no between ? and 4999) "
                        "  and (c.cust_no = o.'customer no') "
                        "  and c.state_prov is not null "

                        "order by c.name desc, o.'Month' asc; ";
   DFLOAT      Cust = 2000.00;

   // Create a Field Descriptor for the parameter
   memset(&FldDesc, 0, sizeof(FLDDesc));
   FldDesc.iFldNum = 1;
   FldDesc.iFldType = fldFLOAT;
   FldDesc.iUnits1 = 1;
   FldDesc.iLen = sizeof(Cust);
   // Allocate a statment handle
   rslt = DbiQAlloc(hTmpDb, qrylangSQL, &hStmt);
   if  (rslt != DBIERR_NONE)
      return rslt;
   // Prepare the SQL statment

   rslt = DbiQPrepare(hStmt, SQL);
   if  (rslt != DBIERR_NONE)
   {
      DbiQFree(&hStmt);
      return rslt;
   }
   // Since only one parameter is used, there is no need for a record
  buffer.
   // Use the Cust variable itsself.
   rslt = DbiQSetParams(hStmt, 1, &FldDesc, (pBYTE)&Cust);
   if  (rslt != DBIERR_NONE)
   {
      DbiQFree(&hStmt);
      return rslt;
   }
   // Execute the SQL statment
   rslt = DbiQExec(hStmt, &hQryCur);
```

```
   if  (rslt != DBIERR_NONE)

   {
      DbiQFree(&hStmt);
      return rslt;
   }
   // Save the Result set to disk and close the result query(hQryCur)
   // Now there is an open cursor on the result set on disk.
   rslt = DbiQInstantiateAnswer(hStmt, hQryCur, TblName, szPARADOX,
                                TRUE, &hNewCur);
   if  (rslt != DBIERR_NONE)
   {
      DbiQFree(&hStmt);
      return rslt;
   }
   // Close the cursors
   if (hNewCur != 0)
      DbiCloseCursor(&hNewCur);
   if (hQryCur != 0)
      DbiCloseCursor(&hQryCur);

   return rslt;
}
```

Inserts GIF image into a Sybase table with a column of type Image

```
#include <stdio.h>
#include <io.h>
#include <fcntl.h>
#include <assert.h>
#include <memory.h>
#include <malloc.h>
#include <string.h>
#include <idapi.h>

int main()
{
   hDBIDb       hDb;
   hDBIStmt     hStmt;
   char         *pszQuery = "insert into blobs values (?, ?)";
   FLDDesc      params[2];
   BLOBParamDesc     blob;
   BYTE         pParamBuf[10 + sizeof (BLOBParamDesc)+ 2];
   pBYTE         pBlobBuf;
   pBYTE        p = NULL;
   DBIResult    rc;
   int          fHandle;
   int          size;
   FILE         *infile;

   // get the file size.
   fHandle = open("c:\\temp\\cs1.gif", O_BINARY);
   assert (fHandle);
   size  = filelength(fHandle);
   close(fHandle);

   pBlobBuf = (pBYTE)malloc(size);
   assert(pBlobBuf);
```

```
 memset(pBlobBuf, 0x0, size);

 // Read the blob
 infile = fopen("c:\\temp\\cs1.gif", "r");
 assert(infile);
 rc = fread (pBlobBuf, size, 1, infile);
 fclose(infile);

 rc = DbiInit(NULL);

 if (rc == DBIERR_NONE)
      rc = DbiOpenDatabase("sybase_database", NULL, dbiREADWRITE,
                           dbiOPENSHARED, "sybpass", NULL, NULL, NULL,
&hDb);


 if (rc == DBIERR_NONE)
     rc = DbiQAlloc(hDb, qrylangSQL, &hStmt);

 if (rc == DBIERR_NONE)
     rc = DbiQPrepare(hStmt, pszQuery);

 if (rc == DBIERR_NONE)
 {
    memset (&params, 0x0, sizeof(params));
    memset (&blob, 0x0, sizeof(blob));
    memset (pParamBuf, 0x0, sizeof (pParamBuf));

    params[0].iFldNum = 1;
    params[0].iFldType = fldFLOAT;
    params[0].iOffset  = 0;
    params[0].iLen     = 8;
    params[0].iNullOffset = 8;

    params[1].iFldNum = 2;
    params[1].iFldType = fldBLOB;
    params[1].iSubType = fldstBINARY;
    params[1].iOffset  = 10;
    params[1].iLen     = sizeof(BLOBParamDesc);
    params[1].iNullOffset = params[1].iOffset + sizeof(BLOBParamDesc);
    params[1].iUnits1  = 0;
    params[1].iUnits2  = 0;

    p = pParamBuf;
    *(pDFLOAT)p = 7;
    p = pParamBuf+params[0].iNullOffset;
    *(pINT16)p = 8;

    blob.pBlobBuffer = pBlobBuf;
    blob.ulBlobLen   = size;

    p = pParamBuf+params[1].iOffset;
    *(pBLOBParamDesc)p = blob;

    p = pParamBuf+params[1].iNullOffset;
    *(pINT16)p = 1;
```

```
      rc = DbiQSetParams(hStmt, 2,params, pParamBuf);
   }

   if (rc == DBIERR_NONE)
       rc = DbiQExec(hStmt, NULL);

   rc = DbiQFree(&hStmt);

   rc = DbiCloseDatabase(&hDb);

   rc = DbiExit();

   if (pBlobBuf)
       free(pBlobBuf);

   return (rc);
}
```

Inserts a long character string into an Informix table

```
#include <memory.h>
#include <malloc.h>
#include <string.h>
#include <idapi.h>

int main()
{
   hDBIDb       hDb;
   hDBIStmt     hStmt;
   char         *pszQuery = "insert into longstrings values (?, ?)";
   FLDDesc      params[2];
   BLOBParamDesc     blob;
   BYTE         pParamBuf[10 + sizeof (BLOBParamDesc)+ 2];
   BYTE         pBlobBuf[32511];
   pBYTE        p = NULL;
   DBIResult    rc;

   rc = DbiInit(NULL);

   if (rc == DBIERR_NONE)
      rc = DbiOpenDatabase("informix_database", NULL, dbiREADWRITE,
                           dbiOPENSHARED, "infpass", NULL, NULL, NULL,
  &hDb);


   if (rc == DBIERR_NONE)
       rc = DbiQAlloc(hDb, qrylangSQL, &hStmt);

   if (rc == DBIERR_NONE)
       rc = DbiQPrepare(hStmt, pszQuery);

   if (rc == DBIERR_NONE)
   {
      memset (&params, 0x0, sizeof(params));
      memset (&blob, 0x0, sizeof(blob));
      memset (pParamBuf, 0x0, sizeof (pParamBuf));
      memset (pBlobBuf, 0x0, 32511);
```

```
      params[0].iFldNum = 1;
      params[0].iFldType = fldFLOAT;
      params[0].iOffset  = 0;
      params[0].iLen     = 8;
      params[0].iNullOffset = 8;

      params[1].iFldNum = 2;
      params[1].iFldType = fldZSTRING;
      params[1].iSubType = 0;
      params[1].iOffset  = 10;
      params[1].iLen     = sizeof(BLOBParamDesc);
      params[1].iNullOffset = params[1].iOffset + sizeof(BLOBParamDesc);
      params[1].iUnits1  = 32511;
      params[1].iUnits2  = 0;

      p = pParamBuf;
      *(pDFLOAT)p = 2;
      p = pParamBuf+params[0].iNullOffset;
      *(pINT16)p = 8;

      blob.pBlobBuffer = pBlobBuf;
      blob.ulBlobLen   = 0;

      p = pParamBuf+params[1].iOffset;
      *(pBLOBParamDesc)p = blob;

      p = pParamBuf+params[1].iNullOffset;
      *(pINT16)p = 1;

      p = pBlobBuf;

      for ( int i = 0; i < 32511; i++)
          *(p+i) = '3';

      rc = DbiQSetParams(hStmt, 2,params, pParamBuf);
   }

   if (rc == DBIERR_NONE)
       rc = DbiQExec(hStmt, NULL);

   rc = DbiQFree(&hStmt);

   rc = DbiCloseDatabase(&hDb);

   rc = DbiExit();

   return (rc);
}
```

Delphi Examples: DbiQSetParams

An example for this function is under development and will be provided in an upcoming Help release.

DbiQSetProcParams {button C Examples,JI(`>example',`exdbiqsetprocparams')} {button Delphi Examples,JI(`>example',`dexdbiqsetprocparams')}

C syntax

```
DBIResult DBIFN DbiQSetProcParams (hStmt, uParamDescs, paParamDescs, pRecBuf
  );
```

Delphi syntax

```
function DbiQSetProcParams (hStmt: hDBIStmt; uParamDescs: Word;
  paParamDescs: pSPParamDesc; pRecBuff: Pointer): DBIResult stdcall;
```

Description

DbiQSetProcParams binds parameters for a stored procedure prepared with DbiQPrepareProc.

Parameters

hStmt Type: hDBIStmt (Input)
Specifies the statement handle.

uParamDescs Type: UINT16 (Input)
Specifies the number of parameter descriptors.

paParamDescs Type: pSPParamDesc (Input)
Pointer to the array of parameter descriptors.

pRecBuf Type: pBYTE (Input)
Pointer to the record buffer. (Or NULL if parameters are not to be bound.)

Usage

You must set **all** parameters (including output parameters) before statement execution. After execution, output parameter values are placed in the specified offset of the client-supplied *pRecBuf*. If the output parameter value is NULL or TRUNCATED, then *indNULL* or *indTRUNC* is placed in the *iNulloffset* of the client-supplied *pRecBuf*. Note that *indNULL* and *indTRUNC* are enums defined by *eINDValues*.

Sybase: Output parameter values are not available until **after** all rows have been fetched from the result set.

InterBase: When calling DbiQSetProcParams and DbiQPrepareProc, all input parameters must be specified **before** output parameters.

Prerequisites

The function DbiQPrepareProc must be called before calling DbiQSetProcParams.

These function calls assume that the client knows the stored procedure parameters, parameter types (such as INPUT, OUTPUT, INPUT/OUTPUT), and parameter datatypes.

DbiResult return values

DBIERR_NONE The value of parameter markers was successfully set.

DBIERR_OBJNOTFOUND A field descriptor references a parameter marker that does not exist.

See also

DbiQPrepareProc, DbiQExecProcDirect, DbiOpenSPList, DbiOpenSPParamList

C Examples: DbiQSetProcParams

An example for this function is under development and will be provided in an upcoming Help release.

Delphi Examples: DbiQSetProcParams

An example for this function is under development and will be provided in an upcoming Help release.

DbiReadBlock {button C Examples,JI(`>example',`exdbireadblock')} {button Delphi Examples,JumpID(>example, dexdbiwriteblock)}

C syntax

```
DBIResult DBIFN DbiReadBlock (hCursor, piRecords, pBuf);
```

Delphi syntax

```
function DbiReadBlock (hCursor: hDBICur; var iRecords: Longint; pBuf:
  Pointer): DBIResult stdcall;
```

Description

DbiReadBlock reads a specified number of records (starting from the current position of the cursor) into a buffer.

Parameters

hCursor Type: hDBICur (Input)
Specifies the cursor handle to the table.

piRecords Type: pUINT32 (Input/Output)
On input, specifies the number of records to read. On output, pointer to the client variable that receives the number of actual records that were read.

pBuf Type: pBYTE (Output)
Pointer to the client buffer that receives the record data.

Usage

This function is equivalent to doing a loop with DbiGetNextRecord for the specified number in *piRecords*, though it can be considered significantly faster than a DbiGetNextRecord loop.

If filters are active, DbiReadBlock reads only the records that meet filter criteria; all others are skipped. The records are not locked. The number of records read may differ from the number of records requested due to conditions such as end of table.

DbiReadBlock can access data in blocks larger than 64Kb, depending on the size you allocate for the buffer.

Completion state

The variable, *piRecords*, contains the number of actual records read after the function completes. The cursor position is updated according to the actual number of records read.

DbiResult return values

DBIERR_NONE The block of records was successfully read.

DBIERR_INVALIDHNDL The specified cursor handle is invalid or NULL, or *piRecords* is NULL, or pBuf is NULL.

DBIERR_EOF An attempt was made to read beyond the end of the file. The cursor is positioned in the crack at the end of the file. *piRecords* contains the number of records, if any, that were read before the end of file was reached.

See also

DbiWriteBlock, DbiGetNextRecord

C Examples: DbiReadBlock

An example for this function is under development and will be provided in an upcoming Help release.

Delphi Examples: DbiReadBlock

An example for this function is under development and will be provided in an upcoming Help release.

DbiRegenIndex {button C Examples,JI(`>example',`exdbiregenindex')} {button Delphi Examples,JI(`>example',`dexdbiregenindex')}

C syntax

```
DBIResult DBIFN DbiRegenIndex (hDb, [hCursor], [pszTableName],
  [pszDriverType], pszIndexName, pszIndexTagName, iIndexId);
```

Delphi syntax

```
function DbiRegenIndex (hDb: hDBIDb; hCursor: hDBICur; pszTableName: PChar;
  pszDriverType: PChar; pszIndexName: PChar; pszIndexTagName: PChar;
  iIndexId: Word): DBIResult stdcall;
```

Description

DbiRegenIndex regenerates an index to ensure that it is up to date (all records currently in the table are included in the index and are in the index order). It can also be used to pack the index on disk.

Parameters

hDb Type: hDBIDb (Input)
Specifies the database handle associated with the database where the table exists.

hCursor Type: hDBICur (Input)
Specifies the cursor on the table. Optional. If *hCursor* is specified, the operation is performed on the table associated with the cursor. If *hCursor* is NULL, *pszTblName* and *pszDriverType* determine the table to be used.

pszTableName Type: pCHAR (Input)
Pointer to the table name. Optional. If *hCursor* is NULL, *pszTableName* and *pszDriverType* determine the table to be used. If both *pszTableName* and *hCursor* are specified, *pszTableName* is ignored.

For Paradox, FoxPro, and dBASE, if *pszTableName* is a fully qualified name of a table, the *pszDriverType* parameter need not be specified. If the path is not included, the path name is taken from the current directory of the database associated with *hDb*.

pszDriverType Type: pCHAR (Input)
Pointer to the table type. Optional. For Paradox, FoxPro, and dBASE tables, this parameter is required if *pszTableName* has no extension. *pszDriverType* can be one of the following values: szDBASE or szPARADOX.

pszIndexName Type: pCHAR (Input)
Pointer to the name of the index. See rules for naming indexes in the IDXDesc section.

pszIndexTagName Type: pCHAR (Input)
Pointer to the tag name of the index in a .MDX or .CDX file. Used for dBASE and FoxPro tables only. This parameter is ignored if the index given by *pszIndexName* is not a .MDX or .CDX index.

iIndexId Type: UINT16 (Input)
Specifies the index number.

Usage

iIndexId, *pszIndexName*, and *pszIndexTagName* are used in various combinations to specify the index to regenerate.

Important: A maintained index is automatically updated when the table is updated. A non-maintained index must use DbiRegenIndex to update the index after the table is modified before it can be used to access data.

Paradox: The effect of regenerating a maintained index is that it becomes more efficient

and compact. (Frequent updates can fragment an index.)

SQL, Access: SQL and Access indexes cannot be regenerated.

dBASE and FoxPro: DbiRegenIndex is normally used to update a non-maintained dBASE or FoxPro index. However, there may be situations when a maintained index needs to be regenerated. Since a non-production index is maintained only when it is in use, it is not actually maintained at all times. If the index is not up to date, DbiRegenIndex can be used to synchronize the index with the current data.

Prerequisites

The table name must be provided and the index must already exist. When regenerating a maintained index, the table must be opened exclusively. When regenerating a non-maintained index, BDE must be able to obtain a write lock on the table.

DbiResult return values

DBIERR_NONE The index specified by *pszIdxName* was successfully regenerated.

DBIERR_NOSUCHINDEX	The given index (*pszIdxName*) does not exist.
DBIERR_INVALIDPARAM	A cursor was not provided for the table, and the table name is either empty or not provided.
DBIERR_INVALIDHNDL	The specified handle was invalid or NULL.
DBIERR_NEEDEXCLACCESS	A cursor was provided for the table, but it was not opened in exclusive mode when regenerating a maintained index.
DBIERR_FILEBUSY	Exclusive access could not be obtained on table.
DBIERR_FILELOCKED	Write lock could not be obtained on table.
DBIERR_NOTSUPPORTED	A SQL index cannot be regenerated.

See also

DbiRegenIndexes

C Examples: DbiRegenIndex

An example for this function is under development and will be provided in an upcoming Help release.

Delphi Examples: DbiRegenIndex

Regenerate an index to ensure that it is up to date.

This example uses the following input:

```
  fDbiRegenIndex(Table1, 'ByCompany', '', 1);
```

The procedure is:

```
procedure fDbiRegenIndex(Tbl: TTable; IndexName, TagName: string; IndexNum:
  Word);
begin
  Check(DbiRegenIndex(Tbl.DBHandle, nil, PChar(Tbl.TableName), nil,
    PChar(IndexName), PChar(TagName), IndexNum));
end;
```

DbiRegenIndexes {button C Examples,JI(`>example',`exdbiregenindexes')} {button Delphi Examples,JI(`>example',`dexdbiregenindexes')}

C syntax
```
DBIResult DBIFN DbiRegenIndexes (hCursor);
```

Delphi syntax
```
function DbiRegenIndexes (hCursor: hDBICur): DBIResult stdcall;
```

Description
DbiRegenIndexes regenerates all indexes associated with a cursor.

Parameters
hCursor Type: hDBICur (Input)
Specifies the cursor handle for the table to be regenerated.

Usage
A maintained index is automatically updated when the table is updated.

dBASE or FoxPro: All open indexes are regenerated.

Paradox: All maintained and non-maintained indexes are regenerated.

SQL, Access: SQL and Access indexes cannot be regenerated.

Prerequisites
There can be more than one index open on a table. A valid cursor handle must be obtained, the table must be opened exclusively, and the index must already exist.

DbiResult return values

DBIERR_NONE	All of the indexes for the table associated with the specified cursor have been successfully regenerated.
DBIERR_INVALIDHNDL	The specified cursor handle is invalid or NULL.
DBIERR_NEEDEXCLACCESS	The table associated with *hCursor* is opened in open shared mode.
DBIERR_NOTSUPPORTED	SQL indexes cannot be regenerated.

See also
DbiRegenIndex

C Examples: DbiRegenIndexes

An example for this function is under development and will be provided in an upcoming Help release.

Delphi Examples: DbiRegenIndexes

Regenerate all indexes associated with a cursor.

This function regenerates the indexes associated with the Ttable specified in the TblName parameter. This example uses the following input:

```
  fDbiRegenIndexes(BIOLIFE_TABLE);
```

The procedure is:

```
procedure fDbiRegenIndexes(TblName: TTable);
begin
  Check(DbiRegenIndexes(TblName.Handle));
end;
```

DbiRegisterCallBack {button C Examples,JI(`>example',`exdbiregistercallback')} {button Delphi Examples,JI(`>example',`dexdbiregistercallback')}

C syntax

```
DBIResult DBIFN DbiRegisterCallBack (hCursor, ecbType, iClientData,
  iCbBufLen, pCbBuf, pfCb);
```

Delphi syntax

```
function DbiRegisterCallBack (hCursor: hDBICur; ecbType: CBType;
  iClientData: Longint; iCbBufLen: Word; CbBuf: Pointer; pfCb:
  pfDBICallBack): DBIResult stdcall;
```

Description

DbiRegisterCallBack registers a callback function for the client application.

Parameters

hCursor Type: hDBICur (Input)
Specifies the cursor handle to which the callback is being registered. Optional. If *hCursor* is NULL, the callback is registered to the current session.

ecbType Type: CBType (Input)
Specifies the type of callback. *ecbType* can be cbGENPROGRESS, cbBATCHRESULT, cbRESTRUCTURE, cbINPUTREQ, cbTABLECHANGED, cbDELAYEDUPD, or cbDBASELOGIN. (See "Usage" below.)

iClientData Type: UINT32 (Input)
Passthrough data specified by the client. This is used to help the client establish the context of the callback (such as a pointer to a client structure, a window handle, and so on.) This data is passed back to the client as a parameter to the callback function.

iCbBufLen Type: UINT16 (Input)
Specifies the callback buffer length.

pCbBuf Type: pVOID (Input)
Pointer to the buffer where the callback data is to be returned. Points to an instantiated callback descriptor, which varies depending upon the type of callback. For example, the cbGENPROGRESS callback type creates a pointer to the CBPROGRESSDesc structure.

The data that is written to *pCbBuf* is the percentage completed or a message string.

pfCb Type: pfDBICallBack (Input)
Pointer to the desired callback function. Optional. If *pfCb* is NULL, DbiRegisterCallBack unregisters the previously registered callback function.

Usage

Callbacks are used when a client application needs clarification about a given BDE function before completing an operation or to return information to the client. DbiRegisterCallBack allows the client to instruct BDE about what further actions should be taken by BDE upon the occurrence of an event. BDE calls the client-registered function when the pertinent event occurs, and the client responds to the callback by telling BDE what to do with the appropriate return code (cbrABORT, cbrCONTINUE, and so on). Advantages of this mechanism are that clients do not have to check every return code on every function call, and BDE can get a user's response without interrupting the normal client process flow.

Callback function declarations and associated parameter lists, function return types, and callback data types are defined in the file IDAPI.H, which is the client interface to BDE.

All callback functions use the following prototype:

```
  typedef CBRType far *pCBRType;
```

```
typedef CBRType (DBIFN * pfDBICallBack)
(
CBType ecbType,     // Callback type
UINT32 iClientData, // Client callback data
pVOID pCbInfo       // Call back info/Client
Input
);
```

For each different callback type, the *pCbInfo* parameter serves a different purpose:

Callback	Description
cbGENPROGRESS	Informs applications about the progress made during large batch operations.
cbRESTRUCTURE	Supplies information about an impending action and requests a response from the caller.
cbBATCHRESULT	Batch processing results.
cbTABLECHANGED	Notifies user that table has changed.
cbCANCELQRY	Allows user to cancel a Sybase query.
cbINPUTREQ	A BDE driver requests input from user.
cbDBASELOGIN	Enables clients to access encrypted dBASE tables.
cbFIELDRECALC	Field(s) recalculation
cbTRACE	Trace
cbDBLOGIN	Database login
cbDELAYEDUPD	Cached updates callback
cbNBROFCBS	Number of callbacks

Prerequisites

The client application is responsible for the following actions:

- Allocating memory for *pCbBuf*.
- Declaring the callback function with an associated predefined parameter list.

Completion state

If a cursor is supplied, any previous callbacks for the given cursor are overwritten. All callbacks are applicable to the current session only. The callback is valid only while the cursor is open; when the cursor is closed, any cursor-specific callbacks are automatically unregistered. If *hCursor* is NULL, then the callback applies to all cursors in the current session that do not have an explicit callback of their own. Supplying a NULL function pointer unregisters the callback.

DbiResult return values

DBIERR_NONE	The callback was registered successfully.
DBIERR_OBJIMPLICITLYDROPPED	The field name was modified.
DBIERR_OBJMAYBETRUNCATED	The field width was reduced.
DBIERR_VALFIELDMODIFIED	Inserted field in position pointed to by an existing VCHKDesc.
DBIERR_VALIDATEDATE	An existing VCHKDesc was modified.
DBIERR_INVALIDFLDXFORM	The field type was modified.
DBIERR_KEYVIOL	An existing IDXDesc was modified.
DBIERR_NOMEMORY	Insufficient memory was allocated for *pCbBuf*.

See also
DbiGetCallBack, DbiBatchMove, DbiDoRestructure, DbiForceReread

C Examples: DbiRegisterCallBack

An example for this function is under development and will be provided in an upcoming Help release.

Delphi Examples: DbiRegisterCallBack

Example 1: Create a callback that handles the condition of a missing .MDX file for a dBASE file or a missing .CDX file for a FoxPro file:

```
//Function called by DbiRegisterCallBack
function myfunc(ecbType : CBType; iClientData : LongInt;
  pCbInfo : pCBInputDesc): CBRType; stdcall;
const
  READONLY = 'Read Only';
  FAILOPEN = 'Fail Open';  //The default
  OPENANDDETACH = 'Open and Detach';
var
  counter : Integer;
begin
  case ecbtype of
    cbINPUTREQ:  //It's a callback of cbINPUTREQ type
      if (pcbInfo.eCbInputId = cbiMDXMIssing) then begin
        for counter:=0 to (pcbInfo.iCount - 1) do
          if (pcbInfo.acbEntry[counter].szKeyword = OPENANDDETACH) then
  begin
            pcbInfo.iSelection := counter + 1;
            pcbInfo.bSave := False;
            break;
          end;
      end
      else //if
        ShowMessage('Unexpected eCbInputId');
  else  //case
    ShowMessage('Unexpected ecbType')
  end;
end;

//Register the callback and open the table
procedure TForm1.Button2Click(Sender: TObject);
var
  cbinfo : CBInputDesc;
begin
  Session.Open;
  Check(DbiRegisterCallBack(
    nil,                          //Cursor (Optional)
    cbINPUTREQ,                   //Type of Callback
    LongInt(0),                   //Pass-through client data
    sizeof(CBInputDesc),          //Callback buffer len
    cbinfo,                       //Pointer to callback function
    @myfunc                       //Call back fn being registered
    ));
  Table1.Open;
  //Unregister the callback
  Check(DbiRegisterCallBack(nil, cbINPUTREQ, 0,
    sizeof(CBInputDesc), nil , nil));
end;
```

DbiRelPersistTableLock {button C Examples,JI(`>example',`exdbirelpersisttablelock')} {button Delphi Examples,JI(`>example',`dexdbirelpersisttablelock')}

C syntax

```
DBIResult DBIFN DbiRelPersistTableLock (hDb, pszTableName, [pszDriverType]);
```

Delphi syntax

```
function DbiRelPersistTableLock (hDb: hDBIDb; pszTableName: PChar;
  pszDriverType: PChar): DBIResult stdcall;
```

Description

DbiRelPersistTableLock releases the persistent table lock on the specified table for the associated session.

Parameters

hDb Type: hDBIDb (Input)
Specifies the database handle.

pszTableName Type: pCHAR (Input)
Pointer to the name of the table. For Paradox, if *pszTableName* is a fully qualified name of a table, the *pszDriverType* parameter need not be specified. If the path is not included, the path name is taken from the current directory of the database associated with *hDb*.

For SQL databases, this parameter can be a fully qualified name that includes the owner name.

pszDriverType Type: pCHAR (Input)
Pointer to the driver type. Optional. For Paradox tables, this parameter is required if *pszTableName* has no extension. *pszDriverType* must be szPARADOX. This parameter is ignored if the database associated with *hDb* is a SQL database.

Usage

This function is valid only with Paradox and SQL tables, since only Paradox and SQL tables can have persistent locks placed on them.

dBASE, FoxPro, Access: This function is not supported with dBASE, FoxPro, and Access tables.

Completion state

The number of persistent locks on the table is decremented. If this is the last persistent lock on the table, the lock is released.

DbiResult return values

DBIERR_NONE	The lock was released successfully.
DBIERR_INVALIDHNDL	The specified database handle is invalid or NULL.
DBIERR_INVALIDPARAM	The specified table name or the pointer to the table name is NULL.
DBIERR_NOTLOCKED	The specified table does not have a persistent lock placed on it.

See also

DbiAcqPersistTableLock

C Examples: DbiRelPersistTableLock

An example for this function is under development and will be provided in an upcoming Help release.

Delphi Examples: DbiRelPersistTableLock

Place and release persistent lock on the TTable T.

The function AcqAndRelPersistentTableLock, below, acquires a persistent table lock on the table used by the TTable specified in the T parameter. This example uses the following input:

```
  AcqAndRelPersistTableLock(Table1);
```

The procedure is:

```
procedure AcqAndRelPersistTableLock(T: TTable);
var
  Drv: PChar;
begin
  with T do begin
    if (TableType = ttParadox) then
      Drv := StrNew(szParadox)
    else if (TableType = ttdBASE) then
      Drv := StrNew(szdBASE)
    else Drv := nil;
    try
      Check(DbiAcqPersistTableLock(DBHandle, PChar(TableName), Drv));
      Check(DbiRelPersistTableLock(DBHandle, PChar(TableName), Drv));
    finally
      if Assigned(Drv) then StrDispose(Drv);
    end;
  end;
end;
```

DbiRelRecordLock {button C Examples,JI(`>example',`exdbirelrecordlock')} {button Delphi Examples,JI(`>example',`dexdbirelrecordlock')}

C syntax
```
DBIResult DBIFN DbiRelRecordLock (hCursor, bAll);
```

Delphi syntax
```
function DbiRelRecordLock (hCursor: hDBICur; bAll: Bool): DBIResult stdcall;
```

Description
DbiRelRecordLock releases the record lock on either the current record of *hCursor* or all the record locks acquired in the current session.

Parameters
hCursor Type: hDBICur (Input)
Specifies the cursor handle.

bAll Type: BOOL (Input)
Specifies which record locks to release. If set to TRUE, all record locks acquired in the current session are released. If set to FALSE, *hCursor* must be positioned on a record in order to release the lock for that record.

Usage
SQL: Optimistic locks are released by this function. The SQL drivers always perform optimistic record locking; therefore, a record lock request does not explicitly attempt to lock the record on the server.

Completion state
The specified record locks are removed.

DbiResult return values
DBIERR_NONE Locks were successfully released.

DBIERR_INVALIDHNDL	The specified cursor handle is invalid or NULL.
DBIERR_NOTLOCKED	The current record is not locked (this error is returned only when *bAll* is FALSE).
DBIERR_NOCURREC	The cursor is not positioned on a record.

See also
DbiGetNextRecord, DbiGetPriorRecord, DbiGetRecord, DbiGetRelativeRecord, DbiIsRecordLocked

C Examples: DbiRelRecordLock

Release all record locks on the table associated with a cursor.

Note: To release only the one record pointed to by the cursor, set *bAll* to FALSE.

```
DBIResult fDbiRelRecordLock(hDBICur hCur)
{
   DBIResult      rslt;
   BOOL           bAll;
   bAll = TRUE;
   rslt = Chk(DbiRelRecordLock(hCur, bAll));
   return rslt;
}
```

Delphi Examples: DbiRelRecordLock

Release the record lock on either the current record or all the record locks in the current session.

This example uses the following input:

```
  fDbiRelRecordLock(Table1.Handle, True);
```

The procedure is:

```
procedure fDbiRelRecordLock(hTmpHandle:hDBICur; bAll: Boolean);
begin
  Check(DbiRelRecordLock(hTmpHandle, bAll));
end;
```

DbiRelTableLock {button C Examples,JI(`>example',`exdbireltablelock')} {button Delphi Examples,JI(`>example',`dexdbireltablelock')}

C syntax

```
DBIResult DBIFN DbiRelTableLock (hCursor, bAll, eLockType);
```

Delphi syntax

```
function DbiRelTableLock (hCursor: hDBICur; bAll: Bool; eLockType:
  DBILockType): DBIResult stdcall;
```

Description

DbiRelTableLock releases table locks of the specified type associated with the session in which *hCursor* was created.

Parameters

hCursor Type: hDBICur (Input)
Specifies the cursor handle.

bAll Type: BOOL (Input)
Determines which table locks to release. If set to TRUE, all locks on the table associated with *hCursor* are released, and *eLockType* is ignored.

eLockType Type: DBILockType (Input)
Specifies the table lock type. *eLockType* is ignored if *bAll* is TRUE.

For dBASE, FoxPro, and SQL tables, dbiREADLOCK is upgraded to dbiWRITELOCK. In that case, if *eLockType* specifies dbiREADLOCK, the write lock is released.

Usage

Only locks acquired by calling DbiAcqTableLock can be released. A separate call to DbiRelTableLock is required to release each lock acquired by DbiAcqTableLock, if *bAll* is not set to TRUE.

dBASE or FoxPro: See the *eLockType* parameter description.

SQL: See the *eLockType* parameter description.

Prerequisites

There must be an existing table lock of the type specified in *eLockType*. However, an existing table lock is not required if all locks are being released (*bAll* is TRUE).

DbiResult return values

DBIERR_NONE	Locks were successfully released.
DBIERR_INVALIDHNDL	The specified cursor handle is invalid or NULL.
DBIERR_NOTLOCKED	The table is not locked with the specified lock type (this error is returned only when *bAll* is FALSE).

See also

DbiAcqTableLock, DbiIsTableLocked, DbiOpenLockList

eLockType
eLockType can be one of the following values:

***eLockType* value**	**Table lock type**
dbiWRITELOCK	Write lock
dbiREADLOCK	Read lock

C Examples: DbiRelTableLock

Release all locks placed on a table by DbiAcqTableLock.

```
DBIResult fDbiRelTableLock(hDBICur hCur)
{
   DBIResult      rslt;
   rslt = Chk(DbiRelTableLock(hCur, TRUE, NULL));
   return rslt;
}
```

Delphi Examples: DbiRelTableLock

Release all locks placed on a table by DbiAcqTableLock.

Delphi users can use the TTable.UnLockTable method rather than directly calling DbiRelTableLock. This method is defined as: Procedure TTable.UnLockTable(LockType: TLockType); This example uses the following input:

```
  fDbiRelTableLock(Table1, True, dbiWRITELOCK);
```

The procedure is:

```
procedure fDbiRelTableLock(TblName: TTable; All: Boolean; Lock:
  DBILockType);
var
  hNewCur: hDbiCur;
begin
  Check(DbiGetCursorForTable(TblName.DBHandle,
    PChar(TblName.TableName), '', hNewCur));
  Check(DbiRelTableLock(hNewCur, All, Lock));
end;
```

Release all locks on the specified table.

Delphi users can use the TTable.UnlockTable method rather than directly calling DbiRelTableLock. This example uses the following input:

```
  fDbiRelTableLock(Table1);
```

The procedure is:

```
procedure fDbiRelTableLock(TblName: TTable);
begin
  Check(DbiRelTableLock(TblName.Handle, True, dbiWRITELOCK));
end;
```

DbiRenameTable {button C Examples,JI(`>example',`exdbirenametable')} {button Delphi Examples,JI(`>example',`dexdbirenametable')}

C syntax

```
DBIResult DBIFN DbiRenameTable (hDb, pszOldName, [pszDriverType],
  pszNewName);
```

Delphi syntax

```
function DbiRenameTable (hDb: hDBIDb; pszOldName: PChar; pszDriverType:
  PChar; pszNewName: PChar): DBIResult stdcall;
```

Description

DbiRenameTable renames the table given in *pszOldName* and all its resources to the new name specified by *pszNewName*.

Parameters

hDb Type: hDBIDb (Input)
Specifies the database handle.

pszOldName Type: pCHAR (Input)
Pointer to the name of existing table. For Paradox, FoxPro, and dBASE tables only, if *pszOldName* contains an extension, *pszDriverType* is not needed. The source driver type determines the destination driver type.

pszDriverType Type: pCHAR (Input)
Pointer to the table type. Optional. For Paradox, FoxPro, and dBASE tables, this parameter is required if *pszOldName* has no extension. This parameter is ignored if the database associated with *hDb* is a SQL database. *pszTableType* can be one of the following values: szDBASE, szMSACCESS, or szPARADOX.

pszNewName Type: pCHAR (Input)
Pointer to the new name for the table.

Usage

When the table is renamed, other resources are also renamed, depending on the database driver.

Paradox: The following files are renamed:

- The table (.DB extension)
- BLOB files (.MB extension)
- All indexes
- Validity check and referential integrity files (.VAL extension)

If the table is encrypted, the master password must be specified, or the DbiRenameTable call fails. A master table in a referential integrity link, the table cannot be renamed. If it is a detail table and the table is renamed into the same directory, the function automatically maintains the link to its master table. If it is a detail table and the table is renamed into the different directory, referential integrity is dropped. Exclusive access to the master table is required.

dBASE or FoxPro: The following files are renamed:

- The table (.DBF extension)
- BLOB files (.DBT or .FPT extension)
- The production index (.MDX extension)
- The compressed index (.CDX extension)

Access: Access tables do not have supporting files.

SQL: All indexes become associated with the new table name. Some SQL servers do not support DbiRenameTable.

Prerequisites

The client application must have permission to lock the table exclusively.

DbiResult return values

DBIERR_NONE	The table was renamed successfully.
DBIERR_INVALIDHNDL	The specified database handle is invalid or NULL.
DBIERR_NOSUCHTABLE	The source table does not exist.
DBIERR_UNKNOWNTBLTYPE	The driver type is unknown.
DBIERR_NOTSUFFTABLERIGHTS	The client application has insufficient rights to the table (Paradox only).
DBIERR_NOTSUFFFAMILYRIGHTS	The client application has insufficient rights to family members (Paradox only).
DBIERR_LOCKED	The table is already in use.

See also

DbiAddPassword, DbiCopyTable, DbiDeleteTable

C Examples: DbiRenameTable

An example for this function is under development and will be provided in an upcoming Help release.

Delphi Examples: DbiRenameTable

Rename the table to the new table name. If ReOpen is True, reset the table's TableName and reopen the table.

Note: Most Delphi users should use TTable.RenameTable method.

```
procedure fDbiRenameTable(Table: TTable; NewName: string; ReOpen: Boolean);
var
  hDb: hDBIDb;
  Props: CURProps;
begin
  if not Table.Active then
    EDatabaseError.Create('Table must be open to complete operation');
  if not Table.Exclusive then
    EDBEngineError.Create(DBIERR_NEEDEXCLACCESS);
  Check(DbiGetCursorProps(Table.Handle, Props));
  // Get the Database Handle from the table cursor since Table.DBHandle will
  //   be invalid once the table is closed
  Check(DbiGetObjFromObj(hDBIObj(Table.Handle), objDATABASE, hDBIObj(hDb)));
  Table.Close;
  Check(DbiRenameTable(hDb, PChar(Table.TableName), Props.szTableType,
  PChar(NewName)));
  if ReOpen then begin
    Table.TableName := NewName;
    Table.Open;
  end;
end;
```

DbiResetRange {button C Examples,JI(`>example',`exdbiresetrange')} {button Delphi Examples,JI(`>example',`dexdbiresetrange')}

C syntax
```
DBIResult DBIFN DbiResetRange (hCursor);
```

Delphi syntax
```
function DbiResetRange (hCursor: hDBICur): DBIResult stdcall;
```

Description
DbiResetRange removes the specified cursor's limited range previously established by the function DbiSetRange.

Parameters
hCursor Type: hDBICur (Input)
Specifies the cursor handle of the table with the range to be removed.

Usage
DbiResetRange preserves the current position of the cursor.

Prerequisites
The cursor must be opened on an index.

Completion state
The function has no effect on existing filters.

If the cursor was positioned on a valid record before the call, it is left on the same record. If it was positioned on a crack, it is positioned there after the call.

DbiResult return values

DBIERR_NONE	The range was reset successfully.
DBIERR_INVAIDHNDL	hCursor is not valid.
DBIERR_NOASSOCINDEX	The specified table does not have an index open.

See also
DbiSetRange

C Examples: DbiResetRange

Reset the range of a table after using DbiSetRange.

This example removes constraints on a result set.

```
DBIResult fDbiResetRange(hDBICur hCur)
{
   DBIResult      rslt;
   rslt = Chk(DbiResetRange(hCur));
   return rslt;
}
```

Delphi Examples: DbiResetRange

Reset the range of a table after using DbiSetRange.

This example removes constraints on a result set.

```
function fDbiResetRange(Handle: hDBICur): DBIResult;
begin
   Return := DbiResetRange(Handle); { remove range }
   Check(Return);       { raise an exception if that failed }
end;
```

DbiSaveChanges {button C Examples,JI(`>example',`exdbisavechanges')} {button Delphi Examples,JI(`>example',`dexdbisetrange')}

C syntax
```
DBIResult DBIFN DbiSaveChanges (hCursor);
```

Delphi syntax
```
function DbiSaveChanges (hCursor: hDBICur): DBIResult stdcall;
```

Description
DbiSaveChanges forces all updated records associated with *hCursor* to disk.

Parameters
hCursor Type: hDBICur (Input)
Specifies the cursor handle.

Usage
If the table associated with *hCursor* is a temporary table (created with DbiCreateTempTable), DbiSaveChanges saves all buffered changes to disk and makes the table permanent. This table will not be removed when the cursor is closed.

SQL: This function is not supported with SQL tables.

DbiResult return values

DBIERR_NONE	All changes have been saved successfully.
DBIERR_INVALIDHNDL	The specified cursor is invalid or NULL.
DBIERR_NODISKSPACE	The changes could not be saved because there is no disk space available.
DBIERR_NOTSUPPORTED	This function is not supported for SQL tables.

See also
DbiMakePermanent

C Examples: DbiSaveChanges

Save changes to the specified table name.

Save changes to the specified table name. The table must be open on the current session. This example uses the following input:

fDbiSaveChanges(hCursor);

```
DBIResult fDbiSaveChanges (pCHAR TblName)
{
   DBIResult      rslt;
   hDBICur        hTmpCur = 0;
   rslt = Chk(DbiGetObjFromName(objCURSOR, TblName, &hTmpCur));
   if (rslt != DBIERR_NONE)
      return rslt;
   rslt = Chk(DbiSaveChanges(hTmpCur));
   return rslt;
}
```

Delphi Examples: DbiSaveChanges

Save all updated records associated with hTmpHandle to disk.
This example uses the following input

```
  fDbiSaveChanges(Table1.Handle);
```

The procedure is:

```
procedure fDbiSaveChanges(hTmpHandle:hDBICur);
begin
  Check(DbiSaveChanges(hTmpHandle));
end;
```

DbiSetCurrSession {button C Examples,JI(`>example',`exdbisetcurrsession')} {button Delphi Examples,JI(`>example',`dexdbisetcurrsession')}

C syntax
```
DBIResult DBIFN DbiSetCurrSession (hSes);
```

Delphi syntax
```
function DbiSetCurrSession (hSes: hDBISes): DbiResult stdcall;
```

Description
DbiSetCurrSession sets the current session of the client application to the session associated with *hSes*.

Parameters
hSes Type: hDBISes (Input)
Specifies the session handle. If *hSes* is NULL, DbiSetCurrSession sets the current session to the default session.

Completion state
All subsequent operations that do not require an object handle (such as cursor, database, or statement) are associated with this session. Any functions that take an explicit database, query, or cursor handle as an argument are not affected by DbiSetCurrSession. Any resources required by these functions are allocated in the context of the session set by DbiSetCurrSession.

DbiResult return values
DBIERR_NONE The session has been successfully set to the session associated with *hSes*.

DBIERR_INVALIDSESHANDLE The specified session handle is invalid.

See also
DbiGetCurrSession, DbiStartSession, DbiCloseSession, DbiGetSysInfo, DbiGetSesInfo

C Examples: DbiSetCurrSession

An example for this function is under development and will be provided in an upcoming Help release.

Delphi Examples: DbiSetCurrSession

Set the current session.

Note: Most Delphi users should use Sessions (TSessionList) and Session (TSession) to alter the current session.

This example uses the following input:

```
  fDbiSetCurrSession(Session.Handle);
```

The procedure is:

```
procedure fDbiSetCurrSession(hSes: hDBISes);
begin
  Check(DbiSetCurrSession(hSes));
end;
```

DbiSetDateFormat {button C Examples,JI(`>example',`exdbisetdateformat')} {button Delphi Examples,JI(`>example',`dexdbisetdateformat')}

C syntax
```
DBIResult DBIFN DbiSetDateFormat (pfmtDate);
```

Delphi syntax
```
function DbiSetDateFormat (var fmtDate: FMTDate): DBIResult stdcall;
```

Description
DbiSetDateFormat sets the date format for the current session.

Parameters
pfmtDate Type: pFMTDate (Input)
Pointer to the date format structure.

Usage
The date format is used by QBE (Query By Example language) for input and wildcard character matching. It is also used by batch operations (such as DbiDoRestructure and DbiBatchMove) to handle data type coercion between character and date types.

DbiResult return values

DBIERR_NONE	The date format was successfully set.
DBIERR_INVALIDHNDL	The pointer to the date format structure is NULL.
DBIERR_INVALIDPARAM	Data within the date format structure is invalid.

See also
DbiGetDateFormat

C Examples: DbiSetDateFormat

An example for this function is under development and will be provided in an upcoming Help release.

Delphi Examples: DbiSetDateFormat

Set the date format for the current session.

This example uses the following input:

```
  fDbiSetDateFormat;
```

The procedure is:

```
procedure fDbiSetDateFormat;
var
  fDate : FMTDate;
begin
  // Specifies date separator character
  fDate.szDateSeparator := '/'; { }
  // Date format. 0 = MDY, 1 = DMY, 2 = YMD
  fDate.iDateMode := 0;
  // If TRUE, write year as four digits
  fDate.bFourDigitYear := False;
  // On input add 1900 to year if True
  fDate.bYearBiased := False;
  // Month displayed with a leading zero if True
  fDate.bMonthLeadingZero := False;
  //. Day displayed with leading zero if True
  fDate.bDayLeadingZero := False;
  Check(DbiSetDateFormat(fDate));
end;
```

DbiSetDirectory {button C Examples,JI(`>example',`exdbisetdirectory')} {button Delphi Examples,JI(`>example',`dexdbisetdirectory')}

C syntax
```
DBIResult DBIFN DbiSetDirectory (hDb, pszDir);
```

Delphi syntax
```
function DbiSetDirectory (hDb: hDBIDb; pszDir: PChar): DBIResult stdcall;
```

Description
DbiSetDirectory sets the current directory for a standard database.

Parameters
hDb Type: hDBIDb (Input)
Specifies a standard database handle.

pszDir Type: pCHAR (Input)
Pointer to the client buffer specifying the new current directory path. If set to NULL, DbiSetDirectory sets the current directory to the default directory.

Usage
SQL, Access: DbiSetDirectory is not applicable to SQL and Access databases.

Prerequisites
If DbiSetDirectory has not been called, the directory is set to whatever was specified as the working directory in the DBIEnv structure in DbiInit. If *pszDir* is set to NULL, the directory reverts to the default directory. The default directory is the application's start-up directory. If an alias was used to open the database, the path that was specified in the alias is used as the current directory.

Completion state
After setting the directory, any TblList or FileList cursors opened on this handle are restricted to this directory, and any call to DbiOpenTable without a specified path is limited to searching to this directory. Any resources acquired before DbiSetDirectory is called, such as opened tables, are not affected by the change.

DbiResult return values
DBIERR_NONE	The current directory has been successfully set.
DBIERR_NOTSUPPORTED	This function is not supported with a non-standard database.
DBIERR_INVALIDHNDL	The specified database handle is invalid or NULL.

See also
DbiGetDirectory, DbiInit, DbiOpenTable

C Examples: DbiSetDirectory

Set the working directory for the specified database.

If the main directory cannot be set, the function attempts to set a backup directory. This example uses the following input:

fDbiSetDirectory(hDb, "c:\bde\\examples\\tables", "c:\\bde32\\examples\\tables", &Main);

```
DBIResult fDbiSetDirectory(hDBIDb hTmpDb, pCHAR MainDir, pCHAR BackupDir,
  pBOOL Main)
{
   DBIResult rslt;
   rslt = Chk(DbiSetDirectory(hTmpDb, MainDir));
   if (rslt == DBIERR_NONE)
      *Main = TRUE;
   else
   {
     rslt = Chk(DbiSetDirectory(hTmpDb, BackupDir));
     if (rslt == DBIERR_NONE)
        *Main = FALSE;
   }
   return rslt;
}
```

Delphi Examples: DbiSetDirectory

Set the current working directory.
The function *fDbiSetDirectory* for the database specified in the *hdb* parameter to the directory in the *Dir* parameter. This example uses the following input:

```
  fDbiSetDirectory(Database1.Handle, 'C:\Tables');
```

The procedure is:

```
procedure fDbiSetDirectory(hdb: hDbiDb; Dir: string);
begin
  Check(DbiSetDirectory(hdb, PChar(Dir)));
end;
```

DbiSetFieldMap {button C Examples,JI(`>example',`exdbisetfieldmap')} {button Delphi Examples,JI(`>example',`dexdbisetfieldmap')}

C syntax

```
DBIResult DBIFN DbiSetFieldMap (hCur, iFields, pFldDesc);
```

Delphi syntax

```
function DbiSetFieldMap (hCur: hDBICur; iFields: Word; pFldDesc: pFLDDesc):
  DBIResult stdcall;
```

Description

DbiSetFieldMap sets a field map of the table associated with the given cursor.

Parameters

hCur Type: hDBICur (Input)
Specifies the cursor handle.

iFields Type: UINT16 (Input)
Specifies the number of fields to map.

pFldDesc Type: pFLDDesc (Input)
Pointer to an array of FLDDesc structures.

Usage

A field map allows the user to effectively reorder the fields of a table or to drop some of the fields from view. This function does not produce a new cursor, but modifies the existing one. The client application specifies a field map by building an array of field descriptors. The order of field descriptors in the array specifies the order in which the cursor presents the fields.

For Paradox, FoxPro, Access, and dBASE, all data retrieval functions map the returned records as specified in the field description; no type conversions are allowed. When a record is updated in a table with a field map, the unmapped fields are left unchanged. When a record is inserted in a table with a field map, the unmapped fields are set to blank.

Paradox: When a record is inserted in a table with a field map, the unmapped fields are set to blank or set to any defined default value.

Text: Since no description of the fields are available when the text file is created with DbiCreateTable, it is a good practice to set a field map on the cursor that is opened on that text file. The text driver uses this field map to interpret the data types of the fields in that text file. The DbiTranslateRecordStructure call can be used to convert the logical or physical fields of a given driver type (such as Paradox or dBASE) to the physical fields of the text driver. These resulting physical text fields can be used in the DbiSetFieldMap call. When a field map is set on a text table, *iFldType*, *iFldNum*, *iUnits1*, and *iUnits2* must be set correctly in all the field descriptors.

Prerequisites

DbiGetFieldDescs must be called to retrieve the array of field descriptors for the table.

Completion state

The underlying table is not affected. All the original fields still exist; they are simply not visible. (To drop fields in the underlying table, use DbiDoRestructure.) Setting *iFields* to 0 removes any existing field map and allows the underlying fields to become visible again.

DbiResult return values

DBIERR_NONE The field map was set successfully.

DBIERR_NA	The field number in the field descriptor is greater than the number of fields in the table, or the specified field name does not exist. Some drivers return this error if the user tried to set a field map on a table that already has a field map set.

See also

DbiGetFieldDescs

C Examples: DbiSetFieldMap

An example for this function is under development and will be provided in an upcoming Help release.

Delphi Examples: DbiSetFieldMap

Set a field map for the current table.

Note: Most Delphi users should use the Fields Editor of a TTable to set the field mapping.

This example uses the following input:

```
  fDbiSetFieldMap(CustomerTbl, [CustomerTbl.FieldByName('Company'),
    CustomerTbl.FieldByName('City')]);
```

The procedure is:

```
procedure fDbiSetFieldMap(Table: TTable; const Fields: array of TField);
var
  CurrentElement, Elements, FldNum: Integer;
  pFields, pOrigFields, pF, pOF: pFLDDesc;
begin
  Elements := sizeof(Fields) div sizeof(TField);
  pFields := AllocMem(Elements * sizeof(FLDDesc));
  pOrigFields := AllocMem(Table.FieldCount * sizeof(FLDDesc));
  pF := pFields;
  try
    // Get the original field descriptors
    Check(DbiGetFieldDescs(Table.Handle, pOrigFields));
    // Iterate through the original fields and create a pFLDDesc structure
    //    for the new field map structure
    for CurrentElement := 0 to (Elements - 1) do begin
      pOF := pOrigFields;
      for FldNum := 1 to Table.FieldCount do begin
        // Add only the field names that match
        if (StrIComp(PChar(Fields[CurrentElement].FieldName), pOF.szName) =
  0)
        then begin
          // Move the original FLDDesc to the new FLDDesc
          move(pOF^, pF^, sizeof(FLDDesc));
          Inc(pF);
          break;
        end
        else
          Inc(pOF);
      end;
    end;
    Check(DbiSetFieldMap(Table.Handle, Elements, pFields));
  finally
    FreeMem(pFields, Elements * sizeof(FLDDesc));
    FreeMem(pOrigFields, Table.FieldCount * sizeof(FLDDesc));
  end;
end;
```

DbiSetLockRetry {button C Examples,JI(`>example',`exdbisetlockretry')} {button Delphi Examples,JI(`>example',`dexdbisetlockretry')}

C syntax
```
DBIResult DBIFN DbiSetLockRetry (iWait);
```

Delphi syntax
```
function DbiSetLockRetry (iWait: SmallInt): DBIResult stdcall;
```

Description
DbiSetLockRetry sets the table and record lock retry time for the current session.

Parameters
iWait Type: INT16 (Input)
Specifies the lock retry time in seconds. The default setting is five seconds.

Value	Description
<= -1	Any negative value causes infinite retries
= 0	No retry is attempted
>= 1	Number of seconds to retry

Usage
DbiSetLockRetry functions only with Paradox, FoxPro, Access, and dBASE tables. Whenever table or record lock fails, the lock is repeatedly attempted until the retry time expires. If *iWait* is 0, no retry is performed, resulting in the immediate failure of any unsuccessful lock request. The default setting is five seconds. The following functions retry locking if the lock fails:

Record locks:
- DbiGetNextRecord
- DbiGetRelativeRecord
- DbiGetPriorRecord
- DbiGetRecord

Table locks:
- DbiAcqTableLock

(Persistent table locks are not affected.)

The following functions do not retry locking if the lock fails:
- DbiOpenDatabase
- DbiOpenTable
- DbiSetDirectory
- DbiSetPrivateDir

SQL: This function is not supported with SQL tables.

Completion state
The number of retry seconds is set. Whenever a Paradox, FoxPro, Access, or dBASE table or record lock fails, the lock will be attempted until the retry time limit is reached.

DbiResult return values
DBIERR_NONE The lock retry time was successfully set for the session.

See also

DbiGetNextRecord, DbiGetPriorRecord, DbiGetRelativeRecord, DbiGetRecord, DbiAcqTableLock, DbiAcqPersistTableLock, DbiSetPrivateDir, DbiSetDirectory, DbiOpenTable

C Examples: DbiSetLockRetry

An example for this function is under development and will be provided in an upcoming Help release.

Delphi Examples: DbiSetLockRetry

Set the specified session's lock retry time. This example uses the following input:

```
  fDbiSetLockRetry(Session, 100);
```

The procedure is:

```
procedure fDbiSetLockRetry(LockSession: TSession; Wait: Integer);
var
  OriginalSession: TSession;
begin
  // Save the current session
  OriginalSession := Sessions.CurrentSession;
  // Set the current session to the specified session
  Sessions.CurrentSession := LockSession;
  // Set the lock retry time
  Check(DbiSetLockRetry(Wait));
  // Set the current session back to the original session
  Sessions.CurrentSession := OriginalSession;
end;
```

DbiSetNumberFormat {button C Examples,JI(`>example',`exdbisetnumberformat')} {button Delphi Examples,JI(`>example',`dexdbisetnumberformat')}

C syntax
```
DBIResult DBIFN DbiSetNumberFormat (pfmtNumber);
```

Delphi syntax
```
function DbiSetNumberFormat (var fmtNumber: FMTNumber): DBIResult stdcall;
```

Description
DbiSetNumberFormat sets the number format for the current session.

Parameters
pfmtNumber Type: pFMTNumber (Input)
Pointer to the client-allocated FMTNumber structure.

Usage
The number format is used by QBE for input and wildcard character matching. It is also used by batch operations (such as DbiDoRestructure and DbiBatchMove) to handle data type coercion between character and numeric types.

DbiResult return values
DBIERR_NONE	The number format was set successfully.
DBIERR_INVALIDHNDL	The pointer to the number format structure is NULL.
DBIERR_INVALIDPARAM	Data within the number format structure is invalid.

See also
DbiGetNumberFormat

C Examples: DbiSetNumberFormat

An example for this function is under development and will be provided in an upcoming Help release.

Delphi Examples: DbiSetNumberFormat

An example for this function is under development and will be provided in an upcoming Help release.

DbiSetPrivateDir {button C Examples,JI(`>example',`exdbisetprivatedir')} {button Delphi Examples,JI(`>example',`dexdbisetprivatedir')}

C syntax
```
DBIResult DBIFN DbiSetPrivateDir (pszDir);
```

Delphi syntax
```
function DbiSetPrivateDir (pszDir: PChar): DbiResult stdcall;
```

Description
DbiSetPrivateDir sets the private directory for the current session.

Parameters
pszDir Type: pCHAR (Input)
Pointer to the full path name of the new private directory. Optional. If NULL, then the private directory is reset to the default startup directory.

Usage
Although DbiSetPrivateDir is specific to Paradox tables, it has one important use for all drivers: all temporary or auxiliary files are created in this directory by default. If no private directory is specified, then all temporary or auxiliary tables are created in the default startup directory. Examples of functions that may create temporary or auxiliary tables are DbiDoRestructure and DbiBatchMove.

If you want the private directory to be the same as the default working directory, you must explicitly set first the private directory, then set the default directory by using DbiSetDirectory.

Prerequisites
The directory must be available for exclusive access. No other BDE users can access the private directory.

DbiResult return values
DBIERR_NONE The private directory was successfully set.

DBIERR_DIRBUSY The specified directory is currently in use.

See also
DbiGetSesInfo

C Examples: DbiSetPrivateDir

Set the private directory.

If the main directory cannot be set, the function attempts to set a backup directory. This example uses the following input:

fDbiSetPrivateDir("c:\\temp", "c:\temp", &Main);

```
DBIResult fDbiSetPrivateDir(pCHAR MainDir, pCHAR BackupDir, pBOOL Main)
{
   DBIResult rslt;
   rslt = Chk(DbiSetPrivateDir(MainDir));
   if (rslt == DBIERR_NONE)
      *Main = TRUE;
   else
   {
     rslt = Chk(DbiSetPrivateDir(BackupDir));
     if (rslt == DBIERR_NONE)
        *Main = FALSE;
   }
   return rslt;
}
```

Delphi Examples: DbiSetPrivateDir

Set the private directory.

Delphi programs should use the TSession, PrivateDir property rather than using the dbi function directly. This sets the private directory for Paradox tables. For all drivers, all temporary or auxiliary files are created/kept in this directory. This example uses the following input:

```
  fDbiSetPrivateDir('C:\Temp');
```

The procedure is:

```
procedure fDbiSetPrivateDir(Dir: string);
begin
  Check(DbiSetPrivateDir(PChar(Dir)));
end;
```

DbiSetProp {button C Examples,JI(`>example',`exdbisetprop')} {button Delphi Examples,JI(`>example',`dexdbisetprop')}

C syntax

```
DBIResult DBIFN DbiSetProp (hObj, iProp, iPropValue);
```

Delphi syntax

```
function DbiSetProp (hObj: hDBIObj; iProp: Longint; iPropValue: Longint):
  DBIResult stdcall;
```

Description

DbiSetProp sets the specified properties of an object to a given value. See Getting and Setting Properties.

Parameters

hObj Type: hDBIObj (Input)
Specifies the object handle to a system, client, session, driver, database, cursor, or statement object.

iProp Type: UINT32 (Input)
Specifies the property to set.

iPropValue Type: UINT32 (Input)
Specifies the value of the property.

Usage

The specified object does not necessarily have to match the type of property as long as the object is associated with the object type of the property. For example, the property drvDRIVERTYPE assumes an object of type objDRIVER, but because a cursor is derived from a driver, a cursor handle (objCURSOR) could also be specified. See DbiGetObjFromObj for details about associated objects.

Example

To set the translation mode of a cursor to xltNONE (see DbiOpenTable), use:

```
DbiSetProp (hCursor, curXLTMODE, (UINT32) xltNONE);
```

For properties wider than 32-bits, pass a pointer to the property, and cast the pointer to (UINT32).

Example

The following example shows how you can use DbiSetProp to specify your preference for live or canned result sets during query execution. A canned result set is like a snapshot or a copy of the original data selected by the query. In contrast, a live result set is a view of the original data; specifically, if you modify a live result set, the changes are reflected in the original data.

```
DbiSetProp(hSt, stmtLIVENESS, (UINT32) wantLIVE);
```

DbiResult return values

DBIERR_NONE	The property of the object was successfully set.
DBIERR_NOTSUPPORTED	Property is not supported for this object.
DBIERR_INVALIDPARAM	*hObj* is null or invalid.

See also

DbiOpenTable, DbiGetProp

C Examples: DbiSetProp

Execute a SQL statement and return a live cursor (if possible).

Note: If a live cursor cannot be created, the SQL statement will not be executed. This example uses the following input:

fDbiSetProp1("SELECT * FROM 'CUST.DBF'", hDb, &hTmpCur)

```
DBIResult fDbiSetProp1(pCHAR QryStr, hDBIDb hTmpDb, phDBICur phTmpCur)
{
   DBIResult   rslt;
   hDBIStmt    hStmt;

   rslt = Chk(DbiQAlloc(hTmpDb, qrylangSQL, &hStmt));
   if (rslt != DBIERR_NONE)
      return rslt;
   rslt = Chk(DbiQPrepare(hStmt, QryStr));
   if (rslt != DBIERR_NONE)
   {
      Chk(DbiQFree(&hStmt));
      return rslt;
   }
   rslt = Chk(DbiSetProp(hStmt, stmtLIVENESS, (UINT32)wantLIVE));
   if (rslt != DBIERR_NONE)
   {
      Chk(DbiQFree(&hStmt));
      return rslt;
   }
   rslt = Chk(DbiQExec(hStmt, phTmpCur));
   Chk(DbiQFree(&hStmt));
   return rslt;
}
```

Delphi Examples: DbiSetProp

Example1: Enable or disable soft deletes.

Set soft deletes to True or False depending on the Boolean parameter *SoftDelete* in the TTable specified in the *Table* parameter. This example uses the following input:

```
  fDbiSetProp1(AnimalTbl, True);
```

The procedure is:

```
procedure fDbiSetProp1(Table: TTable; SoftDelete: Boolean);
var
  rslt: DBIResult;
  Props: CURProps;
begin
  Check(DbiGetCursorProps(Table.Handle, Props));
  if (Props.szTableType <> szDBASE) then
    raise EDBEngineError.Create(DBIERR_NOTSUPPORTED);
  // Make sure that the property can be set
  rslt := DbiValidateProp(hDBIObj(Table.Handle), curSOFTDELETEON, True);
  if (rslt = DBIERR_NONE) then
    // Set the property
    Check(DbiSetProp(hDBIObj(Table.Handle), curSOFTDELETEON,
  Longint(SoftDelete)))
  else
    raise EDBEngineError.Create(rslt);
end;
```

Example 2: Specify the maximum number of rows to be fetched from an SQL statement.

Set the maximum rows fetched in the parameter *MaxRows*, for the SQL table specified in the *Table* parameter. This example uses the following input:

```
  fDbiSetProp2(IBTable, 100);
```

The procedure is:

```
procedure fDbiSetProp2(Table: TTable; MaxRows: Longint);
var
  rslt: DBIResult;
  DBType: string;
  Len: Word;
begin
  SetLength(DBType, DBIMAXNAMELEN);
  Check(DbiGetProp(hDBIObj(Table.DBHandle), dbDATABASETYPE,
    PChar(DBType), DBIMAXNAMELEN, Len));
  SetLength(DBType, StrLen(PChar(DBType)));
  // Make sure the table type is not dBASE or Paradox (must be SQL based)
  if (DBType = 'STANDARD') then
    raise EDBEngineError.Create(DBIERR_NOTSUPPORTED);
  // Make sure that the property can be set
  rslt := DbiValidateProp(hDBIObj(Table.Handle), curMAXROWS, True);
  if (rslt = DBIERR_NONE) then
    // Set the property
    Check(DbiSetProp(hDBIObj(Table.Handle), curMAXROWS, MaxRows))
  else
    raise EDBEngineError.Create(rslt);
```

```
end;
```

DbiSetRange {button C Examples,JI(`>example',`exdbisetrange')} {button Delphi Examples,JI(`>example',`dexdbisetrange')}

C syntax

```
DBIResult DBIFN DbiSetRange (hCursor, bKeyItself, [iFields1], [iLen1],
  [pKey1], bKey1Incl, iFields2, iLen2, [pKey2], bKey2Incl);
```

Delphi syntax

```
function DbiSetRange (hCursor: hDBICur; bKeyItself: Bool; iFields1: Word;
  iLen1: Word; pKey1: Pointer; bKey1Incl: Bool; iFields2: Word; iLen2: Word;
  pKey2: Pointer; bKey2Incl: Bool): DBIResult stdcall;
```

Description

DbiSetRange constrains the result set to the subset bounded by two keys.

Parameters

hCursor Type: hDBICur (Input)
Specifies the cursor handle.

bKeyItself Type: BOOL (Input)
Defines the key buffer type. If set to TRUE, *pKey1* and *pKey2* contain the keys directly; if set to FALSE, *pKey1* and *pKey2* point to record buffers from which the keys can be extracted.

iFields1 Type: UINT16 (Input)
Specifies the number of fields to be used for composite keys, for the beginning of the range. Optional. The *iFields1* and *iLen1* parameters together indicate how much of the key is to be used for matching. If both are zero, the entire key is used. If a partial match is required on a given field of the key, all the key fields preceding it in the composite key must be included. Only character fields can be matched for a partial key; all other field types must be fully matched.

For partial key matches, *iFields1* must be equal to the number (if any) of key fields preceding the field being partially matched. *iLen1* specifies the number of characters in the partial key to be matched.

iLen1 Type: UINT16 (Input)
Specifies the length into the last field to be used for composite keys. If not zero, the last field to be used must be a character type.

pKey1 Type: pBYTE (Input)
Pointer to the key value or record buffer for the beginning of the range. Optional. If NULL, no low limit is set.

bKey1Incl Type: BOOL (Input)
Specifies whether to include the beginning key value in the range. *bKey1Incl* can be either TRUE or FALSE.

iFields2 Type: UINT16 (Input)
Specifies the number of fields to be used for composite keys, for the end of the range. Optional. The *iFields2* and *iLen2* parameters together indicate how much of the key is to be used for matching. If both are zero, the entire key is used. If a match is required on a given field of the key, all the key fields preceding it in the composite key must also be supplied. Only character fields can be matched for a partial key; all other field types must be fully matched.

For partial key matches, *iFields2* must be equal to the number (if any) of key fields preceding the field being partially matched. *iLen2* specifies the number of characters in the partial key to be matched.

iLen2 Type: UINT16 (Input)
Specifies the length into the last field to be used for composite keys. If not zero, the last field to be used must be a character type.

pKey2 Type: pBYTE (Input)
Pointer to the key value or record buffer for the end of the range. Optional. If NULL, no high limit is set.

bKey2Incl Type: BOOL (Input)
Specifies whether to include the end key value in the range. *bKey2Incl* can be either TRUE or FALSE.

Prerequisites

There must be an active index.

Completion state

DbiSetRange positions the cursor at the beginning of the range, not on the first record in the range.

After this function is called, the cursor allows access only to records in the table that fall within the defined range. Any attempt to reference records outside the range results in a BOF or EOF error condition.

Paradox: DbiGetRecordCount now reflects only the records in the range. DbiGetSeqNo is relative to the beginning of the range, rather than the beginning of the table.

DbiResult return values

DBIERR_NONE	The range was set successfully.
DBIERR_INVALIDHNDL	The specified cursor handle is invalid or NULL.
DBIERR_OUTOFRANGE	(iField iLen) is less than the whole key.
DBIERR_NOASSOCINDEX	The specified cursor does not have an active index.

See also

DbiResetRange, DbiExtractKey, DbiSetToKey, DbiGetRecordCount, DbiGetSeqNo

C Examples: DbiSetRange

Set the range for the specified cursor and return the amount of records in the range.

For this example to operate, the first field of the table must be numeric, such as STOCK.DB. This example uses the following input:

fDbiSetRange(hPXCur, &Count);

```
DBIResult fDbiSetRange(hDBICur hTmpCur, pUINT32 Count)
{
   DBIResult      rslt;
   pBYTE          pMinBuf, pMaxBuf;
   DFLOAT         key_min = 1000.00, key_max = 2000.00;
   CURProps       CurProps;
   rslt = Chk(DbiGetCursorProps(hTmpCur, &CurProps));
   if (rslt != DBIERR_NONE)
      return rslt;
   pMinBuf = (pBYTE)malloc(CurProps.iRecBufSize * sizeof(BYTE));
   if (pMinBuf == NULL)
      return DBIERR_NOMEMORY;
   pMaxBuf = (pBYTE)malloc(CurProps.iRecBufSize * sizeof(BYTE));
   if (pMaxBuf == NULL)
      return DBIERR_NOMEMORY;
   rslt = Chk(DbiPutField(hTmpCur, 1, pMinBuf, (pBYTE)&key_min));
   if (rslt != DBIERR_NONE)
   {
      free(pMinBuf); free(pMaxBuf);
      return rslt;
   }
   rslt = Chk(DbiPutField(hTmpCur, 1, pMaxBuf, (pBYTE)&key_max));
   if (rslt != DBIERR_NONE)
   {
      free(pMinBuf); free(pMaxBuf);
      return rslt;
   }
   rslt = Chk(DbiSetRange(hTmpCur, FALSE, 0, 0, (pBYTE)pMinBuf, FALSE,
                          0, 0, (pBYTE)pMaxBuf, TRUE));
   *Count = 0L;
   Chk(DbiGetRecordCount(hTmpCur, Count));
   free(pMinBuf); free(pMaxBuf);
   return rslt;
}
```

Delphi Examples: DbiSetRange

Set the range for the specified cursor and return the amount of records in the range.

Delphi programs should call the SetRange, ApplyRange, ResetRange methods of a TTable. This example sets the range for the specified cursor and returns the amount of records in the range. For this example to operate, the first field of the table must be numeric. such as, STOCK.DB.

This example uses the following input:

```
  fDbiSetRange(Table1.Handle, Count);
```

The procedure is:

```
procedure fDbiSetRange(hTmpCur: hDBICur; var Count: LongInt);
var
   pMinBuf, pMaxBuf: PByte;
   key_min, key_max: double;
   CurProp: CURProps;
begin
  pMinBuf := nil;
  pMaxBuf := nil;
  key_min := 1000.00;
  key_max := 2000.00;

  Check(DbiGetCursorProps(hTmpCur, CurProp));

  GetMem(pMinBuf,CurProp.iRecBufSize);
  if (pMinBuf = nil) then
    Check(DBIERR_NOMEMORY);

  GetMem(pMaxBuf,CurProp.iRecBufSize);
  if (pMaxBuf = nil) then
    Check(DBIERR_NOMEMORY);
  try
    Check(DbiPutField(hTmpCur, 1, pMinBuf, @key_min));
    Check(DbiPutField(hTmpCur, 1, pMaxBuf, @key_max));

    Check(DbiSetRange(hTmpCur, False, 0, 0, pMinBuf, False,
      0, 0, pMaxBuf, True));

    // Set the return count for the number of records in the limited range
    Count := 0;
    Check(DbiGetRecordCount(hTmpCur, Count));

    Check(DbiResetRange(hTmpCur));
  finally
    FreeMem(pMinBuf);
    FreeMem(pMaxBuf);
  end;
end;
```

DbiSetTimeFormat {button C Examples,JI(`>example',`exdbisettimeformat')} {button Delphi Examples,JI(`>example',`dexdbisettimeformat')}

C syntax
```
DBIResult DBIFN DbiSetTimeFormat (pfmtTime);
```

Delphi syntax
```
function DbiSetTimeFormat (var fmtTime: FMTTime): DBIResult stdcall;
```

Description
DbiSetTimeFormat sets the time format for the current session.

Parameters
pfmtTime Type: pFMTTime (Input)
Pointer to the client-allocated FMTTime structure.

Usage
The time format is used by QBE for input and wildcard character matching. It is also used by batch operations (such as DbiDoRestructure and DbiBatchMove) to handle data type coercion between character and time or datetime types.

DbiResult return values

DBIERR_NONE	The time format was successfully set.
DBIERR_INVALIDHNDL	The pointer to the time format structure is NULL.
DBIERR_INVALIDPARAM	Data within the time format structure is invalid.

See also
DbiGetTimeFormat

C Examples: DbiSetTimeFormat

An example for this function is under development and will be provided in an upcoming Help release.

Delphi Examples: DbiSetTimeFormat

An example for this function is under development and will be provided in an upcoming Help release.

DbiSetToBegin {button C Examples,JI(`>example',`exdbisettobegin')} {button Delphi Examples,JI(`>example',`dexdbisettobegin')}

C syntax
```
DBIResult DBIFN DbiSetToBegin (hCursor);
```

Delphi syntax
```
function DbiSetToBegin (hCursor: hDBICur): DBIResult stdcall;
```

Description
DbiSetToBegin positions the cursor to the beginning of the result set.

Parameters
hCursor Type: hDBICur (Input)
Specifies the cursor handle.

Usage
This function is used to reposition the cursor to the beginning of the result set. DbiGetNextRecord or DbiGetRelativeRecord can then be called to position the cursor on the first valid record of the result set.

Completion state
The cursor is positioned on the crack before the first record. There is no current record after DbiSetToBegin completes. (DbiGetRecord returns DBIERR_BOF.)

DbiResult return values
DBIERR_NONE The cursor was successfully set to BOF.

DBIERR_INVALIDHNDL The specified cursor handle is invalid or NULL.

See also
DbiGetNextRecord, DbiGetPriorRecord, DbiGetRelativeRecord, DbiSetToEnd, DbiSetToCursor

C Examples: DbiSetToBegin

Position the cursor at the beginning of the table.

The beginning of the table means the crack before the first record.

```
DBIResult fDbiSetToBegin(hDBICur hCur)
{
   DBIResult      rslt;
   rslt = Chk(DbiSetToBegin(hCur));
   return rslt;
}
```

Delphi Examples: DbiSetToBegin

Position the cursor at the beginning of the table.

Delphi programs should use the First method from a TDataset object. This method positions the cursor at the beginning of the result set.

This example uses the following input:

```
  fDbiSetToBegin(hCur);
```

The procedure is:

```
procedure fDbiSetToBegin(hTmpCur: hDbiCur);
begin
  Check(DbiSetToBegin(hTmpCur));
end;
```

DbiSetToBookMark {button C Examples,JI(`>example',`exdbisettobookmark')} {button Delphi Examples,JI(`>example',`dexdbisettobookmark')}

C syntax

```
DBIResult DBIFN DbiSetToBookMark (hCur, pBookMark);
```

Delphi syntax

```
function DbiSetToBookMark (hCur: hDBICur; pBookMark: Pointer): DBIResult
  stdcall;
```

Description

DbiSetToBookMark positions the cursor to the position saved in the specified bookmark.

Parameters

hCur Type: hDBICur (Input)
Specifies the cursor handle. *hCur* must be compatible with the cursor used when the bookmark was obtained.

pBookMark Type: pBYTE (Input)
Pointer to the bookmark. The bookmark is obtained by a prior call to DbiGetBookMark.

Usage

This function is used to position the cursor to a saved position. To determine if the bookmark is stable, call DbiGetCursorProps and examine the bBookMarkStable property.

Prerequisites

DbiGetBookMark must have been called to retrieve a valid bookmark. The supplied cursor can be different from the one used to retrieve the bookmark information, but the cursor must be opened on the same table, with the same index order, if any.

Note: DbiSwitchToIndex may make bookmarks obtained under a different index order unusable with the new order.

Completion state

The cursor is positioned at the bookmark location. If the record pointed to by the bookmark has been deleted, the cursor is positioned on a crack where the original record was.

Note: If the bookmark is unstable, the cursor may be in an unexpected position.

DbiResult return values

DBIERR_NONE	The call was successful; however, the position may not be the expected one if the record has been deleted, or if the bookmark was unstable.
DBIERR_INVALIDHNDL	The specified cursor handle is invalid or NULL, or the pointer to the bookmark is NULL, or the specified bookmark is NULL.
DBIERR_INVALIDBOOKMARK	The specified bookmark is not from the same table, or the bookmark is corrupt.

See also

DbiOpenTable, DbiGetCursorProps, DbiGetBookMark, DbiCompareBookMarks

C Examples: DbiSetToBookMark

Set the cursor to the bookmark position.

If the bookmark is unstable, the cursor will not be moved. This example uses the following input:

fDbiSetToBookMark(hPXCur, pBook);

```
DBIResult fDbiSetToBookMark(hDBICur hCur, pBYTE pBookMark)
{
   DBIResult   rslt;
   CURProps    CurProps;
   rslt = Chk(DbiGetCursorProps(hCur, &CurProps));
   if (rslt != DBIERR_NONE)
      return rslt;
   if(CurProps.bBookMarkStable != TRUE)
      return DBIERR_INVALIDBOOKMARK;
   rslt = Chk(DbiSetToBookMark(hCur, pBookMark));
   return rslt;
}
```

Delphi Examples: DbiSetToBookMark

Set the cursor to the bookmark position:

Delphi users should use the GoToBookmark method associated with descendents of TDataSet including TTable, TQuery, and TStoredProc rather than directly calling DbiSetToBookmark. This method is defined as:

```
procedure GotoBookmark(Bookmark: TBookmark);
```

The following code moves the the cursor to the record within Table1 to the corresponding bookmark obtained through a call to GetBookmark.

```
Table1.GoToBookmark(SetBookMark);
```

DbiSetToCursor{button C Examples,JI(`>example',`exdbisettocursor')} {button Delphi Examples,JI(`>example',`dexdbisettocursor')}

C syntax
```
DBIResult DBIFN DbiSetToCursor (hDest, hSrc);
```

Delphi syntax
```
function DbiSetToCursor (hDest: hDBICur; hSrc: hDBICur): DBIResult stdcall;
```

Description
DbiSetToCursor sets the position of one cursor (the destination cursor) to the position of the source cursor.

Parameters
hDest Type: hDBICur (Input)
Specifies the destination cursor handle.

hSrc Type: hDBICur (Input)
Specifies the source cursor handle.

Usage
This function synchronizes the position of two cursors on the same table.

Prerequisites
Source and destination cursors must be opened on the same table in the same session, and both must be valid. If both cursors are opened on a single table, they do not have to have the same current index. The source cursor must have a current record if the index order is different.

Completion state
After DbiSetToCursor executes, the destination cursor is positioned on the same record as the source cursor. They remain independent of each other, they do not track each other.

DbiResult return values

DBIERR_NONE	The destination cursor was successfully set to the record of the source cursor.
DBIERR_INVALIDHNDL	The specified source cursor or destination cursor is invalid or NULL.
DBIERR_NOCURRREC	The source cursor has no current record.
DBIERR_NOTSAMESESSION	The source and destination cursors are not opened in the same session.

See also
DbiGetBookMark, DbiSetToBookMark, DbiCloneCursor, DbiOpenTable

C Examples: DbiSetToCursor

An example for this function is under development and will be provided in an upcoming Help release.

Delphi Examples: DbiSetToCursor

Set the position of the destination cursor to the position of the source cursor

Delphi users should use the GoToCurrent method associated the TTable component rather than directly calling DbiSetToCursor. This method is defined as:

```
  procedure GoToCurrent(Table: TTable);
```

The following is an example:

```
Table1.GotoCurrent(Table2);
```

DbiSetToEnd {button C Examples,JI(`>example',`exdbisettoend')} {button Delphi Examples,JI(`>example',`dexdbisettoend')}

C syntax
```
DBIResult DBIFN DbiSetToEnd (hCursor);
```

Delphi syntax
```
function DbiSetToEnd (hCursor: hDBICur): DBIResult stdcall;
```

Description
DbiSetToEnd positions the cursor at the end of the result set.

Parameters
hCursor Type: hDBICur (Input)
Specifies the cursor handle.

Usage
This function is used to reposition the cursor at the end of the result set. DbiGetPriorRecord or DbiGetRelativeRecord can be called to position the cursor on the last valid record of the result set.

Completion state
The cursor is positioned on the crack after the end of the result set. There is no current record after DbiSetToEnd completes. (DbiGetRecord returns DBIERR_EOF.)

DbiResult return values
DBIERR_NONE The cursor was successfully set to the EOF position.

DBIERR_INVALIDHNDL The specified cursor handle is invalid or NULL.

See also
DbiSetToBegin, DbiGetNextRecord, DbiGetPriorRecord, DbiGetRelativeRecord, DbiSetToCursor

C Examples: DbiSetToEnd

Move the cursor to the end of the table.

The end of the table means the crack after the last record.

```
DBIResult fDbiSetToEnd(hDBICur hCur)
{
   DBIResult      rslt;
   rslt = Chk(DbiSetToEnd(hCur));
   return rslt;
}
```

Delphi Examples: DbiSetToEnd

Position the cursor at the end of the result set.
Delphi programs should use the Last method from a TDataset object.

This example uses the following input:

```
  fDbiSetToEnd(hCur);
```

The procedure is:

```
procedure fDbiSetToEnd(hTmpCur: hDbiCur);
begin
  Check(DbiSetToEnd(hTmpCur));
end;
```

DbiSetToKey {button C Examples,JI(`>example',`exdbisettokey')} {button Delphi Examples,JI(`>example',`dexdbisettokey')}

C syntax

```
DBIResult DBIFN DbiSetToKey (hCursor, eSearchCond, bDirectKey, [iFields],
  [iLen], pBuf);
```

Delphi syntax

```
function DbiSetToKey (hCursor: hDBICur; eSearchCond: DBISearchCond;
  bDirectKey: Bool; iFields: Word; iLen: Word; pBuff: Pointer): DBIResult
  stdcall;
```

Description

DbiSetToKey positions an ordered cursor based on the given key value.

Parameters

hCursor Type: hDBICur (Input)
Specifies the cursor handle.

eSearchCond Type: DBISearchCond (Input)
Specifies the search condition: keySEARCHEQ, keySEARCHGT, or keySEARCHGEQ.

bDirectKey Type: BOOL (Input)
Specifies whether the key is supplied directly in *pBuff* or not. If set to TRUE, *pBuf* specifies the pointer to the key in physical format; if set to FALSE, *pBuf* specifies the pointer to the record buffer.

iFields Type: UINT16 (Input)
Specifies the number of complete fields to be used for composite keys. Optional. If *iFields* and *iLen* are both 0, the entire key is used.

iLen Type: UINT16 (Input)
Specifies the length into the last field to be used for composite keys. If not zero, the last field to be used must be a character type.

pBuf Type: pBYTE (Input)
Pointer to either the record buffer or the key itself, determined by *bDirectKey*.

Usage

If no index is currently associated with the cursor, an error is generated and no cursor movement occurs.

There are three possible search conditions: keySEARCHEQ, keySEARCHGT, and keySEARCHGEQ. Searches always result in the cursor being positioned on the crack before the record of the specified key value. Assuming all the arguments are specified correctly, only the (=) search condition can return a DBIERR_RECNOTFOUND error.

(> or >=) always succeeds.

You can specify the key either by setting the key fields in a record buffer and supplying the record buffer or by specifying the key buffer directly as a string of bytes. To construct the key buffer, use DbiExtractKey.

The *iFields* and *iLen* parameters together indicate how much of the key is to be used for matching. If both are 0, the entire key is used. If a partial match is required on a given field of the key, all the key fields preceding it in the composite key must also be specified for match. Only character fields can be matched for a partial key; all other field types must be fully matched.

Prerequisites

A cursor handle must be ordered using an index.

Completion state

A search using keySEARCHEQ or keySEARCHGEQ positions the cursor on the crack just prior to the specified key; using keySEARCHGT positions the cursor on the crack just after the specified key.

DbiResult return values

DBIERR_NONE	The record was successfully found.
DBIERR_NOASSOCINDEX	There is no index to search on.
DBIERR_INVALIDPARAM	One of the specified parameters is invalid (for example, iLen is invalid for the current index).
DBIERR_RECNOTFOUND	No record matches the key value.

See also

DbiSetRange, DbiSwitchToIndex, DbiSetToBookMark, DbiGetNextRecord, DbiGetPriorRecord

C Examples: DbiSetToKey

An example for this function is under development and will be provided in an upcoming Help release.

Delphi Examples: DbiSetToKey

Find the specified value in the table and return the record and record buffer

This example works with the STOCK.DB table open on the primary key and uses the following input:

```
  fDbiSetToKey(hCur, pRecBuf);
```

The function is:

```
function fDbiSetToKey(hTmpCur: hDBICur; pTmpRecBuf: PByte): Longint;
var
  key: Double;
  RecProp: RecProps;
begin
  key:= 1330.00;
  Check(DbiInitRecord(hTmpCur, pTmpRecBuf));
  Check(DbiPutField(hTmpCur, 1, pTmpRecBuf, @key));
  Check(DbiSetToKey(hTmpCur, keySEARCHEQ, False, 0, 0, pTmpRecBuf));
  Check(DbiGetNextRecord(hTmpCur, dbiNoLock, pTmpRecBuf, @RecProp));
  Result := RecProp.iSeqNum;
end;
```

DbiSetToRecordNo {button C Examples,JI(`>example',`exdbisettorecordno')} {button Delphi Examples,JI(`>example',`dexdbisettorecordno')}

C syntax

```
DBIResult DBIFN DbiSetToRecordNo (hCursor, iRecNo);
```

Delphi syntax

```
function DbiSetToRecordNo (hCursor: hDBICur; iRecNo: Longint): DBIResult
  stdcall;
```

Description

DbiSetToRecordNo positions the cursor to the given physical record number.

Parameters

hCursor Type: hDBICur (Input)
Specifies the cursor handle.

iRecNo Type: UINT32 (Input)
Specifies the physical record number.

Usage

This function is currently valid only with dBASE and FoxPro tables. The physical record number can be retrieved from the *iPhyRecNum* field of the RECProps structure in calls to DbiGetRecord, DbiGetNextRecord, DbiGetPriorRecord, or DbiGetRelativeRecord.

If the given record number is beyond the valid range for the cursor, the cursor is set to the beginning or end of the file (BOF/EOF).

DbiResult return values

DBIERR_NONE	The cursor was successfully set to the record specified by *iRecNo*.
DBIERR_INVALIDHNDL	The specified cursor handle is invalid or NULL.
DBIERR_BOF	The specified record number is zero.
DBIERR_EOF	The specified record number is greater than the number of records in the table.
DBIERR_NOTSUPPORTED	This function is not supported for Paradox and SQL tables.

See also

DbiSetToSeqNo

C Examples: DbiSetToRecordNo

An example for this function is under development and will be provided in an upcoming Help release.

Delphi Examples: DbiSetToRecordNo

Position the cursor to the given physical record number.

Valid only for dBASE and FoxPro tables. The function fDbiSetToRecordNo, below, positions the cursor in the TTable specified in the *Tbl* parameter to the record number specified in *RecordNum*. Call the TTable component's Resync method after repositioning the record pointer with DbiSetToRecordNo to synchronize the TTable with the underlying dataset.

This example uses the following input:

```
  fDbiSetToRecordNum(DBASEANIMALS, 20);
```

The procedure is:

```
procedure fDbiSetToRecordNo(Tbl: TTable; RecordNum: LongInt);
var
  rslt: dbiResult;
begin
  rslt:= DbiSetToRecordNo(Tbl.handle, RecordNum);
  if (rslt <> DBIERR_NONE) then begin
    if (rslt = DBIERR_EOF) then
      tbl.last;
    if (rslt = DBIERR_BOF) then
      tbl.first;
  end;
  Tbl.Resync([]);
end;
```

DbiSetToSeqNo {button C Examples,JI(`>example',`exdbisettoseqno')} {button Delphi Examples,JI(`>example',`dexdbisettoseqno')}

C syntax

```
DBIResult DBIFN DbiSetToSeqNo (hCursor, iSeqNo);
```

Delphi syntax

```
function DbiSetToSeqNo (hCursor: hDBICur; iSeqNo: Longint): DBIResult
  stdcall;
```

Description

DbiSetToSeqNo positions the cursor to the specified sequence number of a table. Currently supported by Paradox only.

Parameters

hCursor Type: hDBICur (Input)
Specifies the cursor handle.

iSeqNo Type: UINT32 (Input)
Specifies the logical record number.

Usage

This function is currently valid only with Paradox tables. The sequence number can be retrieved by calling DbiGetSeqNo or from the *iSeqNo* field of the RECProps structure in calls to DbiGetRecord, DbiGetNextRecord, DbiGetPriorRecord, or DbiGetRelativeRecord.

A sequence number is the position of a record in the result set associated with *hCursor*. If the given sequence number is beyond the valid sequence number for the cursor, the cursor is set to the beginning or end of the file (BOF/EOF). For example, if the table is empty, this function leaves the cursor positioned at BOF and returns DBIERR_BOF. If the table is not empty and the user attempts to position the cursor beyond a valid sequence number, the cursor is set to EOF, and DBIERR_EOF is returned.

Note: The sequence number for a given record is not stable. If a record is inserted or deleted before the given index order, the sequence number for the record changes.

DbiResult return values

DBIERR_NONE	The Paradox cursor was successfully set to the sequence number specified by *iSeqNo*.
DBIERR_INVALIDHNDL	The specified cursor handle is invalid or NULL.
DBIERR_EOF	The specified record number is greater than the number of records in the table.
DBIERR_BOF	The specified record number is zero.
DBIERR_NOTSUPPORTED	This function is not supported for SQL or dBASE drivers.

See also

DbiGetSeqNo, DbiSetToRecordNo

C Examples: DbiSetToSeqNo

An example for this function is under development and will be provided in an upcoming Help release.

Delphi Examples: DbiSetToSeqNo

Position the cursor to the specified sequence number of the table.

If the table is not a Paradox type, the cursor is not moved. Call the TTable component's Resync method after repositioning the record pointer with DbiSetToRecordNo to synchronize the TTable with the underlying dataset.

This example uses the following input:

```
  fDbiSetToSeqNo(Table1, 40);
```

The procedure is:

```
procedure fDbiSetToSeqNo(var Tbl: TTable; RecNum: Longint);
var
  Props: CurProps;
begin
  Check(DbiGetCursorProps(Tbl.Handle, Props));
  if (StrComp(Props.szTableType, szPARADOX) = 0) then
    Check(DbiSetToSeqNo(Tbl.Handle, RecNum));
  Tbl.Resync([]);
end;
```

DbiSortTable {button C Examples,JI(`>example',`exdbisorttable')} {button Delphi Examples,JI(`>example',`dexdbisorttable')}

C syntax

```
DBIResult DBIFN DbiSortTable (hDb, pszTableName, pszDriverType, hSrcCur,
  pszSortedName, phSortedCur, hDstCur, iSortFields, piFieldNum,
  [pbCaseInsensitive], [pSortOrder], [*ppfsortFn], bRemoveDups,
  [hDuplicatesCur], [plRecsSort]);
```

Delphi syntax

```
function DbiSortTable (hDb: hDBIDb; pszTableName: PChar; pszDriverType:
  PChar; hSrcCur: hDBICur; pszSortedName: PChar; phSortedCur: phDBICur;
  hDstCur: hDBICur; iSortFields: Word; piFieldNum: PWord; pbCaseInsensitive:
  PBool; pSortOrder: pSORTOrder; ppfSortFn: ppfSORTCompFn; bRemoveDups: Bool;
  hDuplicatesCur: hDBICur; var lRecsSort: Longint): DBIResult stdcall;
```

Description

DbiSortTable sorts an opened or closed table, either into itself or into a destination table. There are options to remove duplicates, to enable case-insensitive sorts and special sort functions, and to control the number of records sorted.

Parameters

hDb Type: hDBIDb (Input)
Optional. Specifies the database handle when *pszTableName* and *pszDriverType* are used to identify the source table (not used when *hSrcCur* is supplied). Must be a valid database handle.

pszTableName Type: pCHAR (Input)
Optional. Pointer to the table name. Must be a defined table name and the table must exist. If *hDb*, *pszTableName,* and *pszTableType* are supplied, *hSrcCur* should be NULL. A valid extension may be specified.

pszDriverType Type: pCHAR (Input)
Optional. Supplied only when *hDb* and *pszTableName* are supplied. Pointer to the driver type. Must be a defined driver type.

hSrcCur Type: hDBICur (Input)
Optional. This parameter is supplied when an opened source table is to be sorted to a destination table, as specified in *pszSortedName*. When the table is to be sorted into itself, *hDb*, *pszTableName*, and *pszDriverType* must be used to identify the table instead of *hSrcCur*.

pszSortedName Type: pCHAR (Input)
Optional. Pointer to the file name to be used as the sorted destination table. The table must be closed. The extension must match that of the source table. (To specify a destination table of a different driver type, *hDstCur* must be used.) If this parameter, *phSortedCur*, and *hDstCur* are all NULL, the source table is sorted into itself.

phSortedCur Type: phDBICur (Output)
Optional. Pointer to a cursor handle on the sorted destination table, with the name specified by *pszSortedName*. If NULL, the cursor handle is not returned.

hDstCur Type: hDBICur (Input)
Optional. Used instead of *pszSortedName* to specify the sorted destination table. In this case, the destination table is already open, and the cursor handle is specified. If this parameter and *phSortedName* are NULL, the source table is sorted into itself.

iSortFields Type: UINT16 (Input)
Specifies the number of sort fields to be used.

piFieldNum Type: pUINT16 (Input)
Pointer to an array of the field numbers on which to sort. The number of elements in the array must equal the number specified in *iSortFields*.

pbCaseInsensitive Type: pBOOL (Input)
Optional. Pointer to an array of values indicating whether the sort is to be case-insensitive for each sort field. TRUE specifies case-insensitive. The number of elements in the array must equal the number specified in *iSortFields*.

If a NULL pointer is given, the default is case-sensitive. Only text fields are affected.

pSortOrder Type: pSORTOrder (Input)
Optional. Pointer to an array of the sort order for each field, either ascending or descending. If a NULL pointer is given, the order is ascending. The number of elements in the array must equal the number specified in *iSortFields*.

**ppfsortFn* Type: pfSORTCompFn (Input)
Optional. Pointer to an array of pointers to client-supplied compare functions. The number of elements in the array must be equal to the number specified in *iSortFields*.

bRemoveDups Type: BOOL (Input)
Specifies whether duplicates are to be removed during sorting or not. If TRUE, duplicates are removed from the destination table. Duplicates may be written to a table associated with *hDuplicatesCur*.

hDuplicatesCur Type: hDBICur (Input)
Optional. If specified, duplicates removed from the table are placed in a Duplicates table associated with the specified cursor. The structure of this table must be the same as the source table.

plRecsSort Type: pUINT32 (Input/Output)
Optional. Used only when the source table is identified by *hSrcCur*. On input, pointer to the number of records to sort, from the current position of the source table cursor. On output, pointer to the client variable that receives the actual number of records sorted into the destination table.

Usage

As the table is sorted, the records are physically ordered according to the specified sort criteria. Source and destination tables can be of different driver types; if so, the destination table must be specified by *hDstCur*.

Paradox: A Paradox table with a primary key cannot be sorted into itself. Autoincrement fields cannot be sorted.

SQL: DbiSortTable is not supported with SQL tables as the destination.

Completion state

The records in the destination table are ordered according to the sort criteria. If *plRecSort* is specified, only *plRecSort* records are sorted, starting from the current position in the table, otherwise the whole table is sorted.

DbiResult return values

DBIERR_NONE	The sort was successful.
DBIERR_INVALIDHNDL	The specified database handle is invalid or NULL.
DBIERR_INVALIDFILENAME	The source table name was not provided.
DBIERR_UNKNOWNTBLTYPE	The source driver type was not provided.
DBIERR_INVALIDPARAM	The specified number of sort fields is invalid.
DBIERR_NOTSUPPORTED	This function is not supported for sort to self on a Paradox table with a primary index.

See also
DbiBatchMove, DbiCreateTable, DbiDoRestructure, DbiCopyTable

C Examples: DbiSortTable

An example for this function is under development and will be provided in an upcoming Help release.

Delphi Examples: DbiSortTable

Sort the source table into the destination table on the given field.

This example uses the following input:

```
  fDbiSortTable(CustomerTbl, CustomerTbl2,
  CustomerTbl.FieldByName('COMPANY'));
```

The function is:

```
function fDbiSortTable(SrcTbl, DestTbl: TTable; SortField: TField): Longint;
var
  Field: Word;
  CaseIns: Boolean;
  Recs: Longint;
begin
  Recs := SrcTbl.RecordCount;
  CaseIns := True;
  Field := SortField.Index + 1;
  if not DestTbl.Active then
    raise EDatabaseError.Create('Cannot complete operation with ' +
      'destination table closed');
  Check(DbiSortTable(SrcTbl.DBHandle, nil, nil, SrcTbl.Handle, nil, nil,
    DestTbl.Handle, 1, @Field, @CaseIns, nil, nil, False, nil, Recs));
  Result := Recs;
end;
```

DbiStartSession {button C Examples,JI(`>example',`exdbistartsession')} {button Delphi Examples,JI(`>example',`dexdbistartsession')}

C syntax

```
DBIResult DBIFN DbiStartSession ([pszName], phSes, [pNetDir]);
```

Delphi syntax

```
function DbiStartSession (pszName: PChar; var hSes: hDBISes; pNetDir:
  PChar): DbiResult stdcall;
```

Description

DbiStartSession starts a new session for the client application.

Parameters

pszName Type: pCHAR (Input)
Pointer to the session name. Allows you to name the newly created session; if NULL, BDE names the session. Optional.

phSes Type: phDBISes (Output)
Pointer to the session handle. Used to identify the session.

pNetDir Type: pCHAR (Input)
Pointer to the network file directory for the session. This directory is used for Paradox locking. Use of this pointer allows you to have different NETDIRs for distinct sessions.

Usage

Use DbiStartSession to create different concurrency schemes.

Completion state

DbiStartSession makes the new session the current session.

DbiResult return values

DBIERR_NONE	The session was successfully started.
DBIERR_INVALIDHNDL	phSes is NULL.
DBIERR_SESSIONSLIMIT	The maximum number of sessions are open.

See also

DbiSetCurrSession, DbiCloseSession

C Examples: DbiStartSession

An example for this function is under development and will be provided in an upcoming Help release.

Delphi Examples: DbiStartSession

Example 1: Start a new session for the client application.

Delphi programs can use the TSession object in the component library.

```
procedure fDbiStartSession(pName: string; var hSes: hDBISes; pNetDir:
  string);
begin
  Check(DbiStartSession(PChar(pName), hSes, PChar(pNetDir)));
end;
```

Example 2: Create a new session and return the session number.

Most Delphi users can use TSession.Open, TSessionList.OpenSession or the TSession component.

This example uses the following input:

```
  SesNo := fDbiStartSession('NewSession', hSes, 'C:\Netdir');
```

The function is:

```
function fDbiStartSession(pName: string; var hSes: hDBISes; pNetDir:
  string): Word;
var
  Ses: SESInfo;
begin
  Check(DbiStartSession(PChar(pName), hSes, PChar(pNetDir)));
  Check(DbiGetSesInfo(Ses));
  Result := Ses.iSession;
end;
```

DbiSwitchToIndex {button C Examples,JI(`>example',`exdbiswitchtoindex')} {button Delphi Examples,JI(`>example',`dexdbiswitchtoindex')}

C syntax

```
DBIResult DBIFN DbiSwitchToIndex (phCursor, pszIndexName, pszTagName,
  iIndexId, bCurrRec);
```

Delphi syntax

```
function DbiSwitchToIndex (var hCursor: hDBICur; pszIndexName: PChar;
  pszTagName: PChar; iIndexId: Word; bCurrRec: Bool): DBIResult stdcall;
```

Description

DbiSwitchToIndex changes the active index order of the given cursor.

Parameters

phCursor Type: phDBICur (Input/Output)
On input, *phCursor* specifies the original cursor handle; on output, pointer to the new cursor handle.

pszIndexName Type: pCHAR (Input)
Pointer to the name of the index or pseudo-index. The *pszIndexName* string is limited to 127 bytes in length.

pszTagName Type: pCHAR (Input)
Pointer to the tag name string. Used for dBASE and FoxPro tables only.

iIndexId Type: UINT16 (Input)
Specifies the index ID.

bCurrRec Type: BOOL (Input)
If TRUE, positions the new cursor on the current record of the original cursor.

Usage

This function allows the user to change the index order of a cursor without closing the cursor and opening another cursor. The original cursor is passed into the function, and a new cursor handle is returned with the new ordering. The original cursor handle becomes invalid and cannot be used.

Setting *pszIndexName*, *pszTagName*, and *iIndexId* to NULL is equivalent to changing the order to the default order. As a result, the cursor is set to one of the following orders:

- Relational order for dBASE, FoxPro and SQL tables.
- Natural order for Access tables.
- Primary index order for a keyed Paradox table or physical order for a Paradox heap table.

If *bCurrRec* is set to TRUE, the new cursor is positioned on the same record as the original cursor. If *bCurrRec* is set to FALSE, the new cursor is positioned at BOF. If the original cursor is not positioned on a valid record (for example, the current record has been deleted and the cursor has not been advanced), this function with *bCurrRec* set to TRUE fails. If this function is used to switch to the same index, then no action is taken.

Note: The size of a bookmark buffer may change after a call to DbiSwitchToIndex.

Pseudo-indexes: To describe a pseudo-index rather than an existing physical index, replace the *pszIndexName* parameter with a string composed of field names. The marker character @ denotes the use of a pseudo-index. For example, "@Customer Number@Order Number" describes a pseudo-index on a key formed by concatenating the Customer Number field with the Order Number field.

Each field identifier in the pseudo-index name must be preceded by the @ character. This

character is illegal in "true" index names. No new index is generated at the server; the behavior of the pseudo-index is simulated entirely by use of the proper ORDER BY clauses on the query populating the local BDE record cache.

Fields can be identified by field numbers as well as by field names. For example, the string "@2@3@11" describes a pseudo-index consisting of the second, third, and eleventh field of the table, concatenated to make up a single key.

Each of the component fields within a *pszIndexName* is assumed to be in ASCENDING order. Ordering is case-sensitive (unless case-sensitivity is not supported on the specific server). If the fields in the *pszIndexName* represent a real unique index on the server, the pseudo-index becomes unique; otherwise, it is non-unique.

Prerequisites

A valid cursor handle must be obtained on a table; not on a query or an in-memory table. If the given index is not open, it is automatically opened by this function before switching to that index order. (Therefore, all error return codes for DbiOpenIndex apply.)

Completion state

Switching the index may change some properties of the cursor, such as bookmark size and the key buffer size. Existing bookmarks on the original cursor cannot be used in the new cursor, so any saved positions will no longer be applicable to the new cursor.

DbiResult return values

DBIERR_NONE The index was successfully changed.

DBIERR_NOCURRREC	Cannot position to the current record because the original cursor is not positioned on a valid record. (Applicable only if bCurrRec is set to TRUE.)
DBIERR_NOSUCHINDEX	No such index exists for the table.
DBIERR_INVALIDHNDL	The specified handle was invalid or NULL.
DBIERR_INDEXOUTOFDATE	An attempt was made to switch to a non-maintained index that is out of date.

See also

DbiAddIndex, DbiOpenIndex, DbiRegenIndex, DbiRegenIndexes, DbiOpenTable

C Examples: DbiSwitchToIndex

Set cursor to the specified index name:

This examples uses the following input:
fDbiSwitchToIndex(&hPXCur, "Vendor No", FALSE);

```
DBIResult fDbiSwitchToIndex(phDBICur phTmpCur, pCHAR IdxName, BOOL SavePos)
{
   DBIResult   rslt;
   rslt = Chk(DbiSwitchToIndex(phTmpCur, IdxName, NULL, NULL, SavePos));
   return rslt;
}
```

Delphi Examples: DbiSwitchToIndex

Set cursor to the specified index name:

Users of TTable objects should use the IndexName property to change indexes. Set cursor to the specified index name and keep the cursor on the same record.

This example uses the following input:

```
  fDbiSwitchToIndex(Table1.Handle, 'VendorNo');
```

The procedure is:

```
procedure fDbiSwitchToIndex(hTmpCur: hDbiCur; IdxName: string);
begin
  Check(DbiSwitchToIndex(hTmpCur, PChar(IdxName), nil, 0, True));
end;
```

DbiTimeDecode {button C Examples,JI(`>example',`exdbitimedecode')} {button Delphi Examples,JI(`>example',`dexdbitimedecode')}

C syntax

```
DBIResult DBIFN DbiTimeDecode (timeT, piHour, piMin, piMilSec);
```

Delphi syntax

```
function DbiTimeDecode (timeT: Time; var iHour: Word; var iMin: Word; var
  iMilSec: Word): DBIResult stdcall;
```

Description

DbiTimeDecode decodes TIME into separate components (hours, minutes, milliseconds).

Parameters

timeT Type: TIME (Input)
Specifies the encoded time.

piHour Type: pUINT16 (Output)
Pointer to the client variable that receives the decoded hours. Valid values range from 0 through 23.

piMin Type: pUINT16 (Output)
Pointer to the client variable that receives the decoded minutes. Valid values range from 0 through 59.

piMilSec Type: pUINT16 (Output)
Pointer to the client variable that receives the decoded milliseconds. Valid values range from 0 through 59999.

Usage

This function enables the client application to interpret time values obtained from DbiGetField. This function is a non-driver related service function; it works for all drivers.

DbiResult return values

DBIERR_NONE	The time was decoded successfully.
DBIERR_INVALIDHNDL	The pointer to the decoded hours, minutes, or milliseconds is NULL.
DBIERR_INVALIDTIME	The specified encoded time is invalid.

See also

DbiTimeEncode, DbiDateDecode, DbiDateEncode, DbiTimeStampDecode, DbiTimeStampEncode

C Examples: DbiTimeDecode

Decode a TIME variable into hour, minutes, and seconds.

This example uses the following input:

fDbiTimeDecode(MyTime, &Hour, &Minute, &Sec);

```
DBIResult fDbiTimeDecode(TIME Time, pUINT16 Hour, pUINT16 Minute, pUINT16
  Seconds)
{
   DBIResult   rslt;
   UINT16      MSeconds;
   rslt = Chk(DbiTimeDecode(Time, Hour, Minute, &MSeconds));
   if (rslt == DBIERR_NONE)
      *Seconds = (UINT16)(MSeconds /  1000);
   return rslt;
}
```

Delphi Examples: DbiTimeDecode

Decode a TIME variable into hour, minutes, and seconds.

This example decodes Hour, Minute, and Seconds fields from a the TIME value specified in the TimeT parameter and returns the time value as a string.

This example uses the following input:

```
  TimeStr := fDbiTimeDecode(MyTime, MyHour, MyMin, MyMilSec);
```

The function is:

```
function fDbiTimeDecode(TimeT: Time; var iHour, iMin, iSec: Word): string;
begin
  Check(DbiTimeDecode(TimeT, iHour, iMin, iSec));
  iSec := iSec div 1000;
  SetLength(Result, 12);
  if (iHour < 12) then begin
    if (iHour = 0) then
      iHour := 12;
    Result := Format('%d:%d:%d AM', [iHour, iMin, iSec]);
  end
  else begin
    if (iHour > 12) then
      dec(iHour, 12);
    Result := Format('%d:%d:%d PM', [iHour, iMin, iSec]);
  end;
  SetLength(Result, StrLen(PChar(Result)));
end;
```

DbiTimeEncode{button C Examples,JI(`>example',`exdbitimeencode')} {button Delphi Examples,JI(`>example',`dexdbitimeencode')}

C syntax

```
DBIResult DBIFN DbiTimeEncode (iHour, iMin, iMilSec, ptimeT);
```

Delphi syntax

```
function DbiTimeEncode (iHour: Word; iMin: Word; iMilSec: Word; var timeT:
  Time): DBIResult stdcall;
```

Description

DbiTimeEncode encodes separate time components into TIME for use by DbiPutField and other functions.

Parameters

iHour Type: UINT16 (Input)
Specifies hours. Valid values range from 0 through 23.

iMin Type: UINT16 (Input)
Specifies minutes. Valid values range from 0 through 59.

iMilSec Type: UINT16 (Input)
Specifies milliseconds. Valid values range from 0 through 59999.

ptimeT Type: pTIME (Output)
Pointer to the client variable that receives the encoded time.

Usage

This function enables the client application to construct a time value for use by DbiPutField. This function is a non-driver related service function; it works for all drivers.

DbiResult return values

DBIERR_NONE	The time was successfully encoded.
DBIERR_INVALIDHNDL	*ptimeT* is NULL.
DBIERR_INVALIDTIME	Ranges of hour, minute, and millisecond parameters are invalid.

See also

DbiDateEncode, DbiDateDecode, DbiTimeStampDecode, DbiTimeStampEncode, DbiPutField

C Examples: DbiTimeEncode

Encode Hour, Minute, and Seconds into a TIME variable.

This example uses the following input:

fDbiTimeEncode(10, 50, 15, &MyTime);

```
DBIResult fDbiTimeEncode(UINT16 Hour, UINT16 Minute, UINT16 Seconds, pTIME
  Time)
{
   DBIResult   rslt;
   UINT16      MSeconds;
   MSeconds = (UINT16)(Seconds * 1000);
   if (MSeconds > 59999)
     return DBIERR_INVALIDTIME;
   else
     rslt = Chk(DbiTimeEncode(Hour, Minute, MSeconds, Time));
   return rslt;
}
```

Delphi Examples: DbiTimeEncode

Encode Hour, Minute, and Seconds into a TIME variable.

This example uses the following input:

```
fDbiTimeEncode(4,20,42, MyTime);
```

The procedure is:

```
procedure fDbiTimeEncode(iHour: Word; iMin: Word; iSec: Word; var TimeT:
  Time);
begin
  if (iSec > 59) then
    Check(dbiErr_InvalidTime);
  iSec := iSec * 1000;
  Check(DbiTimeEncode(iHour, iMin, iSec, TimeT));
end;
```

DbiTimeStampDecode {button C Examples,JI(`>example',`exdbitimestampdecode')} {button Delphi Examples,JI(`>example',`dexdbitimestampdecode')}

C syntax

```
DBIResult DBIFN DbiTimeStampDecode (tsTS, pdateD, ptimeT);
```

Delphi syntax

```
function DbiTimeStampDecode (tsTS: TIMESTAMP; var dateD: DbiDate; var timeT:
  Time): DBIResult stdcall;
```

Description

DbiTimeStampDecode extracts separate encoded DBIDATE and TIME components from the TIMESTAMP.

Parameters

tsTS Type: TIMESTAMP (Input)
Specifies the encoded DATETIME timestamp.

pdateD Type: pDBIDATE (Output)
Pointer to the client variable that receives the encoded DBIDATE component.

ptimeT Type: pTIME (Output)
Pointer to the client variable that receives the encoded TIME component.

Usage

This function enables the client to interpret TIMESTAMP values obtained from DbiGetField.This function is a non-driver related service function; it works for all drivers.

Completion state

DateDecode and *TimeDecode* must be called in order to further decode the date and time elements into their individual components (for example, month, day, year/hours, minutes, milliseconds).

DbiResult return values

DBIERR_OK The timestamp was successfully decoded.

DBIERR_INVALIDHNDL *pdateD* or *ptimeT* is NULL.

See also

DbiTimeStampEncode, DbiGetField

C Examples: DbiTimeStampDecode

Decode a TimeStamp variable into a string including all information.

This example uses the following input:

fDbiTimeStampDecode(TS, Buffer);

```
DBIResult fDbiTimeStampDecode(TIMESTAMP TS, pCHAR TSStr)
{
   DBIResult   rslt;
   DBIDATE     Date;
   TIME        Time;
   UINT16      h, m, ms, M, D;
   INT16       Y;
   CHAR        AMPM[3] = "AM";

   rslt = Chk(DbiTimeStampDecode(TS, &Date, &Time));
   if (rslt != DBIERR_NONE)
      return rslt;

   rslt = Chk(DbiTimeDecode(Time, &h, &m, &ms));
   if (rslt != DBIERR_NONE)
      return rslt;

   rslt = Chk(DbiDateDecode(Date, &M, &D, &Y));
   if (rslt != DBIERR_NONE)
      return rslt;

   if (h > 12)
   {
      strcpy(AMPM, "PM");
      h -= (UINT16)12;
   }

   wsprintf(TSStr, "%d:%d:%d %s, %d/%d/%d", h, m, (ms / 1000), AMPM, M, D,
  Y);
   return rslt;
}
```

Delphi Examples: DbiTimeStampDecode

Decode a TimeStamp variable into a string including all information

This example uses the following input:

```
  fDbiTimeStampDecode(TS, Buffer);
```

The function is:

```
function fDbiTimeStampDecode(timestampTS: TimeStamp): string;
var
   DateVar: dbiDATE;
   TimeVar: TIME;
   hour, min, millsec, Month, Day: Word;
   Year: SmallInt;
begin
  SetLength(Result, 100);
  Check(DbiTimeStampDecode(timestampTS, DateVar, TimeVar));
  Check(DbiTimeDecode(TimeVar, hour, min, millsec));
  Check(DbiDateDecode(DateVar, Month, Day, Year));
  if (hour > 12) then
    Result := Format('Time: %d:%d:%d PM, Date: %d/%d/%d',
      [hour - 12, min, millsec div 1000, Month, Day, Year])
  else
    Result := Format('Time: %d:%d:%d AM, Date: %d/%d/%d',
      [hour, min, millsec div 1000, Month, Day, Year]);
  SetLength(Result, StrLen(PChar(Result)));
end;
```

DbiTimeStampEncode {button C Examples,JI(`>example',`exdbitimestampencode')} {button Delphi Examples,JI(`>example',`dexdbitimestampencode')}

C syntax

```
DBIResult DBIFN DbiTimeStampEncode (dateD, timeT, ptsTS);
```

Delphi syntax

```
function DbiTimeStampEncode (dateD: DbiDate; timeT: Time; var tsTS:
  TimeStamp): DBIResult stdcal;
```

Description

DbiTimeStampEncode encodes the encoded DBIDATE and encoded TIME into a TIMESTAMP.

Parameters

dateD Type: DBIDATE (Input)
Specifies the encoded date.

timeT Type: TIME (Input)
Specifies the encoded time.

ptsTS Type: pTIMESTAMP (Output)
Pointer to the client variable that receives the encoded timestamp.

Usage

This function enables the client application to construct a TIMESTAMP value for use in DbiPutField. This function is a non-driver related service function; it works for all drivers.

DbiResult return values

DBIERR_NONE	The timestamp was successfully encoded.
DBIERR_INVALIDHNDL	*ptsTS* is NULL.
DBIERR_INVALIDTIMESTAMP	The range of date and time parameters is invalid.

See also

DbiTimeStampDecode, DbiPutField

C Examples: DbiTimeStampEncode

Encode a TimeStamp variable from a DBIDATE and TIME variable.

This example uses the following input:

fDbiTimeStampEncode(MyDate, MyTime, &TS);

```
DBIResult fDbiTimeStampEncode(DBIDATE Date, TIME Time, pTIMESTAMP TS)
{
   DBIResult   rslt;
   rslt = Chk(DbiTimeStampEncode(Date, Time, TS));
   return rslt;
}
```

Delphi Examples: DbiTimeStampEncode

Encode a TimeStamp variable from a DBIDATE and TIME variable.
This example uses the following input:

```
  fDbiTimeStampEncode(MyDate, MyTime, TS);
```

The procedure is:

```
procedure fDbiTimeStampEncode(ADate: dbiDate; timeT: TIME; var timestampTS:
  TimeStamp);
begin
  Check(DbiTimeStampEncode(ADate, timeT, timestampTS));
end;
```

DbiTranslateField {button C Examples,JI(`>example',`exdbitranslatefield')} {button Delphi Examples,JI(`>example',`dexdbitranslatefield')}

C syntax

```
DBIResult DBIFN DbiTranslateField (hXlt, pSrc, pDest);
```

Delphi syntax

```
function DbiTranslateField (hXlt: hDBIXlt; pSrc: Pointer; pDest: Pointer):
  DBIResult stdcall;
```

Description

DbiTranslateField translates a logical or physical field value to any compatible logical or physical field value.

Parameters

hXlt Type: hDBIXlt (Input)
Specifies the translate handle.

pSrc Type: pBYTE (Input)
Pointer to the source field.

pDest Type: pBYTE (Output)
Pointer to the destination field.

Usage

This function reads the source field and places the data in the destination field after converting the data to the type of the destination field.

SQL: This function can be used only on fields that are contained with a valid SQL record buffer. You must build the translation object by using a BDE-supplied field descriptor because each field descriptor contains an offset to a NULL indicator and each field translation must read or write this NULL indicator. The offset from the field buffer to the NULL indicator is stored when the translation object is built.

DbiResult return values

DBIERR_NONE	The field was translated successfully.
DBIERR_FIELDISBLANK	The source field is blank.

See also

DbiOpenFieldXlt, DbiCloseFieldXlt

C Examples: DbiTranslateField

Translate a field from IDAPI Logical format to its physical format equivalent or vice versa.

```
DBIResult fDbiTranslateField(hDBICur hCur, hDBIXlt hXlt, pBYTE pTransField)
{
   DBIResult      rslt;
   pBYTE          pFieldBuf;
   pBYTE          pRecBuf;
   CURProps       CurProps;
   Chk(DbiGetCursorProps(hCur, &CurProps));
   pRecBuf = (pBYTE)malloc(CurProps.iRecBufSize);
   pFieldBuf = (pBYTE)malloc(1024);
   pTransField = (pBYTE)malloc(1024);
   Chk(DbiSetToBegin(hCur));
   Chk(DbiGetNextRecord(hCur, dbiNOLOCK, pRecBuf, NULL));
   Chk(DbiGetField(hCur, 1, pRecBuf, pFieldBuf, NULL));
   rslt = Chk(DbiTranslateField(hXlt, pFieldBuf, pTransField));
   return rslt;
}
```

Delphi Examples: DbiTranslateField

An example for this function is under development and will be provided in an upcoming Help release.

DbiTranslateRecordStructure {button C Examples,JI(`>example',`exdbitranslaterecordstructure')} {button Delphi Examples,JI(`>example',`dexdbitranslaterecordstructure')}

C syntax

```
DBIResult DBIFN DbiTranslateRecordStructure (pszSrcDriverType, iFlds,
  pfldsSrc, pszDstDriverType, pszLangDriver, pfldsDst, bCreatable);
```

Delphi syntax

```
function DbiTranslateRecordStructure (pszSrcDriverType: PChar; iFlds: Word;
  pfldsSrc: pFLDDesc; pszDstDriverType: PChar; pszLangDriver: PChar;
  pfldsDst: pFLDDesc; bCreateable: Bool): DBIResult stdcall;
```

Description

DbiTranslateRecordStructure translates the source driver's physical or logical fields to equivalent physical or logical fields of the destination driver.

Parameters

pszSrcDriverType Type: pCHAR (Input)
Pointer to the source driver type. If NULL, it is assumed that the source fields are logical with a NULL driver type.

iFlds Type: UINT16 (Input)
Specifies the number of fields.

pfldsSrc Type: pFLDDesc (Input)
Pointer to an array of the logical or physical types of the source fields.

pszDstDriverType Type: pCHAR (Input)
Pointer to the destination driver type. If NULL, it is assumed that the destination fields are logical with a NULL driver type.

pszLangDriver Type: pCHAR (Input)
Pointer to the destination driver's language driver name. This language driver is used to validate the destination field names after the translation.

pfldsDst Type: pFLDDesc (Output)
Pointer to an array of the destination fields.

bCreatable Type: BOOL (Input)
If True, map to creatable fields only

Usage

This function takes the logical or physical fields of the source driver and attempts to map them to equivalent logical or physical fields of the destination driver. If an exact match is not found, the function attempts to map to the closest possible logical or physical fields of the destination driver. If a close match is not found, this returns the error DBIERR_NOTSUPPORTED.

DbiResult return values

DBIERR_NONE The translation was successfully completed.

DBIERR_NOTSUPPORTED Returned if source fields cannot be translated into equivalent destination fields.

C Examples: DbiTranslateRecordStructure

Create a new table of the specified type by borrowing a field structure from another table.

The new table is created in the same directory or server as the source table. Return the cursor to the newly created table. This example uses the following input:

fDbiTranslateRecordStructure(hIBCur, "NEWCUST", "INTRBASE", &hTmpCur);

```
DBIResult fDbiTranslateRecordStructure(hDBICur hSrcCur, pCHAR NewTblName,
                     pCHAR DrvType, phDBICur phDstCur)
{
   DBIResult   rslt;
   pFLDDesc    SrcFldDesc, DestFldDesc;
   CURProps    CurProps;
   CRTblDesc   TblDesc;
   hDBIDb      hTmpDb;

   rslt = Chk(DbiGetCursorProps(hSrcCur, &CurProps));
   if (rslt != DBIERR_NONE)
      return rslt;
   SrcFldDesc = (pFLDDesc)malloc(CurProps.iFields * sizeof(FLDDesc));
   DestFldDesc = (pFLDDesc)malloc(CurProps.iFields * sizeof(FLDDesc));
   rslt = Chk(DbiGetFieldDescs(hSrcCur, SrcFldDesc));
   if (rslt != DBIERR_NONE)
      return rslt;
   rslt = Chk(DbiTranslateRecordStructure(NULL, CurProps.iFields,
                    SrcFldDesc, DrvType, NULL, DestFldDesc, FALSE));
   if (rslt != DBIERR_NONE)
   {
        free(SrcFldDesc); free(DestFldDesc);
        return rslt;
   }
   memset((void *) &TblDesc , 0, sizeof(CRTblDesc));
   strcpy(TblDesc.szTblName, NewTblName);
   strcpy(TblDesc.szTblType, DrvType);
   TblDesc.iFldCount = CurProps.iFields;
   TblDesc.pfldDesc = DestFldDesc;
   rslt = Chk(DbiGetObjFromObj(hSrcCur, objDATABASE, &hTmpDb));
   if (rslt != DBIERR_NONE)
   {
        free(SrcFldDesc); free(DestFldDesc);
        return rslt;
   }
   rslt = Chk(DbiCreateTable(hTmpDb, TRUE, &TblDesc));
   if (rslt != DBIERR_NONE)
   {
        free(SrcFldDesc); free(DestFldDesc);
        return rslt;
   }
   rslt = Chk(DbiOpenTable(hTmpDb, NewTblName, DrvType, NULL, NULL, 0,
  dbiREADWRITE,
                   dbiOPENSHARED, xltFIELD, FALSE, NULL, phDstCur));
   free(SrcFldDesc); free(DestFldDesc);
   return rslt;
}
```

Delphi Examples: DbiTranslateRecordStructure

Creates an empty version of SrcTbl to DestTbl. This will convert from any source type to any destination type--Paradox to InterBase and so on. The Table does not have any indexes.

This example uses the following input:

```
  fDbiTranslateRecordStructure(AnimalTbl, NewTbl, AnimalTbl.DBHandle);
```

The procedure is:

```
procedure fDbiTranslateRecordStructure(SrcTbl, DestTbl: TTable; DestDB:
  hDBIDb);
var
  pSrcFlds, pDestFlds: pFLDDesc;
  TblDesc: CRTblDesc;
  DBType: string;
  W: Word;
begin
  pSrcFlds := AllocMem(SrcTbl.FieldCount * sizeof(FLDDesc));
  pDestFlds := AllocMem(SrcTbl.FieldCount * sizeof(FLDDesc));
  try
    SetLength(DBType, DBIMAXNAMELEN);
    // Get the destination database type
    Check(DbiGetProp(hDBIObj(DestDb), dbDATABASETYPE,
      PChar(DBType), DBIMAXNAMELEN, W));
    SetLength(DBType, StrLen(PChar(DBType)));
    if (DBType = 'STANDARD') then begin
      if (UpperCase(ExtractFileExt(DestTbl.TableName)) = '.DB') then
        DBType := szParadox
      else if (UpperCase(ExtractFileExt(DestTbl.TableName)) = '.DBF') then
        DBType := szDbase
      else if (UpperCase(ExtractFileExt(DestTbl.TableName)) = '.') then
        DBType := szParadox
      else
        raise EDBEngineError.Create(DBIERR_UNKNOWNDRIVER);
    end;
    // Get the source field information
    Check(DbiGetFieldDescs(SrcTbl.Handle, pSrcFlds));
    // Translate the source fields into the destination fields
    Check(DbiTranslateRecordStructure(nil, SrcTbl.FieldCount, pSrcFlds,
      PChar(DBType), nil, pDestFlds, False));
    FillChar(TblDesc, sizeof(TblDesc), #0);
    StrPCopy(TblDesc.szTblName, DestTbl.TableName);
    StrPCopy(TblDesc.szTblType, DBType);
    TblDesc.iFldCount := SrcTbl.FieldCount;
    TblDesc.pFldDesc := pDestFlds;
    // Create the destination table
    Check(DbiCreateTable(DestDB, True, TblDesc));
  finally
    FreeMem(pSrcFlds, SrcTbl.FieldCount * sizeof(FLDDesc));
    FreeMem(pDestFlds, SrcTbl.FieldCount * sizeof(FLDDesc));
  end;
end;
```

DbiTruncateBlob {button C Examples,JI(`>example',`exdbitruncateblob')} {button Delphi Examples,JI(`>example',`dexdbitruncateblob')}

C syntax
```
DBIResult DBIFN DbiTruncateBlob (hCursor, pRecBuf, iField, iLen);
```

Delphi syntax
```
function DbiTruncateBlob (hCursor: hDBICur; pRecBuf: Pointer; iField: Word;
  iLen: Longint): DBIResult stdcall;
```

Description
DbiTruncateBlob is used to shorten the size of the contents of a BLOB field, or to delete the contents of a BLOB field from the record, by shortening it to zero.

Parameters
hCursor Type: hDBICur (Input)
Specifies the cursor handle.

pRecBuf Type: pBYTE (Input)
Pointer to the record buffer.

iField Type: UINT16 (Input)
Specifies the ordinal number of BLOB field within the record buffer.

iLen Type: UINT32 (Input)
Specifies the new shorter length of the BLOB. If zero is specified, the whole BLOB is truncated.

Usage
This is the only way to delete a BLOB without deleting the entire record.

Standard, Access: It is advisable to lock the record before opening the BLOB in read-write mode to ensure that another client application does not lock the record.

Prerequisites
The current record must contain a BLOB field. The BLOB field must be open in dbiREADWRITE mode by a call to DbiOpenBlob.

Completion state
After shortening the BLOB field, DbiModifyRecord must be called to post the altered record to the table.

DbiResult return values
DBIERR_NONE	The BLOB field was successfully truncated.
DBIERR_BLOBNOTOPENED	The specified BLOB field was not opened via a call to DbiOpenBlob.
DBIERR_INVALIDBLOBHANDLE	The BLOB handle supplied in the record buffer is invalid.
DBIERR_NOTABLOB	The specified field number does not correspond to a BLOB field.
DBIERR_INVALIDBLOBOFFSET	The specified *iOffSet* is greater than the length of the BLOB field.
DBIERR_READONLYFLD	The BLOB field was opened in dbiREADONLY mode and cannot be modified.

See also
DbiGetBlob, DbiOpenBlob, DbiPutBlob, DbiFreeBlob, DbiModifyRecord

C Examples: DbiTruncateBlob

Copy a table from the specified cursor and empty all the blob fields.

Packing the newly created table will free up space. This example uses the following input:
fDbiTruncateBlob(hPXBlobCur, "NEWBIO", &hTmpCur)

```
DBIResult fDbiTruncateBlob(hDBICur hTmpCur, pCHAR NewTblName, phDBICur
  phTmpCur)
{
   DBIResult    rslt;
   hDBIDb       hTmpDb;
   CURProps     CurProps;
   pFLDDesc     pFldDesc;
   UINT16       FldCount;
   pBYTE        pRecBuf;

   rslt = Chk(DbiGetObjFromObj(hTmpCur, objDATABASE, &hTmpDb));
   if (rslt != DBIERR_NONE)
      return rslt;
   rslt = Chk(DbiGetCursorProps(hTmpCur, &CurProps));
   if (rslt != DBIERR_NONE)
      return rslt;
   rslt = Chk(DbiCopyTable(hTmpDb, TRUE, CurProps.szName,
  CurProps.szTableType, NewTblName));
   if (rslt != DBIERR_NONE)
      return rslt;
   rslt = Chk(DbiOpenTable(hTmpDb, NewTblName, CurProps.szTableType, NULL,
  NULL, 0,
                  dbiREADWRITE, dbiOPENSHARED, xltFIELD, FALSE, NULL,
  phTmpCur));
   if (rslt != DBIERR_NONE)
      return rslt;
   pFldDesc = (pFLDDesc)malloc(CurProps.iFields * sizeof(FLDDesc));
   pRecBuf = (pBYTE)malloc(CurProps.iRecBufSize * sizeof(BYTE));
   rslt = Chk(DbiGetFieldDescs(*phTmpCur, pFldDesc));
   if (rslt != DBIERR_NONE)
   {
      free(pFldDesc); free(pRecBuf);
      return rslt;
   }
   while (DbiGetNextRecord(*phTmpCur, dbiWRITELOCK, pRecBuf, NULL) ==
  DBIERR_NONE)
   {
      for (FldCount = 0; FldCount < CurProps.iFields; FldCount++)
      {
         if (pFldDesc[FldCount].iFldType == fldBLOB)
         {
            rslt = Chk(DbiOpenBlob(*phTmpCur, pRecBuf,
  pFldDesc[FldCount].iFldNum,
                                   dbiREADWRITE));
            if (rslt != DBIERR_NONE)
            {
               free(pFldDesc); free(pRecBuf);
               return rslt;
            }
            rslt = Chk(DbiTruncateBlob(*phTmpCur, pRecBuf,
  pFldDesc[FldCount].iFldNum, 0));
```

```
            if (rslt != DBIERR_NONE)
            {
               free(pFldDesc); free(pRecBuf);
               return rslt;
            }
            rslt = Chk(DbiModifyRecord(*phTmpCur, pRecBuf, TRUE));
            if (rslt != DBIERR_NONE)
            {
               free(pFldDesc); free(pRecBuf);
               return rslt;
            }
            rslt = Chk(DbiFreeBlob(*phTmpCur, pRecBuf,
  pFldDesc[FldCount].iFldNum));
            if (rslt != DBIERR_NONE)
            {
               free(pFldDesc); free(pRecBuf);
               return rslt;
            }
         }
      }
   }
   rslt = Chk(DbiSetToBegin(*phTmpCur));
   free(pFldDesc); free(pRecBuf);
   return rslt;
}
```

Delphi Examples: DbiTruncateBlob

Truncate all BLOBs in the specified field to zero. If any error occurs while removing BLOB information, stop at that record.

This example uses the following input:

```
  fDbiTruncateBlob(BiotestTbl, BiotestTbl.FieldByName('Notes').Index);
```

The procedure is:

```
procedure fDbiTruncateBlob(BlobTbl: TTable; Index: Word);
var
  hCur: hDBICur;
  pRecBuf: pBYTE;
begin
  hCur := nil;
  // Make sure the field specified is a BLOb type
  if (BlobTbl.Fields[Index] is TblobField) then begin
    pRecBuf := AllocMem(BlobTbl.RecordSize);
    try
      // Clone a cursor to the table so data aware controls keep their place
      Check(DbiCloneCursor(BlobTbl.Handle, False, False, hCur));
      Check(DbiSetToBegin(hCur));
      // Iterate throuth the table removing BLOb information
      while (DbiGetNextRecord(hCur, dbiWRITELOCK, pRecBuf, nil) =
  DBIERR_NONE)
      do begin
        // BDE funcstions use a 1 for the first field vs. Delphi's 0;
        // add 1 to the index
        Check(DbiOpenBlob(hCur, pRecBuf, Index + 1, dbiREADWRITE));
        Check(DbiTruncateBlob(hCur, pRecBuf, Index + 1, 0));
        Check(DbiModifyRecord(hCur, pRecBuf, True));
        Check(DbiFreeBlob(hCur, pRecBuf, Index + 1));
      end;
    finally  // Close cloned cursor and free record buffer memory
      if (hCur <> nil) then
        Check(DbiCloseCursor(hCur));
      FreeMem(pRecBuf, BlobTbl.RecordSize);
    end;
  end
  else
    raise EDatabaseError.Create('Field: ' +
      BlobTbl.Fields[Index].FieldName + ', is not a blob type');
end;
```

DbiUndeleteRecord {button C Examples,JI(`>example',`exdbiundeleterecord')} {button Delphi Examples,JI(`>example',`dexdbiundeleterecord')}

C syntax
```
DBIResult DBIFN DbiUndeleteRecord (hCursor);
```

Delphi syntax
```
function DbiUndeleteRecord (hCursor: hDBICur): DBIResult stdcall;
```

Description
DbiUndeleteRecord undeletes a dBASE or FoxPro record that has been marked for deletion (a soft delete).

Parameters
hCursor Type: hDBICur (Input)
Specifies the dBASE or FoxPro cursor handle.

Usage
dBASE or FoxPro: This function is supported with dBASE or FoxPro tables only.

Paradox, Access: This function is not supported with Paradox and Access tables.

SQL: This function is not supported with SQL tables.

Prerequisites
The cursor must be positioned on a record. The cursor must have the property bDeletedOn set to TRUE.

Completion state
The current record is recalled if it was marked for deletion.

DbiResult return values

DBIERR_NONE	The dBASE or FoxPro record was successfully undeleted.
DBIERR_INVALIDHNDL	The specified cursor handle is invalid or NULL.
DBIERR_BOF	The cursor is positioned on the crack at the beginning of the file.
DBIERR_EOF	The cursor is positioned on the crack at the end of the file.
DBIERR_NA	The specified record was not deleted; cannot undelete the record.
DBIERR_TABLEREADONLY	The specified table is read-only; cannot undelete the record.
DBIERR_FILELOCKED	The table is locked by another user; cannot undelete the record.
DBIERR_NOTSUPPORTED	The function is supported only for dBASE or FoxPro tables.
DBIERR_NOCURRREC	The cursor is not positioned on a valid record.

See also
DbiDeleteRecord, DbiPackTable

C Examples: DbiUndeleteRecord

An example for this function is under development and will be provided in an upcoming Help release.

Delphi Examples: DbiUndeleteRecord

Undeletes a dBASE record if it is supported.

This example uses the following input:

```
  fDbiUndeleteRecord(AnimalTbl);
```

The procedure is:

```
procedure fDbiUndeleteRecord(dBASETbl: TTable);
var
  CProps: CurProps;
begin
  Check(DbiGetCursorProps(dBASETbl.Handle, CProps));
  // Raise an EDBEngineError exception if the table is not dBASE
  if (StrIComp(CProps.szTableType, szDBASE) <> 0) then
    raise EDBEngineError.Create(DBIERR_NOTSUPPORTED);
  // Raise an EDatabaseError exception if the cursor does not have soft
  deletes on
  if (CProps.bDeletedOn = False) then
    raise EDatabaseError.Create('Soft deletes is not on');
  Check(DbiUndeleteRecord(dBASETbl.Handle));
end;
```

DbiUnlinkDetail {button C Examples,JI(`>example',`exdbiunlinkdetail')} {button Delphi Examples,JI(`>example',`dexdbiendlinkmode')}

C syntax

```
DBIResult DBIFN DbiUnlinkDetail (hDetlCursor);
```

Delphi syntax

```
function DbiUnlinkDetail (hDetlCursor: hDBICur): DBIResult stdcall;
```

Description

DbiUnlinkDetail removes the link from a detail cursor and its master.

Parameters

hDetlCursor Type: hDBICur (Input)
Specifies the detail cursor handle.

Usage

Links should be removed before calling DbiEndLinkMode.

Prerequisites

A call to DbiLinkDetail or DbiLinkDetailToExp.

Completion state

The cursors are no longer related to each other, but remain in the linked cursor mode. The function unlinks *hDetlCursor* from its master table, leaving *hDetlCursor* as a linked cursor associated with no master cursor. Thus, the detail cursor is not constrained by its master.

DbiResult return values

DBIERR_NONE	The link between the detail and master cursors was removed successfully.
DBIERR_INVALIDHNDL	The specified cursor handle is invalid or NULL.

See also

DbiLinkDetail, DbiLinkDetailToExp, DbiBeginLinkMode, DbiEndLinkMode

C Examples: DbiUnlinkDetail

An example for this function is under development and will be provided in an upcoming Help release.

Delphi Examples: DbiUnlinkDetail

An example for this function is under development and will be provided in an upcoming Help release.

DbiUseIdleTime

C syntax

```
DBIResult DBIFN DbiUseIdleTime (VOID);
```

Delphi syntax

```
function DbiUseIdleTime: DBIResult stdcall;
```

Description

This function is no longer supported. Use DbiSaveChanges instead.

DbiValidateProp {button C Examples,JI(`>example',`exdbivalidateprop')} {button Delphi Examples,JI(`>example',`dexdbisetprop')}

C syntax

```
DBIResult DBIFN DbiValidateProp (hObj, iProp, bSetting );
```

Delphi syntax

```
function DbiValidateProp (hObj: hDBIObj; iProp: Longint; bSetting: Bool):
  DBIResult stdcall;
```

Description

DbiValidateProp validates a property for a specified object handle.

Parameters

hObj Type: hDBIObj (Input)
Specifies the object handle.

iProp Type: UINT32 (Input)
Specifies the property to validate.

bSetting Type: BOOL (Input)
Set to TRUE if DbiValidateProp is setting the property; to FALSE if DbiValidateProp is getting the property.

Usage

Use DbiValidateProp to determine whether a given property can be changed or retrieved from the supplied object handle. You can call DbiValidateProp before DbiSetProp and DbiGetProp to determine if a property is valid for a given object.

DbiResult return values

DBIERR_NONE	The data meets all the requirements for the specified property.
DBIERR_NOTSUPPORTED	The property is invalid for the given object.

See also

DbiSetProp, DbiGetProp

C Examples: DbiValidateProp

An example for this function is under development and will be provided in an upcoming Help release.

Delphi Examples: DbiValidateProp

An example for this function is under development and will be provided in an upcoming Help release.

DbiVerifyField {button C Examples,JI(`>example',`exdbiverifyfield')} {button Delphi Examples,JI(`>example',`dexdbiverifyfield')}

C syntax

```
DBIResult DBIFN DbiVerifyField (hCursor, iField, pSrc, [pbBlank]);
```

Delphi syntax

```
function DbiVerifyField (hCursor: hDBICur; iField: Word; pSrc: Pointer; var
  bBlank: Bool): DBIResult stdcall;
```

Description

DbiVerifyField verifies that the data specified in *pSrc* is a valid data type for the field specified by *iField*, and that all validity checks specified for the field are satisfied. It can also be used to check if a field is blank.

Parameters

hCursor Type: hDBICur (Input)
Specifies the cursor handle.

iField Type: UINT16 (Input)
Specifies the ordinal number of the field in the record.

pSrc Type: pBYTE (Input)
Pointer to the buffer containing the data to be verified. If NULL, the function verifies whether a blank value is allowed.

pbBlank Type: pBOOL (Output)
Pointer to the client variable that is set to TRUE if the field is blank; otherwise, it is set to FALSE.

Usage

If the translation mode of the cursor is xltFIELD, *pSrc* is assumed to contain field data in BDE logical format, otherwise it is considered to be the driver's physical format. The validity checking aspect of this function enables the client application to report errors without actually attempting to write the data. It can also be used to check if a field is blank. If *pSrc* is NULL, the function verifies whether or not a blank value is allowed.

DbiVerifyField is not supported with BLOB fields.

dBASE or FoxPro: For dBASE and FoxPro tables, this function can be used only to determine if a field is blank.

Paradox: For Paradox tables, this function evaluates field-level validity checks; it does not evaluate referential integrity constraints.

Completion state

If the field is blank, the variable pointed to by *pbBlank* is set to TRUE. If any field-level validity check has failed, an error message is returned, indicating which type of validity check the field has failed.

DbiResult return values

DBIERR_NONE	The data meets all the requirements for the specified field.
DBIERR_MINVALERR	The data is less than the required minimum value.
DBIERR_MAXVALERR	The data is greater than the required maximum value.
DBIERR_REQDERR	The field cannot be blank.
DBIERR_LOOKUPTABLEERR	The value cannot be located in the assigned lookup table.

See also
DbiOpenTable, DbiPutField, DbiInsertRecord, DbiModifyRecord, DbiAppendRecord

C Examples: DbiVerifyField

Verify that a given field is valid.

Information was placed into the field buffer by using DbiPutField or DbiGetField.

```
DBIResult fDbiVerifyField(hDBICur hCur, INT16 FldNum, pBYTE FldBuf)
{
   DBIResult      rslt;
   rslt = Chk(DbiVerifyField(hCur, FldNum, FldBuf, NULL));
   return rslt;
}
```

Delphi Examples: DbiVerifyField

Verifiy that the data specified is valid for the first field.
In this example, the field must be of type double. Blank is set to True if the field is blank.

This example uses the following input:

```
  fDbiVerifyField(Table1.Handle, Blank);
```

The function is:

```
function fDbiVerifyField(hTmpCur: hDBICur; var Blank: Boolean): DbiResult;
var
  Key: Double;
begin
  Key:= 20000.00;
  Result := DbiVerifyField(hTmpCur, 1, @key, Blank);
end;
```

DbiWriteBlock {button C Examples,JI(`>example',`exdbiwriteblock')} {button Delphi Examples,JI(`>example',`dexdbiwriteblock')}

C syntax
```
DBIResult DBIFN DbiWriteBlock (hCursor, piRecords, pBuf);
```

Delphi syntax
```
function DbiWriteBlock (hCursor: hDBICur; var iRecords: Longint; pBuf:
  Pointer): DBIResult stdcall;
```

Description
DbiWriteBlock writes a block of records to the table associated with *hCursor*.

Parameters
hCursor Type: hDBICur (Input)
Specifies the cursor handle to the table.

piRecords Type: pUINT32 (Input/Output)
On input, *piRecords* is a pointer to the number of records to write. On output, pointer to the client variable that receives the actual number of records written. The number actually written may be less than requested if an integrity violation or other error occurred.

pBuf Type: pBYTE (Input)
Pointer to the buffer containing the records to be written.

Usage
This function is similar to calling DbiAppendRecord for the specified number of *piRecords*. DbiWriteBlock can access data in blocks larger than 64Kb, depending on the size you allocate for the buffer.

Note: This function cannot be used if the records contain non-empty BLOBs.

Paradox: This function verifies any referential integrity requirements or validity checks that may be in place. If either fails, the write operation is canceled.

Completion state
The cursor is positioned at the last record that was inserted.

DbiResult return values

DBIERR_NONE	The block of records contained in *pBuf* has been successfully written to the table specified by *hCursor*.
DBIERR_INVALIDHNDL	The specified cursor handle is invalid or NULL, or *piRecords* is NULL, or *pBuf* is NULL.
DBIERR_TABLEREADONLY	The table is opened read-only; cannot write to it.
DBIERR_NOTSUFFTABLERIGHTS	Insufficient table rights to insert a record. (Paradox only.)
DBIERR_NODISKSPACE	Insertion failed due to insufficient disk space.

See also
DbiReadBlock, DbiAppendRecord, DbiInsertRecord

C Examples: DbiWriteBlock

An example for this function is under development and will be provided in an upcoming Help release.

Delphi Examples: DbiWriteBlock

Add multiple records to a table

This example assumes that the Customer TTable object is the Customer.DB table.

It uses the following input:

```
fDbiWriteBlock(Table1; NumRecs);
```

The procedure is:

```
procedure fDbiWriteBlock(Customer: TTable; var RecordsToInsert: Longint);
var
  pRecordsBuf, pTmpBuf: pBYTE;
  Rec: Longint;
  CustNo: Double;
begin
  Randomize;
  GetMem(pRecordsBuf, Customer.RecordSize * RecordsToInsert);
  pTmpBuf := pRecordsBuf;
  try
    for Rec := 1 to RecordsToInsert do begin
      CustNo := Random(1000000);
      // Iterate through the entire record buffer filling each
      // individual record with information
      with Customer do begin
        Check(DbiInitRecord(Handle, pTmpBuf));
        Check(DbiPutField(Handle, FieldByName('CustNo').Index + 1, pTmpBuf,
          pBYTE(@CustNo)));
        Check(DbiPutField(Handle, FieldByName('Company').Index + 1, pTmpBuf,
          PChar('INPRISE Corporation')));
        Inc(pTmpBuf, RecordSize);
      end;
    end;
    Check(DbiWriteBLock(Customer.Handle, RecordsToInsert, pRecordsBuf));
  finally
    FreeMem(pRecordsBuf, Customer.RecordSize * RecordsToInsert);
  end;
end;
```

Data structures

This topic is an overview of various data structures used by BDE or its drivers, and miscellaneous data structures defined in the BDE header file IDAPI.H.

For topics on other data structures, click here:

{button ,AL(`types')} Other data structure topics

Major data structures used in BDE are listed in this table:

Structure	Description
BATTblDesc	Batch table definition
CANExpr	Expression tree descriptor
CANHdr	Header for all filter node classes
CBPROGRESSDesc	Progress callback
CBRESTcbDesc	Restructure callback
CFGDesc	Configuration descriptor
CLIENTInfo	Describes a client/application
CRTblDesc	Defines the general attributes of a table
CURProps	Describes the most commonly used cursor properties
DBDesc	Database descriptor
DBIEnumFld	Defines an enumerated field
DBIEnv	Defines the BDE environment
DBIErrInfo	Provides error information
DBIFUNCArgDesc	Returns arguments for a remote data source function, including field type and sub type.
DBIFUNCDesc	Describes a remote data source function, including name, overloads, and flags.
DBIQryProgress	Describes the status of a query
DRVType	Describes the driver and its capabilities
FILEDesc	File descriptor
FILTERInfo	Provides filter information
FLDDesc	Field descriptor
FLDType	Describes a field type
FMLDesc	Describes family of files in language driver descriptor
FMTBcd	Provides binary coded decimal format
FMTDate	Provides date format
FMTNumber	Provides number format
FMTTime	Provides time format
IDXDesc	Index descriptor
IDXType	Describes an index type
LDDesc	Describes a language driver
LOCKDesc	Lock descriptor
RECProps	Describes the record properties

RINTDesc	Provides referential integrity options
SECDesc	Describes each security descriptor
SESInfo	Provides session information
SPDesc	Describes a stored procedure
SPParamDesc	Describes the parameters to a stored procedure
SYSConfig	Provides basic system configuration information
SYSInfo	Provides BDE system status
SYSVersion	Provides BDE system version information
TBLBaseDesc	Provides basic information about a table
TBLExtDesc	Provides additional information about a table
TBLFullDesc	Provides a complete description of the table
TBLType	Describes a table's capabilities
USERDesc	Describes a user
VCHKDesc	Provides information about validity checking constraints
XInfo	Transaction descriptor

BATTblDesc (batch table definition)

The BATTblDesc structure defines a batch table, using the following fields:

Field	Type	Description
hDb	hDBIDb	Specifies the database handle.
szTblName	DBIPATH	Specifies the table name.
szTblType	DBINAME	Specifies the driver type; optional.
szUserName	DBINAME	Not currently used.
szPassword	DBINAME	Not currently used.

CANHdr (filter descriptor)

The CANHdr structure is the header for all filter node classes. It contains the following fields:

nodeClass **Type: NODEClass**
The following node classes are valid:

Node Class	Description
nodeUNARY	Node is a unary operator.
nodeBINARY	Node is a binary operator.
nodeCOMPARE	Node is a compare operator.
nodeFIELD	Node is a field.
nodeCONST	Node is a constant.
nodeTUPLE	Node is a record. Not currently used.
nodeCONTINUE	Node is a continue node.

CANExpr (expression tree descriptor)

For details on usage of this descriptor, see Using an expression tree.

Nodes and literals are in this structure:

Type	Name	Description
UINT16	*iVer*	Version tag of expression
UINT16	*iTotalSize*	Size of this structure
UINT16	*iNodes*	Number of nodes
UINT16	*iNodeStart*	Starting offset of nodes
UINT16	*iLiteralStart*	Starting offset of literals

canOP Type: CANOp

The following operators are valid:

Relational operators

canNOTDEFINED	Make this the first one
canISBLANK	Unary; is operand blank
canNOTBLANK	Unary; is operand not blank
canEQ	Binary; equal
canNE	Binary; not equal
canGT	Binary; greater than
canLT	Binary; less than
canGE	Binary; greater or equal
canLE	Binary; less or equal

Logical operators

canNOT	Unary; NOT
canAND	Binary; AND
canOR	Binary; OR

Operators identifying leaf operands

canTUPLE	Unary; entire record is operand
canFIELD	Unary; operand is field
canCONST	Unary; operand is constant

Miscellaneous operators

canCONTINUE	Unary; Stops evaluating records when operand evaluates to false. This is provided as a stop at high range filter value

CBPROGRESSDesc (progress callback)

The progress callback enables the client to be kept up to date as to the progress of a potentially long-running operation (such as DbiBatchMove or DbiQExec). When the client registers the callback, a callback buffer must be supplied. The buffer must be at least as large as sizeof(CBPROGRESSDesc). During query execution, the supplied callback function is called after certain milestones have been reached, giving the client an update on how execution is progressing. The CBPROGRESSDesc structure is stored in the client's call back buffer.

The CBPROGRESSDesc structure contains the following fields:

Field	Type	Description
iPercentDone	UINT16	Any number from -1 to 100 is valid. A value between 1 and 100 specifies the percentage done; for example, the value 50 indicates that the execution is half complete. If the value is -1, the progress of execution is indicated via the string *szMsg*, rather than with a percentage.
szMsg	DBIMSG	Specifies a string containing a message. This message serves as a progress report; for example, "Steps completed: 5." The message is displayed when iPercentDone is -1.

CBRESTcbDesc (restructure callback)

The CBRESTcbDesc structure contains the following fields:

Field	Type	Description
iErrCode	DBIResult	Specifies the error code number.
iTblNum	UINT16	Specifies the table number.
iObjNum	UINT16	For old objects *iObjNum* is the sequence or field number; for new objects *iObjNum* is the order in CRTblDesc.
eRestrObjType	RESTErrObjType	Specifies the object type.

eRestrObjType
Object type is a union of the following structures:

Structure	Type	Description
fldDesc	FLDDesc	Field descriptor
idxDesc	IDXDesc	Index descriptor
vchkDesc	VCHKDesc	Validity check descriptor
rintDesc	RINTDesc	Referential integrity descriptor
secDesc	SECDesc	Security descriptor

CFGDesc (configuration descriptor)

The CFGDesc structure describes the BDE configuration. It contains the following fields:

Field	Type	Description
szNodeName	DBINAME	Specifies the name of the leaf node.
szDescription	DBINAME	Specifies detailed information about the configuration leaf node.
iDataType	UINT16	Specifies the data type, which is always a string.
szValue	CHAR	Specifies a value large enough to hold any value [DBIMAXSCFLDLEN].
bHasSubnodes	BOOL	TRUE, if not a leaf node.

CLIENTInfo (client information)

The CLIENTInfo structure describes a client/application. It contains the following fields:

Field	Type	Description
szName	DBINAME	Specifies the documentary name.
iSessions	UINT16	Specifies the number of sessions.
szWorkDir	DBIPATH	Specifies the working directory.
szLang	DBINAME	Specifies the language of the client (for messages). See szLang

CRTblDesc (table descriptor)

DbiDoRestructure and DbiCreateTable both use the CRTblDesc structure, but the way they use the structure is quite different. Some of the fields within CRTblDesc are not specified at create time for use with DbiCreateTable; they are specified only with DbiDoRestructure to modify the table.

CRTblDesc for creating a table

The CRTblDesc structure defines the general attributes of the table and supplies pointers to arrays of field, index, and other descriptors. The following CRTblDesc structure defines the table structure:

Field	Type	Description
szTblName	DBITBLNAME	Specifies the table name, including optional path and extension.
szTblType	DBINAME	Specifies the driver type.
szErrTblName	DBIPATH	Reserved.
szUserName	DBINAME	Reserved.
szPassword	DBINAME	Specifies the master password (if *bProtected* is TRUE). (Paradox only.)
bProtected	BOOL	TRUE if encryption is desired (Paradox only).
iFldCount	UINT16	Specifies the number of field definitions supplied.
pfldDesc	pFLDDesc	Specifies the array of field descriptors.
iIdxCount	UINT16	Specifies the number of index definitions supplied.
pidxDesc	pIDXDesc	Specifies the array of index descriptors.
iSecRecCount	UINT16	Specifies the number of security definitions given (Paradox only).
psecDesc	pSECDesc	Specifies the array of security descriptors (Paradox only).
iValChkCount	UINT16	Specifies the number of validity checks (Paradox and SQL only).
pvchkDesc	pVCHKDesc	Specifies the array of validity check descriptors (Paradox and SQL only).
iRintCount	UINT16	Specifies the number of referential integrity specifications (Paradox and SQL only).
printDesc	pRINTDesc	Specifies the array of referential integrity specifications (Paradox, dBASE and SQL only).
iOptParams	UINT16	Specifies the number of optional parameters.
pfldOptParams	pFLDDesc	Specifies the array of field descriptors for optional parameters.
pOptData	pBYTE	Specifies the values of optional parameters.

CRTblDesc for restructuring a table

A complete description of CRTblDesc, as used to restructure a table is described below.

Type	Name	Description
DBITBLNAME	*szTblName*	Required; specifies the source table name. The table name can contain an extension.

DBINAME	*szTblType*	If specified, it must match the driver type associated with the source table.
DBIPATH	*szErrTblName*	Not currently used.
DBINAME	*szUserName*	Not currently used.
DBINAME	*szPassword*	Optional; if bProtected is set to TRUE, specifies the password of the destination table.
BOOL	*bProtected*	Optional; If TRUE, specifies that a master password is supplied for the destination table. Paradox only.
BOOL	*bPack*	Optional; If TRUE, specifies packing for restructure.

iFldCount, *pecrFldOp*, and *pfldDesc* are required to describe the new record structure:

UINT16	*iFldCount*	Optional; used if the record structure is changing. Specifies the number of field operators and field descriptors passed in *pecrFldOp* and pFldDesc for the new record structure.
pCROpType	*pecrFldOp**	Optional; pointer to an array of CROpType structures, one for each field in the record; used if the record structure is changing, it indicates how the fields are rearranged. For each changed field, set it to crADD if the field is added, crMODIFY if the field is modified, or crCOPY if the field is moved.
pFLDDesc	*pfldDesc*	Optional; used if the record structure is changing. Specifies an array of physical field descriptors for the new record structure. *ifldNum* in each pfldDesc must be 0 if the field is added. Otherwise, it must contain the field position (1 to n) in the old record structures. If a field is dropped, its descriptor is simply left out of the new record structure. Additionally, any changes to dependent objects are made automatically (that is, all single field indexes, validity checks, and auxiliary passwords are dropped).

For all the following objects, only the changes must be input:

UINT16	*IdxCount*	Optional; specifies the number of index operators and index descriptors passed in pIdxDesc.
pCROpType	*pecrIdxOp*	Optional; to change an index, specify crADD, crMODIFY, crREDO, or crDROP.
pIDXDesc	*pidxDesc*	Optional; specifies an array of index descriptors.
UINT16	*iSecRecCount*	Optional; for Paradox only; specifies the number of security definitions passed in psecDesc.
pCROpType	*pecrSecOp*	Optional; to change a security definition, specify crADD, crMODIFY, or crDROP.
pSECDesc	*psecDesc*	Optional; for Paradox only; specifies an array of security descriptors.
UINT16	*iValChkCount*	Optional; for Paradox only; specifies the number of validity checks passed in *pecrValChkOp* and *pvchkDesc*.
pCROpType	*pecrValChkOp*	Optional; for Paradox only; to change a validity check, specify crADD, crMODIFY, or crDROP.
pVCHKDesc	*pvchkDesc*	Optional; for Paradox only; specifies an array of validity check descriptors.

UINT16	*iRintCount*	Optional; for Paradox only; specifies the number of referential integrity operators passed in printDesc.
pCROpType	*pecrRintOp*	Optional; for Paradox only; to change a referential integrity operator, specify crADD, crMODIFY, or crDROP. crMODIFY cannot be used to change the name of a referential integrity constraint. To modify the name, use crDROP and crADD.
pRINTDesc	printDesc	Optional; for Paradox only; specifies an array of referential integrity specifications.
UINT16	*iOptParams*	Optional; specifies the number of optional parameters (for example, language driver information).
pFLDDesc	*pfldOptParams*	Optional; specifies an array of field descriptors for optional parameters.
pBYTE	*pOptData*	Optional; specifies values of optional parameters.

The following operation types are valid only for restructuring the table:

Operation type	Value	Description
crNOOP	0	Perform no operation
crADD	1	Add a new element
crCOPY	2	Copy an existing element
crMODIFY	3	Modify an element
crDROP	4	Removes an element

CURProps (cursor properties)

The cursor properties (CURProps) structure describes the most commonly used cursor properties, using the following fields:

Field	Type	Description
szName	DBITBLNAME	Specifies the table name.
iFNameSize	UINT16	Specifies the size of the buffer needed to retrieve full table name (including extension and path, if applicable).
szTableType	DBINAME	Specifies the driver type.
iFields	UINT16	Specifies the number of fields in the table. The client must allocate a buffer whose size is: [iFields * sizeof(FLDDesc)] in order to get the field descriptors for the table.
iRecSize	UINT16	Specifies the record size, depending on the xltMODE for the cursor. If the xltMODE is xltFIELD, *iRecSize* specifies the logical record size. In other words, it is the size of the record if all fields were represented as BDE logical types. If the xltMODE is xltNONE, *iRecSize* specifies the physical record size.
iRecBufSize	UINT16	Specifies the physical record size. This is the size of the record buffer that the client must allocate in order to retrieve the records using DbiGetNextRecord, DbiGetPriorRecord, and other functions. This size can change if DbiSetFieldMap is called.
iKeySize	UINT16	Specifies the key size of the current active index (if any). This is the size of the key buffer that the client must allocate in order to retrieve a key using DbiExtractKey. This size changes if DbiSwitchToIndex is called.
iIndexes	UINT16	Specifies the number of currently open indexes for this cursor. The client can call DbiGetIndexDesc with iIndexSeqNo set from 1 to iIndexes, to have all the index descriptors returned. The client could also allocate a buffer whose size is [iIndexes * sizeof(IDXDesc)] and have all the index descriptors returned by calling DbiGetIndexDescs.
iValChecks	UINT16	Specifies the number of validity checks existing for this table.
iRefIntChecks	UINT16	Specifies the number of referential integrity constraints existing for this table.
iBookMarkSize	UINT16	Specifies the size of the bookmark. Bookmarks are always allocated by the client before DbiGetBookMark is called. Note that the size of the bookmark could change if DbiSwitchToIndex is called.
bBookMarkStable	BOOL	TRUE, if this cursor supports stable bookmarks. Stable bookmarks are those that remain unchanged after another user has modified the table. For example, this value is TRUE for Paradox tables having a primary key, but FALSE for Paradox heap tables.
eOpenMode	DBIOpenMode	Specifies the open mode that this cursor was opened

		with.
eShareMode	DBIShareMode	Specifies the share mode that this cursor was opened with:
bIndexed	BOOL	This value is TRUE if there is a current active index for this cursor. In other words, it is TRUE if there is a non-default order associated with this cursor.
iSeqNums	INT16	This is an enumerated value which is interpreted as follows: 1 This cursor supports the sequence number concept (Paradox). 0 This cursor supports the record number concept (dBASE and FoxPro). < 0 (-1, -2. . .): None (SQL and Access)
bSoftDeletes	BOOL	This value is set to TRUE if this cursor supports soft deletes (dBASE and FoxPro only).
bDeletedOn	BOOL	This value is set to TRUE if the curSOFTDELETEON property is TRUE. This field makes sense only if the cursor supports the soft delete concept. If TRUE, deleted records can be seen while using this cursor (dBASE and FoxPro only).
iRefRange	UINT16	Not currently used.
exltMode	XLTMode	Specifies the value of the translate mode property for this cursor.
iRestrVersion	UINT16	Specifies the restructure version number for the table. (Paradox only.)
bUniDirectional	BOOL	This value is set to TRUE if this cursor is unidirectional (SQL only.)
eprvRights	PRVType	Specifies an enumerated value that gives the table-level rights for the user who opened the table.
iFmlRights	UINT16	Reserved.
iPasswords	UINT16	Specifies the number of auxiliary passwords for this table. (Paradox only).
iCodePage	UINT16	Specifies the code page associated with the table. If the code page is unknown, the value is 0.
bProtected	BOOL	This value is set to TRUE if the table is protected by a password.
iTblLevel	UINT16	Specifies the table level. This value is driver dependent.
szLangDriver	DBINAME	Specifies the name of the language driver associated with the table.
bFieldMap	BOOL	This value is set to TRUE if a field map is active for this cursor.
iBlockSize	UINT16	Specifies the value of the BLOCKSIZE for the table, in bytes.
bStrictRefInt	BOOL	This value applies only to Paradox for DOS tables and the Paradox engine. If TRUE, it means that a referential integrity check has been specified and that the STRICT bit is set in the header, which makes the table inaccessible using Paradox for DOS.

iFilters	UINT16	Specifies the number of filters currently on the cursor.
bTempTable	BOOL	TRUE, if the cursor is on a temporary table. For queries, this means the result set is canned, rather than live. This field can be examined to determine whether the requested preference for LIVENESS in the DbiSetProp call were honored.

eOpenMode
The following open modes are valid:

Open Mode	Description
Read and write (default)	
Read-only	

eShareMode

The following share modes are valid:

Share Mode	Description
Open shared (default)	
Open exclusive	

Note: This might not always be the same value used by the client to call DbiOpenTable. In particular, dbiOPENSHARED can be promoted to dbiOPENEXCL in some cases.

exltMode
The translate mode values supported are:

Translate Mode	Description
	No translation; use physical types
	Field-level translation; use logical types

eprvRights
The table-level rights supported are:

Privilege	Description
No privileges	
Read-only table or field	
Read and modify fields	
Insert	all of above
Delete	all of above
Full rights	
Unknown	

DBDesc (database descriptor)

The DBDesc structure describes a database, using the following fields:

Field	Type	Description
szName	DBINAME	Specifies the database alias name.
szText	DBINAME	Descriptive text.
szPhyName	DBIPATH	Specifies the physical name/path.
szDbType	DBINAME	Specifies the database type.

DBIEnumFld (enumerated field information)

The DBIEnumFld structure defines an enumerated field, using the following fields:

Field	Type	Description
szDisplayStr	DBINAME	Specifies the display string for the value
abVal [DBIMAXENUMFLDLEN+1]	BYTE	Specifies the data value

DBIEnv (environment information)

The DBIEnv structure defines the BDE environment, using the following fields:

Field	Type	Description
szWorkDir	DBIPATH	Specifies the working directory.
szIniFile	DBIPATH	Specifies the fully qualified file name of the configuration file.
bForceLocalInit	BOOL	If TRUE, forces local initialization.
szLang	DBINAME	Specifies the language of the client. This value is the primary language ID from WIN32 (as shown in WINNT.H).
szClientName	DBINAME	Specifies the client name.

szLang

szLang is part of the DBIEnv structure which is passed to DbiInit. The language of the client is specified as the primary language ID from WIN32 (as shown in WINNT.H).

Note: You must add two leading zero's to this value.

For example, the primary language ID for French is "0c". Thus, to start BDE so that it uses French messages and French QBE keywords, you would add two leading zero's to 0c and set *szLang* equal to "000c".

Here is a table of possible *szlang* values :

Language	**szLang value**
Danish	0006
English	0009
French	000c
German	0007
Italian	0010
Norwegian	0014
Portuguese	0016
Spanish	000a
Swedish	001d

DBIErrInfo (error information)

The DBIErrInfo structure describes error information, using the following fields:

Field	Type	Description
iError	DBIResult	Specifies the last error code returned.
szErrCode	DBIMSG	Specifies the error code.
szContext1	DBIMSG	Specifies the context-dependent information at the top level of the error stack.
szContext2	DBIMSG	Specifies the context-dependent information at the second level of the error stack.
szContext3	DBIMSG	Specifies the context-dependent information at the third level of the error stack.
szContext4	DBIMSG	Specifies the context-dependent information at the fourth level of the error stack.

DBIQryProgress (query progress)

The DBIQryProgress structure describes the status of a query, using the following fields:

Field	Type	Description
stepsInQry	UINT16	Specifies the total number of steps in the query.
stepsCompleted	UINT16	Specifies the number of steps completed out of the total.
totElemInStep	UINT32	Specifies the total number of elements in the current step.
elemCompleted	UINT32	Specifies the number of elements completed in the current step.

DRVType (driver capabilities)

The DRVType structure describes the driver and its capabilities, using the following fields:

Field	Type	Description
szType	DBINAME	Specifies the symbolic name identifying the driver.
szText	DBINAME	Descriptive text.
edrvCat	DRVCat	Specifies the driver category.
bTrueDb	BOOL	If TRUE, the driver supports the true database concept.
szDbType	DBINAME	Specifies the database type.
bMultiUser	BOOL	If TRUE, the driver supports multiuser access.
bReadWrite	BOOL	If TRUE, the driver supports read-write access; otherwise, the driver supports only read-only access.
bTrans	BOOL	If TRUE, the driver supports transactions.
bPassThruSQL	BOOL	If TRUE, the driver supports passthrough SQL.
bLogIn	BOOL	If TRUE, the driver requires explicit login.
bCreateDb	BOOL	If TRUE, the driver can create a database.
bDeleteDb	BOOL	If TRUE, the driver can drop a database.

edrvCat
The following driver categories are valid:

Driver Category	Description
drvFILE	File-based (Paradox, dBASE, FoxPro, Access, Text)
drvOTHERSERVER	Other kind of server
drvSQLBASEDSERVER	SQL-based server

FILEDesc (file descriptor)

The FILEDesc structure describes a file, using the following fields:

Field	Type	Description
szFileName	DBIPATH	File name (no directory or extension).
szExt	DBIEXT	Specifies the file extension.
bDir	BOOL	If TRUE, this file is a directory.
iSize	UINT32	Specifies the file size in bytes.
dtDate	DBIDATE	Specifies the date on the file.
tmTime	TIME	Specifies the time on the file.

FILTERInfo (filter information descriptor)

The FILTERInfo structure describes a filter using the following fields:

Field	Type	Description
iFilterId	UINT16	Specifies the ID for the filter.
hFilter	hBBIFilter	Specifies the filter handle.
iClientData	UINT32	Not used.
iPriority	UINT16	Not used.
bCanAbort	BOOL	Not used.
pfFilter	pfGENFilter	Not used.
pCanExpr	pVOID	Specifies the supplied expression.
bActive	BOOL	TRUE, if the filter is active.

FLDDesc (field descriptor)

The FLDDesc structure defines a field in a table, using the properties in the following table:

Note: The same descriptor structure is used both in creating a table and in inquiring about the table structure after it is opened. The application developer does not specify the last five properties in the field descriptor structure when a table is created.

Field	Type	Description
iFldNum	UINT16	On input, specifies the field number. This value can be from 1 to *curProps.iFields*. On output, this is the invariant field ID. **Note:** Do *not* use this value as a field number.
szName	DBINAME	Specifies the name of the field. **Note:** ADT field names do not include the parent names. For exampe, 'ID.BIRTHDAY.NAME' will be "NAME'. Use DbiGetProp() with curFULLFIELDNAME to obtain the full name of an ADT field.
iFldType	UINT16	Specifies the type of the field. In output mode, if translate mode is set to xltNONE, field types represent the physical types of that driver type, otherwise, the types are BDE logical types.
iSubType	UINT16	Specifies the subtype of the field. This could be a BDE logical subtype or a driver physical subtype depending on the translate mode setting.
iUnits1	INT16	Specifies the number of characters, digits, and so on. The interpretation of this field can be dependent on the driver and also on the specific field type. For most drivers, if the field is of the numeric type, *iUnits1* is the precision and *iUnits2* is the scale.
iUnits2	INT16	Specifies the number of decimal places, and so on. The interpretation of this field can depend on the driver and also on the specific field type. For most drivers, if the field is of the numeric type, *iUnits1* is the precision and *iUnits2* is the scale.
iOffset	UINT16	Reports the offset of this field in the record buffer. This offset depends on the translation mode; it could be the offset in the physical or logical representation of the record. This field applies only to existing tables; it is not applicable when a table is created.
iLen	UINT16	Reports the length in bytes of this field. The length depends on the translation mode; that is, it could be the length of the logical or physical representation of the field. The application developer uses this value to allocate a buffer in which to retrieve the field value. This field applies only to existing tables; it is not applicable when a table is created.
iNullOffset	UINT16	Reports the offset of the NULL indicator for this field in the record buffer. If zero, there is no NULL indicator. Otherwise, *iNullOffset* is the offset to an INT16 value, which is -1 if the field is NULL. This field applies only to existing tables; it is not specified when a table is created.
efldvVchk	FLDVchk	Reports the types of validity checks associated with this field

		(this field applies only to existing tables; it is not specified when a table is created). The following validity check types can be reported: fldvNOCHECKS, fldvHASCHECKS, or fldvUNKOWN.
efldrRights	FLDRights	Reports the field level rights for this user (this field applies only to existing tables; it is not specified when a table is created). Field rights can be one of the following values: fldrREADWRITE, fldrREADONLY, fldrNONE, or fldrUNKOWN.
bCalcField	BOOL16	TRUE, if field is a calculated field (computed).
iUnUsed3	UINT16	Reserved for future use

BLOBParamDesc (BLOB or long string descriptor)

The BLOBParamDesc structure is used with FLDDesc to bind a BLOB or long string (greater than 255 characters) parameter in a query, using the properties in the following table:

Field	Type	Description
pBlobBuffer	pBYTE	Pointer to a client-allocated buffer containing the BLOB or long string to bind.
ulBlobLen	UINT32	Specifies the length of the buffer referenced by *pBlobBuffer*.
iUnUsed4	UINT16	Reserved for future use.

FLDType (field types)

The FLDType structure describes a field type using the following fields:

Field	Type	Description
iId	UINT16	Specifies the ID of the field type.
szName	DBINAME	Specifies the symbolic name of field type; for example, ALPHA.
szNativeName	DBINAME	Name used in SQL DDL statements.
szText	DBINAME	Descriptive text.
iPhyType	UINT16	Specifies the physical/native type.
iXltType	UINT16	Specifies the default translated type.
iXltSubType	UINT16	Specifies the default translated subtype.
iMaxUnits1	UINT16	Specifies the maximum units allowed (1).
iMaxUnits2	UINT16	Specifies the maximum units allowed (2).
iPhySize	UINT16	Specifies the physical size in bytes (per unit).
bRequired	BOOL	If TRUE, supports required option.
bDefaultVal	BOOL	If TRUE, supports user-specified default.
bMinVal	BOOL	If TRUE, the field supports the minimum validity constraint.
bMaxVal	BOOL	If TRUE, the field supports the maximum validity constraint.
bRefIntegrity	BOOL	If TRUE, the field can participate in referential integrity.
bOtherChecks	BOOL	If TRUE, the field supports other kinds of checks.
bKeyed	BOOL	If TRUE, the field type can be keyed.
bMultiplePerTable	BOOL	If TRUE, the table can have more than one of this type.
iMinUnits1	UINT16	Specifies the minimum units required (1).
iMinUnits2	UINT16	Specifies the minimum units required (2).
bCreateable	BOOL	If TRUE, the field type can be created.

FMLDesc (family language driver descriptor)

Files belonging to a given table are considered a "family" that must be kept together. FMLDesc returns the filenames of the files in a language driver family.

Field	Type	Description
szName	DBINAME	Member name (documentary)
iId	UINT16	Id (if applicable)
eType	FMLType	Member type
szFileName	DBIPATH	File name of member

FMTBcd (binary coded decimal format)

The FMTBcd structure describes the format for binary coded decimal, using the following fields:

Field	Type	Description
iPrecision	BYTE	Any specified number between 1 to 64 is considered valid.
iSignSpecialPlaces	BYTE	Specifies the following values: sign bit on: negative number special bit on: number is blank places: number of decimals (0 to iPrecision).
iFraction[32]	BYTE	Specifies an array of BCD nibbles, 00 to 99 per byte, high nibble first. The number of significant nibbles in *iFraction* is *iPrecision*; the rest are ignored.

See Also

DbiBcdToFloat, DbiBcdFromFloat

FMTDate (date format)

The FMTDate structure describes the date format for the session, using the following fields:

Field	Type	Description
szDateSeparator[4]	CHAR	Specifies the date separator character.
iDateMode	INT8	Specifies the date format: 0 = MDY, 1 = DMY, 2=YMD.
bFourDigitYear	INT8	If TRUE, write year as four digits.
bYearBiased	INT8	If TRUE, on input add 1900 to year.
bMonthLeadingZero	INT8	If TRUE, the month is displayed with a leading zero.
bDayLeadingZero	INT8	If TRUE, the day is displayed with a leading zero.

FMTNumber (number format)

The FMTNumber structure describes the number format for the current session, using the following fields:

Field	Type	Description
cDecimalSeparator	CHAR	Specifies the character to be used as the decimal separator (for example, ".").
cThousandSeparator	CHAR	Specifies the character to be used as the thousands separator (for example, ",").
iDecimalDigits	INT8	Specifies the number of decimal digits.
bLeadingZero	INT8	If TRUE, use leading zeros.

FMTTime (time format)

The FMTTime structure describes the time format for the current session, using the following fields:

Field	Type	Description
cTimeSeparator	CHAR	Specifies the time separator character (for example, ".").
bTwelveHour	INT8	If TRUE, represent as 12-hour time.
szAmString[6]	CHAR	Specifies the string to use for designating AM time (only for 12-hour time).
szPmString[6]	CHAR	Specifies the string to use for designating PM time (only for 12-hour time).
bSeconds	INT8	If TRUE, show seconds.
bMilSeconds	INT8	If TRUE, show milliseconds.

DBIFUNCArgDesc (argument descriptor)

The DBIFUNCArgDesc structure describes the arguments to a remote data source function, using the following fields:

Field	Type	Description
uArgNum	UINT16	Argument position number. 0 for fn return
uFldType	UINT16	Field type
uSubType	UINT16	Field subtype. if applicable.
ufuncFlags	UNIT16	Function flags

DBIFUNCDesc (function descriptor)

The DBIFUNCDesc structure describes a remote data source function, using the following fields:

Field	Type	Description
szName	DBINAME	Remote function name
szDesc[255]	CHAR	Short description
uOverload	UINT16	Number of function overloads
eStdFn	DBISTDFuncs	Corresponding to DBI standard function

IDXDesc (index descriptor)

The IDXDesc structure describes each index in a table. The same structure is used both in creating an index and inquiring about the index after a cursor is opened. The application does not specify the following fields in the index descriptor structure when creating an index: *iRestrNum*, *bOutofDate*, and *iKeyLen*.

The fields required in this structure vary by driver type and index type.

Note: The first three fields, *szName*, *iIndexId*, and *szTagName* are used to identify the index. A different combination of these three fields is used, depending on the driver type and on the specific index type. The rules are given below:

Driver Type	Index Type
Access	Either *iIndexId* or *szName* identifies the index.
dBASE	.NDX style: *szName* alone identifies the index.
	.MDX style: *szName* and *szTagName* together identify the index.
FoxPro	.CDX style: *szName* and *szTagName* together identify the index.
Paradox	Either *iIndexId* or *szName* identifies the index.
Text driver	Indexing not supported.
All SQL drivers	*szName* alone identifies the index. *pszIndexName* may be used to identify a pseudo-index.

Field	Type	Description
szName	DBITBLNAME	Specifies the index name.
iIndexId	UINT16	Specifies the number identifying the index.
szTagName	DBINAME	Specifies the index tag name. Supported for dBASE and FoxPro only.
szFormat	DBINAME	Currently, for information only. Describes the physical index format type (for example, BTREE or HASH).
bPrimary	BOOL16	TRUE, if the key is primary.
bUnique	BOOL16	TRUE, if the key is unique.
abDescending	BOOL16	An array of booleans describing whether the corresponding field in *aiKeyFld* is descending. Used only when *bDescending* is set to TRUE.
bDescending	BOOL16	TRUE, if the key is descending.
bMaintained	BOOL16	TRUE, if the key is maintained.
bSubset	BOOL16	TRUE, if the index is a subset index. Supported for dBASE and FoxPro only.
bExpIdx	BOOL16	TRUE, if the index is an expression index. Supported for dBASE and FoxPro only.
iCost	UINT16	Reserved.
iFldsInKey	UINT16	Specifies the number of key fields in a composite index. If the index is an expression, set to 0.
iKeyLen	UINT16	Not specified while index is created. Specifies the physical length of the key in bytes. The application developer needs to allocate a buffer of iKeyLen bytes to use as a key buffer. A key buffer is used with functions such as DbiExtractKey and DbiSetToKey.

bOutofDate	BOOL16	Not specified while index is created; TRUE, if the index is out-of-date.
iKeyExpType	UINT16	Specifies the type of the key expression (dBASE and FoxPro only). This value can be one of the following: fldDBCHAR, fldDBKEYNUM, or fldDBKEYBCD.
aiKeyFld	DBIKEY	Specifies an array of field numbers in the key.
szKeyExp	DBIKEYEXP	Specifies the key expression for an expression index (dBASE and FoxPro only). This field is used only if *bExpIdx* = TRUE. The expression is stated as a dBASE expression.
szKeyCond	DBIKEYEXP	Specifies the expression that defines the subset condition (dBASE and FoxPro only). This field is used only if bSubset = TRUE. The expression is stated as a dBASE expression.
bCaseInsensitive	BOOL16	TRUE, if the index is case-insensitive.
iBlockSize	UINT16	Specifies the internal block size in bytes for this index.
iRestrNum	UINT16	Not specified while index is created. Specifies the internal restructure number for this index. This number is set when the index descriptor is retrieved and should not be changed when passing the descriptor back to DbiDoRestructure.

Note: The following four fields, explained in detail above, are used to describe the key for an index: *iFldsInKey*, *aiKeyFld*, *bExpIdx*, *szKeyExp*. The key is described by specifying either one of the following combinations:

For traditional indexes	**For expression indexes**
iFldsInKey and *aiKeyFld*	*bExpIdx* and *szKeyExp*

See Also

IDXDesc characteristics by driver

szName

The following table describes how to name Paradox indexes:

Index ID Param	Non-composite index	Composite index
szName	Same as field name (only if the secondary index is a case-sensitive index)	Can be any legal name not used as a field name; must be unique
iIndexID	Same as field number (1 to 255)	Valid ID (256 to 511) Output only; not specified while index is created

IDXDesc characteristics by driver

dBASE and FoxPro

For dBASE production .MDX indexes, the IDXDesc structure requires the following elements:

Type	Name	Value	Description
DBITBLNAME	*szName*	Ignored	
UINT16	*iIndexId*	Ignored	
DBINAME	*szTagName*	10 character name	Name for index; required
DBINAME	*szFormat*	Ignored	
BOOL	*bPrimary*	FALSE	Is primary; required
BOOL	*bUnique*	TRUE/FALSE	Is unique; required
BOOL	*bDescending*	TRUE/FALSE	Is descending; required
BOOL	*bMaintained*	TRUE	Is maintained; required
BOOL	*bSubset*	TRUE/FALSE	Is subset; required
BOOL	*bExpIdx*	TRUE/FALSE	Is expression; must be TRUE if *iFldsInKey* is not supplied
UINT16	*iCost*	Ignored	
UINT16	*iFldsInKey*	1	Number of fields in composite index; required if *bExpIdx* is FALSE
UINT16	*iKeyLen*	Ignored	
BOOL	*bOutofDate*	Ignored	
UINT16	*iKeyExpType*	Ignored	
DBIKEY	*aiKeyFld*	Field number; required if *iFldsInKey* is supplied and *bExpIdx* is FALSE	
DBIKEYEXP	*szKeyExp*	Expression for expression index; required if *bExpIdx* is set to TRUE	
DBIKEYEXP	*szKeyCond*	Condition for subset index; required if *bSubset* is set to TRUE	
BOOL	*bCaseInsensitive*	FALSE	Case-insensitive index; required
UINT16	*iBlockSize*	Block size in bytes; optional	
UINT16	*iRestrNum*	Ignored	
UINT16	*iUnUsed[16]*	Unused space	

dBASE production .MDX indexes have the following general characteristics:

- A production index is considered part of the family.
- Up to 47 indexes are allowed in a production .MDX file.
- A production index is always maintained.
- A production index can be ascending or descending.
- If *szName* is NULL and *bMaintained* is TRUE, reference to the production index is assumed.

For dBASE non-maintained .NDX style indexes, the IDXDesc structure requires the following elements:

Type	Name	Value	Description
DBITBLNAME	*szName*	Any 8-character name and extension. Extension is usually .NDX.	Index name; required
UINT16	*iIndexId*	Ignored	
DBINAME	*szTagName*	Ignored	
DBINAME	*szFormat*	Ignored	
BOOL	*bPrimary*	FALSE	Is primary; required
BOOL	*bUnique*	TRUE/FALSE	Is unique; required
BOOL	*bDescending*	FALSE	Is descending; required
BOOL	*bMaintained*	FALSE	Is maintained; required
BOOL	*bSubset*	Ignored	
BOOL	*bExpIdx*	TRUE/FALSE	Is expression; must be TRUE if *iFldsInKey* is not supplied
UINT16	*iFldsInKey*	1	Number of fields in composite index; required if *bExpIdx* is FALSE
UINT16	*iKeyExpType*	Ignored	
DBIKEY	*aiKeyFld*	Field number; required if *iFldsInKey* is given and *bExpIdx* is FALSE	
DBIKEYEXP	*szKeyExp*	Expression for expression index; required if *bExpIdx* is TRUE	
DBIKEYEXP	*szKeyCond*	Ignored	
BOOL	*bCaseInsensitive*	FALSE	Case-insensitive index; required
UINT16	*iBlockSize*	Ignored	
UINT16	*iRestrNum*	Restructure number	
UINT16	*iUnUsed*[16]	Unused space	

dBASE non-maintained .NDX style indexes have the following general characteristics:

- The non-maintained index cannot have the same file name and extension as the production index.
- Only one index is allowed per index file.
- Only ascending order is allowed.
- The index is maintained only when the index is opened.
- Only the index name is required to use the index.

Naming convention: any file name and extension can be used except <*tbl_name*>.MDX. Or, if the table is named <*tbl_name*>.111, the index cannot be named <*tbl_name*>.11X (this name is reserved for production indexes).

For dBASE non-maintained .MDX-style indexes and FoxPro .CDX-style compressed indexes (version 2.0, 2.5, and 2.6), the IDXDesc structure has the following elements:

Type	Name	Value	Description
DBITBLNAME	*szName*	<*any_name*>.MDX	Index name; required

		except <*tbl_name*>.MDX **or** <*any_name*>.CDX except <*tbl_name*>.CDX	
UINT16	*iIndexId*	Ignored	
DBINAME	*szTagName*	Name for index; required	
DBINAME	*szFormat*	Ignored	
BOOL	*bPrimary*	FALSE	Is primary; required
BOOL	*bUnique*	TRUE/FALSE	Is unique; required
BOOL	*bDescending*	TRUE/FALSE	Is descending; required
BOOL	*bMaintained*	FALSE	Is maintained; required
BOOL	*bSubset*	TRUE/FALSE	Is subset; required
BOOL	*bExpIdx*	TRUE/FALSE	Is expression; must be TRUE if *iFldsInKey* is not supplied
UINT16	*iFldsInKey*	1	Number of fields in composite index; required if *bExpIdx* is FALSE
UINT16	*iKeyExpType*	Ignored	
DBIKEY	*aiKeyFld*	Field number in composite index; required if *iFldsInKey* is supplied and *bExpIdx* is FALSE	
DBIKEYEXP	*szKeyExp*	Expression for expression index; required if *bExpIdx* is set to TRUE	
DBIKEYEXP	*szKeyCond*	Condition for subset index; required if *bSubset* is set to TRUE	
BOOL	*bCaseInsensitive*	FALSE	Case insensitive index; required
UINT16	*iBlockSize*	Block size in bytes; optional	
UINT16	*iRestrNum*	Restructure number	
UINT16	*iUnUsed*[16]	Unused space	

Special case, non-maintained dBASE .MDX indexes have the following characteristics:

- Up to 47 indexes are allowed in a single .MDX file.
- All indexes are maintained when the .MDX file is opened.
- Both ascending and descending order may be used.
- Index subset conditions are available.
- Both the index name and the tag name are required to use the index.

Naming convention: <*any_name*>.MDX, except <*tbl_name*>.MDX, or <*tbl_name*>.11X where the table is called <*tbl_name*>.111.

Paradox

For Paradox primary indexes (*bPrimary* = TRUE), the IDXDesc structure has the following elements:

Type	Name	Value	Description
DBITBLNAME	*szName*	Ignored	
UINT16	*iIndexId*	Must be 0	
DBINAME	*szTagName*	Ignored	
DBINAME	*szFormat*	Ignored	
BOOL	*bPrimary*	TRUE	Is primary; required
BOOL	*bUnique*	TRUE	Is unique; required
BOOL	*bDescending*	FALSE	Is descending; required
BOOL	*bMaintained*	TRUE	Is maintained; required
BOOL	*bSubset*	FALSE	Is subset; required
BOOL	*bExpIdx*	FALSE	Is expression; required
UINT16	*iFldsInKey*	1 to 16	Number of fields in composite index; required
UINT16	*iKeyExpType*	Ignored	
DBIKEY	*aiKeyFld*	Array of field numbers in composite index; required	
DBIKEYEXP	*szKeyExp*	Ignored	
DBIKEYEXP	*szKeyCond*	Ignored	
BOOL	*bCaseInsensitive*	FALSE	Case-insensitive index; required
UINT16	*iBlockSize*	Block size in bytes	
UINT16	*iRestrNum*	Restructure number	
UINT16	*iUnUsed*[16]	Unused space	

Paradox primary indexes have the following general characteristics:

- The index must start from the first field.
- The index is always case-sensitive, maintained, and unique.

The following invalid settings for Paradox primary indexes (*bPrimary* = TRUE) return an error message to the application:

Setting	Solution
iIndexId is not set to 0	Primary ID must be is set to 0.
bUnique is set to FALSE	Must be TRUE.
bDescending is set to TRUE	Must be FALSE.
bMaintained is set to FALSE	Must be TRUE.
bSubset is set to TRUE	Must be FALSE.
bExpIdx is set to TRUE	Must be FALSE.
iFldsInKey is 0 or greater than 16	Must be 1 to 16.
aiKeyFld	BLOB fields can't be indexed; defined fields must be available; field cannot be used more than once in index.
szKeyCond is not set to NULL	Not available.
bCaseInsensitive is set to TRUE	Must be FALSE.

For Paradox 3.5-style indexes, the IDXDesc structure has the following elements:

Type	Name	Value	Description
DBITBLNAME	*szName*	Ignored	
UINT16	*iIndexId*	Ignored	
DBINAME	*szTagName*	Ignored	
DBINAME	*szFormat*	Ignored	
BOOL	*bPrimary*	FALSE	Is primary; required
BOOL	*bUnique*	FALSE	Is unique; required
BOOL	*bDescending*	FALSE	Is descending; required
BOOL	*bMaintained*	TRUE/FALSE	Is maintained; required
BOOL	*bSubset*	FALSE	Is subset; required
BOOL	*bExpIdx*	FALSE	Is expression; required
UINT16	*iFldsInKey*	1	Number of fields in index; single field only; required
UINT16	*iKeyExpType*	Ignored	
DBIKEY	*aiKeyFld*	1	Array of field numbers in index; single field only; required
DBIKEYEXP	*szKeyExp*	Ignored	
DBIKEYEXP	*szKeyCond*	Ignored	
BOOL	*bCaseInsensitive*	TRUE/FALSE	Case-insensitive index; required
UINT16	*iBlockSize*	Block size in bytes	
UINT16	*iRestrNum*	Restructure number	
UINT16	*iUnUsed*[16]	Unused space	

Paradox 3.5 style indexes have the following general characteristics:

- Each index can have only one field.
- Up to 255 indexes are allowed.
- Indexes can be maintained (primary is required) or non-maintained.
- The index name becomes the field name.
- The index ID is used to reference the index or the field name.
- If the index is maintained, it is maintained at all times. All indexes are opened when the table is opened.

The following invalid settings for Paradox 3.5 style indexes return an error message to the application:

Setting	Solution
bUnique is set to TRUE	Must be FALSE.
bDescending is set to TRUE	Must be FALSE.
bSubset is set to TRUE	Must be FALSE.
bExpIdx is set to TRUE	Must be FALSE.
iFldsInKey is 0 or greater than 1	Must be 1.
aiKeyFld	BLOB fields can't be indexed; defined fields must be available; field cannot be used more than once in index.

bCaseInsensitive is set to TRUE	Must be FALSE.

For Paradox 4.0, 5.0, and 7.0 style indexes, the IDXDesc structure has the following elements:

Type	Name	Value	Description
DBITBLNAME	*szName*	Index name; required	
UINT16	*iIndexId*	Ignored	
DBINAME	*szTagName*	Ignored	
DBINAME	*szFormat*	Ignored	
BOOL	*bPrimary*	FALSE	Is primary; required
BOOL	*bUnique*	FALSE	Is unique; required
BOOL	*bDescending*	TRUE/FALSE	Is descending; required (TRUE valid for Paradox 7.0 indexes only)
BOOL	*bMaintained*	TRUE/FALSE	Is maintained; required
BOOL	*bSubset*	FALSE	Is subset; required
BOOL	*bExpIdx*	FALSE	Is expression; required
UINT16	*iFldsInKey*	1	Number of fields in index; single field only; required
UINT16	*iKeyExpType*	Ignored	
DBIKEY	*aiKeyFld*	1	Array of field numbers in index; single field only; required
DBIKEYEXP	*szKeyExp*	Ignored	
DBIKEYEXP	*szKeyCond*	Ignored	
BOOL	*bCaseInsensitive*	TRUE/FALSE	Case-insensitive index; required
UINT16	*iBlockSize*	Block size in bytes	
UINT16	*iRestrNum*	Restructure number	
UINT16	*iUnUsed*[16]	Unused space	

Paradox 4.0 and 5.0 style indexes have the following general characteristics:

- Both maintained (primary is required) and non-maintained indexes are allowed.
- Indexes may be case-sensitive, or case-insensitive.
- A composite index can have more than one field.
- Only the index name is required to reference the index
- Up to 320 indexes are allowed.
- If the index is maintained, it is maintained at all times. All indexes are opened when the table is opened.

The following invalid settings for Paradox 4.0 and 5.0 style indexes return an error message to the application:

Setting	Solution
szName is set to NULL	Name required to find index.
bUnique is set to TRUE	Must be FALSE.

bDescending is set to TRUE	Must be FALSE.
bSubset is set to TRUE	Must be FALSE.
bExpIdx is set to TRUE	Must be FALSE.
iFldsInKey is 0 or greater than 16	Must be 1 to 16.
aiKeyFld	BLOB fields can't be indexed; defined fields must be available; field cannot be used more than once in index.

The following naming conventions must be followed for Paradox indexes:

Index Type	**Naming convention**
Non-composite	Same as field name.
Non-composite, case-insensitive	Any legal name. Must be unique. A field name can be used if not already used by a non-composite index.
Composite	Any legal name not used as a field name. Must be unique. A field name can be used if not already used by a non-composite index.

Paradox 7.0 style indexes follow the same conventions as Paradox 4.0 and 5.0 style indexes except *bDescending* can be TRUE or FALSE.

IDXType (index types)

The IDXType structure describes an index type, using the following fields:

Field	Type	Description
iId	UINT16	Specifies the ID of the index type.
szName	DBINAME	Specifies the symbolic name of the index type.
szText	DBINAME	Descriptive text.
szFormat	DBINAME	Optional. Information only about the format (for example, BTREE, HASH).
bComposite	BOOL	If TRUE, supports composite keys.
bPrimary	BOOL	If TRUE, this index type supports a primary index.
bUnique	BOOL	If TRUE, this index type supports unique indexes.
bKeyDescending	BOOL	If TRUE, the key can be descending.
bFldDescending	BOOL	If TRUE, the key can be descending at the field level.
bMaintained	BOOL	If TRUE, this index type supports the maintained option.
bSubset	BOOL	If TRUE, this index type supports the subset expression (dBASE and FoxPro only).
bKeyExpr	BOOL	If TRUE, the key can be an expression (dBASE and FoxPro only).
bCaseInsensitive	BOOL	If TRUE, this index type supports case-insensitive keys.

LDDesc (language driver descriptor)

The LDDesc structure describes a language driver, using the following fields:

Field	Type	Description
szName	DBINAME	Specifies the driver's symbolic name.
szDesc	DBINAME	Specifies the driver description.
iCodePage	UINT16	Specifies the code page number.
PrimaryCpPlatform	UINT16	Unused.
AlternateCpPlatform	UINT16	Unused.

PrimaryCpPlatform
The following table shows valid values:

Value	Description
1	DOS (OEM) platform
2	Windows (ANSI) platform
6	HP UNIX (ROMAN8) platform

LOCKDesc (lock descriptor)

The LOCKDesc structure describes a lock, using the following fields:

Field	Type	Description
iType	UINT16	Specifies the lock type (0 for record lock).
szUserName	DBIUSERNAME	Specifies the user name.
iNetSession	UINT16	Specifies the net level session number.
iSession	UINT16	Specifies the BDE session number, if BDE lock.
iRecNum	UINT32	Specifies the record number for the record lock, if this is a record lock.
iInfo	UINT16	Specifies information for table locks (Paradox only).

iType

The following table shows valid values:

Value	Description
0	Record lock
4	No table lock
5	Table read lock
6	Table write lock
7	Table opened exclusively

Note: Record write lock and record lock are the same. Record write lock is more descriptive.

RECProps (record properties)

The RECProps structure describes the record properties, using the following fields:

Field	Type	Description
iSeqNum	UINT32	Specifies the sequence number of the record. Applicable if the cursor supports sequence numbers (Paradox only).
iPhyRecNum	UINT32	Specifies the record number of the record. Applicable only when physical record numbers are supported (dBASE and FoxPro only).
bRecChanged	UINT16	Determine the currenct status of a record in delayed update mode (i.e. Unmodified = 0, Modified = 1, Inserted = 2, Deleted = 3).
bSeqNumChanged	BOOL	Not currently used.
bDeleteFlag	BOOL	Specifies if the record is deleted. Applicable only when soft delete is supported (dBASE and FoxPro only).

RINTDesc (referential integrity)

The RINTDesc structure describes the referential integrity options for a table (currently Paradox and dBASE only), using the following fields:

Field	Type	Description
iRintNum	UINT16	Specifies the referential integrity number.
szRintName	DBINAME	Specifies the referential integrity name.
eType	RINTType	Specifies the type, either rintMASTER or rintDEPENDENT.
szTblName	DBIPATH	Specifies the other table name.
eModOp	RINTQual	Specifies the modify qualifier, either rintRESTRICT or rintCASCADE.
eDelOp	RINTQual	Specifies the delete qualifier, either rintRESTRICT or rintCASCADE.
iFldCount	UINT16	Specifies the number of fields in the linking key.
aiThisTabFld	DBIKEY	For Paradox, specifies the field numbers that make up this referential integrity constraint in this table. For dBASE, specfies the index ID that make up the referential integrity constraint in this table.
aiOthTabFld	DBIKEY	For Paradox, specifies the number of fields in the other table. For dBASE, specfies the index ID in the other table.

For more information on referential integrity options, see Integrity Constraints

SECDesc (security descriptor)

The SECDesc structure describes each security descriptor in the table (currently, Paradox only), using the following fields:

Field	Type	Description
iSecNum	UINT16	Specifies the number identifying the descriptor.
eprvTable	PRVType	Specifies the table privileges: prvNONE, prvREADONLY, prvMODIFY, prvINSERT, prvINSDEL, prvFULL, prvUNKNOWN.
iFamRights	UINT16	Specifies the family rights: NOFAMRIGHTS, FORMRIGHTS, RPTRIGHTS, VALRIGHTS, SETRIGHTS, ALLFAMRIGHTS.
szPassword	DBINAME	Specifies a NULL terminated string.
aprvFld	PRVType	Specifies the field privileges: prvNONE, prvREADONLY, prvFULL. [DBIMAXFLDSINSEC]

SESInfo (session information)

The SESInfo structure provides information about a session, using the following fields:

Field	Type	Description
iSession	UINT16	Specifies the session ID (1 to n).
szName	DBINAME	Specifies the documentary name of the session.
iDatabases	UINT16	Specifies the number of open databases.
iCursors	UINT16	Specifies the number of open cursors.
iLockWait	INT16	Specifies the lock wait time (in seconds).
szNetDir	DBIPATH	Specifies the directory location for the network control file.
szPrivDir	DBIPATH	Specifies the private directory.

SPDesc (stored procedure information)

The SPDesc structure provides information about a stored procedure, using the following fields:

Field	Type	Description
szName	DBISPNAME	Specifies the documentary name of the stored procedure.
dtDate	DBIDATE	Specifies the date on the stored procedure.
tmTime	TIME	Specifies the time on the stored procedure.
MaxSPNameLen	UINT16	Specifies the maximum stored procedure field name length.

SPParamDesc (stored procedure parameters)

The SPParamDesc structure describes the parameters of a stored procedure, using the following fields:

Field	Type	Description
uParamNum	UINT16	Specifies the parameter number.
szName	DBINAME	Specifies the name of the parameter.
eParamType	STMTParamType	Specifies the type of the parameter.
uFldType	UINT16	Specifies the field type.
uSubType	UINT16	Specifies the sub-type (if applicable).
iUnits1	INT16	Specifies the number of characters and digits.
iUnits2	INT16	Specifies the number of decimal places.
uOffset	UINT16	Specifies the computed offset.
uLen	UINT16	Specifies the computed length in bytes.
uNullOffset	UINT16	Specifies the computed offset for NULL bits.

SYSConfig (system configuration)

The SYSConfig structure provides basic system configuration information, using the following fields:

Field	Type	Description
bLocalShare	BOOL	TRUE, if local files will be shared with non-BDE applications.
iNetProtocol	UINT16	Not currently used.
bNetShare	BOOL	Not currently used.
szNetType	DBINAME	Specifies the network type.
szUserName	DBIUSERNAME	Specifies the network user name.
szIniFile	DBIPATH	Specifies the fully qualified configuration file name.
szLangDriver	DBINAME	Specifies the system language driver.

SYSInfo (system status and information)

The SYSInfo structure provides BDE system status and information, using the following fields:

Field	Type	Description
iBufferSpace	UINT16	Specifies the size of the buffer space in kilobytes.
iHeapSpace	UINT16	Specifies the size of the heap space in kilobytes.
iDrivers	UINT16	Specifies the number of currently loaded drivers.
iClients	UINT16	Specifies the number of active clients.
iSessions	UINT16	Specifies the number of sessions (for all clients).
iDatabases	UINT16	Specifies the number of open databases (for all clients).
iCursors	UINT16	Specifies the number of cursors (for all clients).

SYSVersion (system version information)

The SYSVersion structure provides the BDE system version information, using the following fields:

Field	Type	Description
iVersion	UINT16	Specifies the BDE version.
iIntfLevel	UINT16	Specifies the client interface level.
dateVer	DBIDATE	Specifies the version date.
timeVer	TIME	Specifies the version time.

STMTBaseDesc (statement base table descriptor)

The STMTBaseDesc structure provides basic information about the original columns upon which the result set is based, using the following fields:

Field	Type	Description
szDatabaseName	DBINAME	Specifies the database name.
szTableName	DBITBLNAME	Specifies the table name (no extension or directory).
szFieldName	DBINAME	Specifies the field name.
bExpression	BOOL	When the SQL query has an expression (Select Col1+5...) *bExpression* is set to TRUE and the field is the first field encountered in the expression. Thus the following SQL: (SelectCol1 + Col2 from ':alias:Table') would return a record looking like "alias Table Col1 1 0 0".
bConstant	BOOL	When the SQL query has a constant (Select 'test'...) *bConstant* is set to TRUE and field is the constant value.
bAggregate	BOOL	If the SQL has an aggregate, bAggregate is set to TRUE. The order of the records in the cursor represents the order of the items in the select list. The following SQL, (Select Col3, Col2, Col1...) would result in a table with the records in this order: Col3, Col2, Col1.

TBLBaseDesc (base table descriptor)

The TBLBaseDesc structure provides basic information about a table, using the following fields:

Field	Type	Description
szName	DBITBLNAME	Specifies the table name (no extension or directory).
szFileName	DBITBLNAME	Specifies the file name.
szExt	DBIEXT	Specifies the file extension.
szType	DBINAME	Specifies the driver type.
dtDate	DBIDATE	Specifies the date on the table.
tmTime	TIME	Specifies the time on the table.
iSize	UINT32	Specifies the size in bytes.
bView	BOOL	TRUE, if this a view (SQL only).
bSynonym	BOOL16	TRUE, if the object is a synonym

bSynonym

Synonyms are supported by a new field in the TBLBaseDesc structure called bSynonym. The field bSynonym is a BOOL16, which is set to TRUE if the object is a synonym.

The BDE configuration option LIST SYNONYMS can be found in the Registry's DB OPEN section for Oracle DRIVERS and DATABASES. See BDE Administrator Help

The value of LIST SYNONYMS determines whether or not to include synonyms in the schema table returned from DbiOpenTableList and DbiOpenFileList, as shown in the following table.

Value	Meaning
NONE	Do not include any synonyms (Default)
PRIVATE	Only include private synonyms
ALL	Include both private and public synonyms

Oracle: Oracle has PUBLIC synonyms that show up in the table list when the value of LIST SYNONYMS = ALL. However, to open a PUBLIC synonym, the user must also have SELECT privileges on the base object of the synonym. If the user does not have SELECT privileges and tries to open the PUBLIC SYNONYM, Oracle returns the error "Table or view does not exist".

Oracle has PUBLIC synonyms to a set of dynamic performance tables. Even though these are PUBLIC synonyms, they are accessible only to the DBA user SYS, by default (other users can be granted privileges). These synonym names are in the format, V$... (that is, V$DATABASE, V$ACCESS, and so on).

TBLExtDesc (extended table descriptor)

The TBLExtDesc structure provides additional information about a table, using the following fields:

Field	Type	Description
szStruct	DBINAME	Specifies the physical structure.
iRestrVersion	UINT16	Specifies the version number.
iRecSize	UINT16	Specifies the physical record size.
iFields	UINT16	Specifies the number of fields.
iIndexes	UINT16	Specifies the number of indexes.
iValChecks	UINT16	Specifies the number of field validity checks.
iRintChecks	UINT16	Specifies the number of referential integrity checks.
iRecords	UINT32	Specifies the number of records in table.
bProtected	BOOL	TRUE, if the table is protected.
bValidInfo	BOOL	If FALSE, all or some of the extended data is not available.

TBLFullDesc (full table descriptor)

The TBLFullDesc structure provides a complete description of the table (base extended), using the following fields:

Field	Type	Description
tblBase	TBLBaseDesc	Specifies the base description.
tblExt	TBLExtDesc	Specifies the extended description.

TBLType (table capabilities)

The TBLType structure describes the table's capabilities, using the following fields

Field	Type	Description
iId	UINT16	Specifies the ID of the table type.
szName	DBINAME	Specifies the descriptive name of the table type; for example, dBASE5.
szText	DBINAME	Descriptive text.
szFormat	DBINAME	Specifies the format; for example, HEAP.
bReadWrite	BOOL	If TRUE, the user can read and write.
bCreate	BOOL	If TRUE, the user can create new tables of this type.
bRestructure	BOOL	If TRUE, BDE can restructure a table of this type.
bValChecks	BOOL	If TRUE, the user can specify validity checks for this table type.
bSecurity	BOOL	If TRUE, a table of this type can be protected.
bRefIntegrity	BOOL	If TRUE, a table of this type can participate in referential integrity.
bPrimaryKey	BOOL	If TRUE, a table of this type supports the primary key concept.
bIndexing	BOOL	If TRUE, a table of this type can have indexes.
iFldTypes	UINT16	Specifies the number of physical field types supported.
iMaxRecSize	UINT16	Specifies the maximum record size.
iMaxFldsInTable	UINT16	Specifies the maximum fields in a table.
iMaxFldNameLen	UINT16	Specifies the maximum field name length.
iTblLevel	UINT16	Specifies the driver dependent table level (version).

USERDesc (user information descriptor)

The USERDesc structure describes a user, using the following fields:

Field	Type	Description
szUserName	DBIUSERNAME	Specifies the user name.
iNetSession	UINT16	Specifies the net level session number.
iProductClass	UINT16	Specifies the product class of the user (Paradox only).
szSerialNum[22]	CHAR	Specifies the serial number (Paradox only).

VCHKDesc (validity check)

The VCHKDesc structure provides information about validity checking constraints on a field (Paradox, dBASE, and SQL tables only), using the following fields (*bRequired* is the only option supported by the SQL):

Field	Type	Description
iFldNum	UINT16	Specifies the field number (1 to n). For dBASE, can be zero, where the szPict will then be checked for an expression to be used in the validity check.
bRequired	BOOL	Specifies whether or not the field is required: TRUE, FALSE.
bHasMinVal	BOOL	Has minimum value: TRUE, FALSE, or TODAYVAL.
bHasMaxVal	BOOL	Has maximum value: TRUE, FALSE, or TODAYVAL.
bHasDefVal	BOOL	Has default value: TRUE, FALSE, or TODAYVAL.
aMinVal	DBIVCHK	Specifies the minimum value.
aMaxVal	DBIVCHK	Specifies the maximum value.
aDefVal	DBIVCHK	Specifies the default value.
szPict	DBIPICT	Specifies the picture string.
elkupType	LKUPType	Specifies the lookup type. (Only for Paradox)
szLkupTblName	DBIPATH	Specifies the lookup table name; for information only.

elkupType
The following lookup and fill types are valid for Paradox tables:

Lookup Type	Description
lkupNONE	The table has no lookup.
lkupPRIVATE	Only current field private.
lkupALLCORRESP	All corresponding no help.
lkupHELP	Only current field help and fill.
lkupALLCORRESPHELP	All corresponding help.

Xinfo (Information Transactions)

The XInfo structure describes a transaction, using the following fields:

Field	Type	Description
exState	eXState	Specifies the transaction state: xsACTIVE or xsINACTIVE.
eXIL	eXILType	Specifies the transaction isolation level.
uNests	UINT16	Specifies the transaction children.

eXIL

The following transaction isolation levels are valid:

Isolation Level	Description
xilDIRTYREAD	Uncommitted changes; no phantoms
xilREADCOMMITTED	Committed changes; no phantoms
xilREPEATABLEREAD	Full read repeatability

CANUnary (unary node descriptor)

Type	Name	Description
NODEClass	*nodeClass*	Unary node
CANOp	*canOp*	Operator
UINT16	*iOperand1*	Byte offset of operand

CANBinary (binary node descriptor)

Type	Name	Description
NODEClass	*nodeClass*	Binary node
CANOp	*canOp*	Operator
UINT16	*iOperand1*	Byte offset of operand 1
UINT16	*iOperand2*	Byte offset of operand 2

CANCompare (extended compare node descriptor)

Type	Name	Description
NODEClass	*nodeClass*	Extended compare node
CANOp	*canOp*	Operator
BOOL	*bCaseInsensitive*	3 values: UNKNOWN, "fastest", "native"
UINT16	*iOperand1*	Byte offset of Operand1
UINT16	*iOperand2*	Byte offset of Operand2

CANField (field node descriptor)

Type	Name	Description
NODEClass	*nodeClass*	Field node
CANOp	*canOp*	Operator
UINT16	*iFieldNum*	Field number
UINT16	*iNameOffset*	Name offset in literal pool

CANConst (constant node descriptor)

Type	Name	Description
NODEClass	*nodeClass*	Constant
CANOp	*canOp*	Operator
UINT16	*iType*	Constant type
UINT16	*iSize*	Constant size (in bytes)
UINT16	*iOffset*	Offset in literal pool

CANTuple (tuple node descriptor)

Type	Name	Description
NODEClass	*nodeClass*	Tuple (record)
CANOp	*canOp*	Operator
UINT16	*iSize*	Constant size (in bytes)

CANContinue (break node descriptor)

Type	Name	Description
NODEClass	*nodeClass*	Break node
CANOp	*canOp*	Operator
UINT16	*iContOperand*	Continue if operand is TRUE; otherwise, stop evaluating records.

Callback definitions

The following callbacks are defined in the header file IDAPI.H:

Callback	Description
cbGENPROGRESS	Informs applications about the progress made during large batch operations.
cbRESTRUCTURE	Supplies information about an impending action and requests a response from the caller.
cbBATCHRESULT	Batch processing results.
cbTABLECHANGED	Notifies user that table has changed.
cbCANCELQRY	Allows user to cancel a Sybase query.
cbINPUTREQ	A BDE driver requests input from user.
cbDBASELOGIN	Enables clients to access encrypted dBASE tables.
cbFIELDRECALC	Field(s) recalculation
cbTRACE	Trace
cbDBLOGIN	Database login
cbDELAYEDUPD	Cached updates callback
cbNBROFCBS	Number of callbacks

{button ,AL(`types')} Other data type topics

cbGENPROGRESS

pCbBuf is assumed to be of the type cbPROGRESSDesc. This callback is issued by BDE to inform applications about the progress made during large batch operations, such as DbiBatchMove. The Generic Progress Report callback allows the client to obtain progress reports during an operation, and to cancel the operation, if desired. The client registers a progress callback function using cbGENPROGRESS as the value for *ecbType*. The body of the progress callback function (written by the client) should cast the callback buffer as a structure of type cbPROGRESSDesc.

The BDE returns either a percentage done (returned in the *iPercentDone* parameter of the cbPROGRESSDesc structure), or a message string to display on the status bar. The client should assume the following: if the *iPercentDone* value is negative, then the message string is valid; otherwise, the iPercentDone value should be considered. The message string format should always be *<Text String><:><Value>* to allow easy international translations. For example,

```
Records copied: 250
```

In the message string, the value and colon fields are optional. Possible return values are: cbrABORT (stop processing), or cbrCONTINUE (continue processing).

cbRESTRUCTURE

pCbBuf is assumed to be of the type RESTCbDesc. This callback may be issued several times during a call to DbiDoRestructure. Each time it is issued, BDE supplies information about an impending action and requests a response from the caller. The *iErrCode* in the CBRESTCbDesc structure is used to inform the caller about the different actions. Other fields of CBRESTCbDesc describes, if applicable, the object (for example, field, index, or validity check) to which this callback refers. Any callback may return with a cbrABORT that aborts the restructure. The batch result callback would be issued in the following different situations:

- When *iErrCode* == DBIERR_OBJMAYBETRUNCATED, a YES response forces data trimming. A NO response forces record that would be trimmed to a problems table.
- When *iErrCode* ==DBIERR_TABLELEVELCHANGED, a YES response allows the table level to change. A NO response aborts the restructure operation.
- When *iErrCode* == DBIERR_VALIDATEDATA, a YES force validity checks to be applied to existing data. A NO response applies validity checks to new data only.
- When *iErrCode* == DBIERR_OBJIMPLICITLYMODIFIED, this is a warning that an object was implicitly modified. For example, when a field that is part of a composite secondary index restructure is dropped, that field is implicitly dropped from the index.
- When *iErrCode* == DBIERR_OBJIMPLICITLYDROPPED, this is a warning that an object was dropped.
- When *iErrCode* == DBIERR_VALFIELDMODIFIED, this is a warning that the type or size of a field containing a validity check was modified.
- When *iErrCode* == DBIERR_VCHKMAYNOTBEENFORCED, this is a warning that because of referential integrity constraints on fields in the master table, new validity checks on these fields cannot be enforced on existing data.

cbBATCHRESULT
pCbBuf is assumed to be of the type RESTCbDesc. See (CBRESTCbDesc) This callback may be issued several times during a call to DbiBatchMove.

cbTABLECHANGED

pCbBuf is not used for this callback. The Table Changed callback is used to inform applications about changes to the table associated with a cursor. This callback is supported only by the Paradox driver.

cbCANCELQRY

Allows the user to cancel a long running Sybase query. The installed callback function is called periodically, and the user can return cbrABORT to cancel.

Any other return code will have no affect. No other BDE API calls are allowed from within the callback function.

cbINPUTREQ

The cbINPUTREQ callback is used when a BDE driver needs to communicate with the end user. This callback is used in the following cases:

a) a dBASE BLOB (.MDX) file is missing: cbiMDXMISSING
b) a Paradox BLOB (.MB) file is missing: cbiPDXBLOB
c) a Paradox lookup table is missing: cbiPDXLOOKUP
d) a dBASE (.DBT) file is missing: cbiDBTMISSING

The structure passed to the callback function is defined as follows:

```
typedef struct {
   CBInputId   eCbInputId;             // Id for this input request
   INT16       iCount;                 // Number of entries
   INT16       iSelection;             // Selection 1..n  (In/Out)
   BOOL16      bSave;                  // Save this option  (In/Out)
   DBIMSG      szMsg;                  // Message to display
   CBEntry     acbEntry[MAXCBENTRIES]; // Entries
 } CBInputDesc;
```

Structure	Type	Description
eCbInputId	CBInputId	*eCbInputId* is an enumerated type indicating what this input request is for. This will match one of the aforementioned values (cbiMDXMISSING,...).
iCount	INT16	*iCount* refers to the number of entries in the array *acbEntry.* (See below.)
iSelection	INT16	*iSelection* is used as both input to the callback function and output back to the driver. The input value from the driver indicates what the default choice in *acbEntry* should be. The output value is used to tell the driver which choice was selected.
bSave	BOOL16	The *bSave* element is used to tell the driver if it encounters a similar error on a different relation to take the same action as this time.
szMsg	DBIMSG	*szMsg* is a string the client can display to indicate what the problem is.
acbEntry	CBEntry	This array contains a list of operations that the driver can take to remedy the problem (such as Open the base table as read-only Abort the operation). The array also contains a help string for each of the choices. The array *acbEntry* is defined as:

```
typedef struct {                    // Entries for input requested
callback
      DBINAME        szKeyWord;     // Keyword to display
      DBIMSG         szHelp;        // Help String
   } CBEntry;
```

Where *szKeyWord* is a string indicating an operation that the driver can perform for this case. The *szHelp* element contains a help string associated with the operation that the client can display.

cbDBASELOGIN

Use the callback cbDBASELOGIN to enable clients to access encrypted dBASE tables.

The cbDBASELOGIN structure contains the following fields:

Structure	Type	Description
szUserName	DBINAME	Login name of user
szGroupName	DBINAME	Group to log in to
szUserPassword	DBINAME	User password

In some cases, no login may be performed. This may occur when either:

a) the optional login security has been turned off in dBASE; or

b) another client is using secured dBASE tables.

When no login has been performed in dBASE, you can call DbiOpenTable to attempt to open an encrypted table or you can call DbiCreateTable to create and encrypt a table (with Security enabled.)

In either case, when no login has been performed, the driver issues a cbDBASELOGIN callback. The client then displays a login screen with group name, user name, and password. The data from this screen is returned to the driver, which verifies it and sets the group name and user name in the session level properties. If the information is invalid (such as an invalid password, or the GroupName and UserName does not exist),then an error is returned, and the table is not opened/created.

The structure passed to the callback function is defined as follows:

```
// dBASE login callback structure
typedef struct
   {
      DBINAME  szUserName;        // Login name of user
      DBINAME  szGroupName;       // Group to log in to
      DBINAME  szUserPassword;    // User password
   } CBLoginDesc;

 typedef  CBLoginDesc far * pCBLoginDesc;
```

cbFIELDRECALC
Used for recalculation of fields.

cbTRACE

The cbTRACE is a system-level callback that can be used to retrieve trace information. The trace string retrieved through the callback is the same as that which goes to the debug window via OutputDebugString.

This structure is used to return trace info to the callback:

```
typedef struct       // trace callback info
   {
      TRACECat    eTraceCat;     // trace category
      UINT16      uTotalMsgLen;  // total message length
      CHAR        pszTrace[];    // trace string
                                 // (recommended size = DBIMAXTRACELEN
  (8192))
} TRACEDesc;

typedef enum         // trace categories
   {
      traceUNKNOWN   = 0x0000,
      traceQPREPARE  = 0x0001,   // prepared query statements
      traceQEXECUTE  = 0x0002,   // executed query statements
      traceERROR     = 0x0004,   // vendor errors
      traceSTMT      = 0x0008,   // statement ops (i.e. allocate, free)
      traceCONNECT   = 0x0010,   // connect / disconnect
      traceTRANSACT  = 0x0020,   // transaction
      traceBLOB      = 0x0040,   // blob i/o
      traceMISC      = 0x0080,   // misc.
      traceVENDOR    = 0x0100,   // vendor calls
   } TRACECat;
```

The TRACECat enums have the same bit sequence used to set the TRACE MODE configuration option, and can also be used (singularly or |'d together) as input to the dbTRACEMODE database property. The uTotalMsgLen field of the TRACEDesc struct can be used to determine whether the returned string (in pszTrace) has been truncated.

EXAMPLE of registering the cbTRACE callback:

```
      DbiRegisterCallBack
              (NULL,
              cbTRACE,
              iClientData,
              sizeof (TRACEDesc) + DBIMAXTRACELEN,
              (pVOID)pTraceInfo,          // ptr to client-allocated
  TRACEDesc
              (pfDBICallBack) lpfnTrace);
```

cbDBLOGIN
Database login.

cbDELAYEDUPD

This CallBack mechanism is invoked when the cached updates feature fails to write a modified record to the database. Because updates are not sent to the underlying table until the commit time, no errors (such as integrity constraint violation, and so on) are detected before the commit operation. If an error occurs at the commit time, you are prompted with an error message indicating what sort of error has happened. The clients should register a CallBack function for delayed updates by using the DbiRegisterCallBack function (ecbType for this CallBack is cbDELAYEDUPD) to be notified of the errors during the commit.

Here is the CallBack descriptor, cbDELAYEDUPD, for delayed updates:

```
// type of delayed update object (delayed updates CallBack)
typedef enum
{
      delayupdNONE        = 0,
      delayupdMODIFY      = 1,
      delayupdINSERT      = 2,
      delayupdDELETE      = 3
} DelayUpdErrOpType;

// delayed updates CallBack descriptor.
typedef struct
{
      DBIResult           iErrCode;
      DelayUpdErrOpType   eDelayUpdErrOpType;
      // Record size (physical record)
      UINT16              iRecBufSize;
      pBYTE               pNewRecBuf;
      pBYTE               pOldRecBuf;
} DELAYUPDCbDesc;
```

In the above CallBack descriptor, *eDelayUpdErrOpType* indicates the operation type , such as insert, delete or modify and *iErrCode* indicates what sort of error has occurred during the eDelayUpdErrOpType operation.

Clients should allocate enough memory for *pNewRecBuf* and *pOldRecBuf*. Each record buffer should be at least the delayed update cursor's physical record buffer size. The new (after the update) and old (before the update) record buffers are returned to the clients through *pNewRecBuf* and *pOldRecBuf* record buffers.

Clients can respond to this CallBack function with cbrABORT, cbrSKIP, cbrCONTINUE and cbrRETRY return codes. The following actions are taken depending on the return codes.

- If the return code is cbrABORT, the entire commit operation is aborted. Rollback of the committed updates will occur depending on the delayed updates cursor's property.
- If the return code is cbrSKIP or cbrCONTINUE, the failed update operation is discarded and the commit process continues with the remaining updates.
- If the return code is cbrRETRY, the failed update operation is tried again.

If no CallBack function is registered, the default return code is cbrABORT.

cbNBROFCBS
Indicates the maximum number of callback types.

Paradox, dBASE, and FoxPro physical types

These two lists show physical types supported by Paradox, dBASE (including FoxPro), respectively:

Paradox physical types	dBASE and FoxPro physical types
fldPDXCHAR	fldDBCHAR
fldPDXNUM	fldDBNUM
fldPDXMONEY	fldDBMEMO
fldPDXDATE	fldDBBOOL
fldPDXSHORT	fldDBDATE
fldPDXMEMO	fldDBFLOAT
fldPDXBINARYBLOB	fldDBLOCK (dBASE only)
fldPDXFMTMEMO	fldDBBINARY (dBASE only)
fldPDXOLEBLOB	fldDBOLEBLOB
fldPDXGRAPHIC	fldDBBYTES
fldPDXBLOB	fldDBLONG (dBASE 7.0 table format only)
fldPDXLONG	fldDBDATETIME (dBASE 7.0 table format only)
fldPDXTIME	fldDBDOUBLE (dBASE 7.0 table format only)
fldPDXDATETIME	fldDBAUTINC (dBASE 7.0 table format only)
fldPDXBOOL	
fldPDXAUTOINC	
fldPDXBYTES	
fldPDXBCD	

{button ,AL(`types')} Other data type topics

Data type translations

When a table is copied or appended to a table of a different driver type, data type translations take place according to the following tables. (You can widen this Help window to display the full width of the chart.)

Note: FoxPro uses the same data type translations as dBASE. MS SQL uses the same data type translations as Sybase.

From Paradox	To dBASE	To Oracle	To Sybase	To InterBase	To Informix
Alpha	VarChar	VarChar	Varying	VarChar	Character
Number	Number	Float	Double	Float	Float {20.4}
Money	Number	Money	Double	Money {16.2}	Number {20.4}
Date	DateTime	Date	Date		Date Time
Short	Number	SmallInt	Short	SmallInt	Number {6.0}
Memo	Long	Text	Blob/1	Text	Memo
Binary	LongRaw	Image	Blob	Byte	Memo
Formatted memo	LongRaw	Image	Blob	Byte	Memo
OLE	Image	Blob	Byte		OLE LongRaw
Graphic	LongRaw	Image	Blob	Byte	Binary
Long	Int	Long	Integer		Long Number
Time	Character {>8}	Character {>8}	Character {>8}	Character {>8}	Character {>8}
DateTime	Date	DateTime	Date	DateTime	Date Time
Bool	Character {1}	Bit	Character {1}	Character	Bool
AutoInc	Number	Int	Long	Integer	AutoInc
Bytes	LongRaw	Image	Varying	Byte	Memo
BCD	N/A	N/A	N/A		N/A

From Access	To Paradox	To dBASE	To InterBase
Autoincrement	Autoincrement	Numeric	Long
Bit	Logical	Logical	VarChar
Byte	Number	Numeric	Long
Char	Alpha	Character	VarChar
DateTime	Timestamp	DateTime	Date
Double	Number	Double	Double
Float	Number	Double/Float	Double
Long	Long Numeric	Long	
LongBinary	Binary	Ole	N/A
LongText	Memo	Memo	Text Blob
Money	Money	Numeric	Double
Short	Short	Numeric	Short
VarChar	Alpha	Character	VarChar

From Access	To Oracle	To Sybase	To Informix
Autoincrement	Number	Int	Integer
Bit	VarChar2	Bit	VarChar

Byte	Number	Int	Integer
Char	VarChar2	Char	VarChar
DateTime	Date DateTime	DateTime	DateTime
Double	N/A	Float	Float
Float	Number	Float	Float
Long	Number	Int	Integer
LongBinary	Number	Image	Byte
LongText	Long Text	Text	
Money	Number	Money	Money
Short	Number	SmallInt	SmallInt
VarChar	VarChar2	Char	VarChar

From dBASE	To Paradox	To Oracle	To Sybase	To InterBase	To Informix
Character	Alpha	Character	VarChar	Varying	VarChar
Number iUnits2=0 && iUnits1<5	Short	Number	SmallInt	Short	SmallInt
others	Number	Number	Float	Double	Float
Float	Number	Number	Float	Double	Float
Date	Date	DateTime	Date	Date	
Memo	Memo	Long	Text	Blob/1	Text
Bool	Bool Character {1}		Bit	Character {1}	Character
Lock	Alpha {24}	Character {24}	Character {24}	Character {24}	Character
OLE	OLE LongRaw	Image	Blob	Byte	
Binary	Binary	LongRaw	Image	Blob	Byte
Bytes	Bytes		LongRaw	Image (temp tables only)	Blob Byte
Long	Long	Number	Int		Integer
DateTime	DateTime	DateTime	DateTime	Date	DateTime
Double	Number	N/A	Float	Double	Float
AutoInc	AutoInc	Number	Int	Long	Integer

From Oracle	To Paradox	To dBASE	To Sybase	To InterBase	To Informix
Character	Alpha	Character	VarChar	Varying	Character
Raw	Number	Float {20.4}	Float	Double	Float
Date	DateTime	Date	DateTime	Date	DateTime
Number	Number	Double	Float	Double	Float
Long	Memo	Memo	Text	Blob/1	Text
LongRaw	Binary	Memo	Image	Varying	Byte

From Sybase	To Paradox	To dBASE	To Oracle	To InterBase	To Informix
Character	Alpha	Character	Character	Varying	Character
Var Character	Alpha	Character	Character	Varying	Character
Int	Number	Number {11.0}	Number	Long	Integer
Small Int	Short	Number {6.0}	Number	Short	SmallInt
Tiny Int	Short	Number {6.0}	Number	Short	SmallInt
Float	Number	Double	Number	Double	Float

Money	Money	Number {20.4}	Number	Double	Money {16.2}
Text	Memo	Memo	Long	Blob/1	Text
Binary	Binary	Memo	Raw	Varying	VarChar
Var Binary	Binary	Memo	Raw	Varying	VarChar
Image	Binary	Memo	LongRaw	Blob	Byte
Bit	Alpha	Bool	Character	Varying	Character
DateTime	DateTime	DateTime	Date	Date	DateTime
TimeStamp	Binary	Memo	Raw	Varying	VarChar
Float4	Number	Double	Number	Double	Float
Money4	Money	Number {20.4}	Number	Double	Money {16.2}
DateTime4	DateTime	DateTime	Date	Date	DateTime

From InterBase	To Paradox	To dBASE	To Oracle	To Sybase	To Informix
Short	Short	Number {6.0}	Number	Small Int	SmallInt
Long	Number	Number {11.0}	Number	Int	Integer
Float	Number	Float {20.4}	Number	Float	Float
Double	Number	Float {20.4}	Number	Float	Float
Char	Alpha	Character	Character	VarChar	Character
Varying	Alpha	Character	Character	VarChar	Character
wDate	DateTime	Date	Date	DateTime	DateTime
Blob	Binary	Memo	LongRaw	Image	Byte
Blob/1	Memo	Memo	Long	Text	Text

From Informix	To Paradox	To dBASE	To Oracle	To Sybase	To InterBase
Char	Alpha	Character	Character	VarChar	Varying
Smallint	Short	Number {6.0}	Number	Small Int	Short
Integer	Number	Number {11.0}	Number	Int	Long
Smallfloat	Number	Float {20.4}	Number	Float	Double
Float	Number	Float {20.4}	Number	Float	Double
Money	Money	Number {20.4}	Number	Float	Double
Decimal	Number	Float	Number	Float	Double
Date	Date Date	Date	DateTime	Date	
Datetime	DateTime	Date	Date	DateTime	Date
Interval	Alpha	Character	Character	VarChar	Varying
Serial	Number	Number {11.0}	Number	Int	Long
Byte	Binary	Memo	LongRaw	Image	Blob
Text	Memo	Memo	Long	Text	Blob/1
VarChar	Alpha	Character	Character	VarChar	Varying

{button ,AL(`types')} Other data type topics

Logical types and driver-specific physical types

The following tables show physical types translated into logical types, and then into the physical type of a different driver. (You might need to widen this Help window to display the full width of the chart.)

Note: FoxPro uses the same physical types as dBASE.

From Paradox physical type	To BDE logical type	To dBASE physical type
fldPDXBINARYBLOB	fldBLOB/fldstBINARY	fldDBMEMO
fldPDXBLOB	fldPDXMEMO	fldDBMEMO
fldPDXCHAR	fldZSTRING	fldDBCHAR
fldPDXDATE	fldDATE	fldDATE
fldPDXFMTMEMO	fldBLOB/fldstFMTMEMO	fldDBMEMO
fldPDXGRAPHIC	fldBLOB/fldstGRAPHIC	fldDBBINARY
fldPDXMEMO	fldBLOB/fldstMEMO	fldDBMEMO
fldPDXMONEY	fldFLOAT/fldstMONEY	if dBASE table Level < 7 fldDBFLOAT {20.4} else fldDBDOUBLE
fldPDXNUM	fldFLOAT	if dBASE table Level < 7 fldDBFLOAT {20.4} else fldDBDOUBLE
fldPDXOLEBLOB	fldBLOB/fldstOLEOBJ	fldDBOLEBLOB
fldPDXSHORT	fldINT16	fldDBNUM {6.0}
Paradox level 5 data types:		
fldPDXAUTOINC	fldINT32/fldstAUTOINC	fldDBAUTOINC
fldPDXBCD	fldBCD	fldDBCHAR
fldPDXBOOL	fldBOOL	fldDBBOOL
fldPDXBYTES	fldBYTES	fldDBMEMO
fldPDXDATETIME	fldTIMESTAMP	fldDBDATETIME
fldPDXLONG	fldINT32	fldDBLONG
fldPDXTIME	fldTIME	fldDBCHAR {>8}

From dBASE physical type	To BDE logical type	To Paradox physical type
fldDBBINARY	fldBLOB/fldstTYPEDBINARY	fldPDXBINARYBLOB
fldDBLOCK	fldLOCKINFO	fldPDXCHAR {24}
fldDBBOOL	fldBOOL	fldPDXBOOL
fldDBBYTES	fldBYTES	fldPDXBYTES (only for temp tables)
fldDBCHAR	fldZSTRING	fldPDXCHAR
fldDBDATE	fldDATE	fldPDXDATE
fldDBFLOAT	fldFLOAT	fldPDXNUM
fldDBMEMO	fldBLOB/fldstMEMO	fldPDXMEMO

fldDBNUM	if (iUnits2=0 && iUnits1<5) fldINT16	fldPDXSHORT	
	else fldFLOAT	fldPDXNUM	
fldDBOLEBLOB	fldBLOB/fldstDBSOLEOBJ	fldPDXOLEBLOB	

From Access physical type	To BDE logical type	To Paradox physical type	To dBASE physical type
fldACCAUTOINC	fldINT32/fldstAUTOINC	fldPDXLONG	fldDBAUTOINC
fldACCBINARY *	fldBYTES	fldPDXBYTES	fldDBBYTES
fldACCBIT	fldBOOL	fldPDXBOOL	fldDBBOOL
fldACCBYTE	fldINT16	fldPDXSHORT	fldDBNUM {6.0}
fldACCCHAR	fldZSTRING	fldPDXCHAR	fldDBCHAR
fldACCDATETIME	fldTIMESTAMP	fldPDXDATETIME	fldDBDATETIME
fldACCDOUBLE	fldFLOAT	fldPDXNUM	if dBASE table Level < 7 fldDBFLOAT {20.4} else fldDBDOUBLE
fldACCFLOAT	fldFLOAT	fldPDXNUM	if dBASE table Level < 7 fldDBFLOAT {20.4} else fldDBDOUBLE
fldACCLONG	fldINT32	fldPDXLONG	fldDBLONG
fldACCLONGBINARY	fldBLOB/fldstACCOLEOBJ	fldPDXBINARYBLOB	fldDBMEMO
fldACCLONGTEXT	fldBLOB/fldstMEMO	fldPDXMEMO	fldDBMEMO
fldACCMONEY	fldFLOAT/fldstMONEY	fldPDXMONEY	if dBASE table Level < 7 fldDBFLOAT {20.4} else fldDBDOUBLE
fldACCSHORT	fldINT16	fldPDXSHORT	fldDBNUM {6.0}
fldACCVARCHAR	fldZSTRING	fldPDXCHAR	fldDBCHAR

Note: fldACCBINARY can't be created, but can be read and written to.

From Oracle physical type	To BDE logical type	To Paradox physical type	To dBASE physical type
fldORACHAR	fldZSTRING	fldPDXCHAR	fldDBCHAR
fldORARAW	fldVARBYTES	fldPDXBINARYBLOB	fldDBMEMO
fldORADATE	fldTIMESTAMP	fldPDXDATETIME	fldDBDATETIME
fldORANUMBER	fldFLOAT	fldPDXNUM	if dBASE table Level < 7 fldDBFLOAT {20.4} else

			fldDBDOUBLE
fldORALONG	fldBLOB/fldstMEMO	fldPDXMEMO	fldDBMEMO
fldORALONGRAW	fldBLOB/fldstBINARY	fldPDXBINARYBLOB	fldDBMEMO
fldORAVARCHAR	fldZSTRING	fldPDXCHAR	fldDBCHAR
fldORAVARCHAR2			
iUnits1 <=255	fldSTRING	fldPDXCHAR	fldDBCHAR
iUnits1 >255	fldBLOB/fldstMEMO	fldPDXMEMO	fldDBMEMO
fldORAFLOAT	fldFLOAT	fldPDXNUM	if dBASE table Level < 7 fldDBFLOAT {20.4}
			else fldDBDOUBLE

From Sybase physical type	**To BDE logical type**	**To Paradox physical type**	**To dBASE physical type**
fldSYBBINARY	fldBYTES	fldPDXBYTES	fldDBMEMO
fldSYBBIT	fldBOOL	fldPDXBOOL	fldDBBOOL
fldSYBCHAR	fldZSTRING	fldPDXCHAR	fldDBCHAR
fldSYBDATETIME	fldTIMESTAMP	fldPDXDATETIME	fldDBDATETIME
fldSYBDATETIME4	fldTIMESTAMP	fldPDXDATETIME	fldDBDATETIME
fldSYBFLOAT	fldFLOAT	fldPDXNUM	if dBASE table Level < 7 fldDBFLOAT {20.4}
			else fldDBDOUBLE
fldSYBFLOAT4	fldFLOAT	fldPDXNUM	if dBASE table Level < 7 fldDBFLOAT {20.4}
			else fldDBDOUBLE
fldSYBIMAGE	fldBLOB/fldstBINARY	fldPDXBINARYBLOB	fldDBMEMO
fldSYBINT	fldINT32	fldPDXLONG	fldDBLONG
fldSYBMONEY	fldFLOAT/fldstMONEY	fldPDXMONEY	if dBASE table Level < 7 fldDBFLOAT {20.4}
			else fldDBDOUBLE
fldSYBMONEY4	fldFLOAT/fldstMONEY	fldPDXMONEY	if dBASE table Level < 7 fldDBFLOAT {20.4}
			else fldDBDOUBLE
fldSYBSMALLINT	fldINT16	fldPDXSHORT	fldDBNUM {6.0}
fldSYBTEXT	fldBLOB/fldstMEMO	fldPDXMEMO	fldDBMEMO

fldSYBTIMESTAMP	fldVARBYTES	fldPDXBINARYBLOB	fldDBMEMO
fldSYBTINYINT	fldINT16	fldPDXSHORT	fldDBNUM {6.0}
fldSYBVARBINARY	fldVARBYTES	fldPDXBINARYBLOB	fldDBMEMO
fldSYBVARCHAR	fldZSTRING	fldPDXCHAR	fldDBCHAR

If you are using Sybase System 10, the following additional Sybase physical types are available:

From Sybase physical type	To BDE logical type	To Paradox physical type	To dBASE physical type
fldSYBDECIMAL	fldFLOAT	fldPDXNUM	fldDBFLOAT(20,4)
fldSYBNUMERIC	fldFLOAT	fldPDXNUM	fldDBFLOAT(20,4)

From MS SQL physical type	To BDE logical type	To Paradox physical type	To dBASE physical type
fldMSSBINARY	fldBYTES	fldPDXBYTES	fldDBMEMO
fldMSSBIT	fldBOOL	fldPDXBOOL	fldDBBOOL
fldMSSCHAR	fldZSTRING	fldPDXCHAR	fldDBCHAR
fldMSSDATETIME	fldTIMESTAMP	fldPDXDATETIME	fldDBDATETIME
fldMSSDATETIME4	fldTIMESTAMP	fldPDXDATETIME	fldDBDBDATETIME
fldMSSDECIMAL	fldFLOAT	fldPDXNUM	fldDBFLOAT(20,4)
fldMSSFLOAT	fldFLOAT	fldPDXNUM	fldDBFLOAT {20.4}
fldMSSFLOAT4	fldFLOAT	fldPDXNUM	fldDBFLOAT {20.4}
fldMSSIMAGE	fldBLOB/fldstBINARY	fldPDXBINARYBLOB	fldDBMEMO
fldMSSINT	fldINT32	fldPDXLONG	fldLONG
fldMSSMONEY	fldFLOAT/fldstMONEY	fldPDXMONEY	if dBASE table Level < 7 fldDBFLOAT {20.4} else fldDBDOUBLE
fldMSSMONEY4	fldFLOAT/fldstMONEY	fldPDXMONEY	if dBASE table Level < 7 fldDBFLOAT {20.4} else fldDBDOUBLE
fldMSSNUMERIC	fldFLOAT	fldPDXNUM	if dBASE table Level < 7 fldDBFLOAT {20.4} else fldDBDOUBLE
fldMSSSMALLINT	fldINT16	fldPDXSHORT	fldDBNUM {6.0}
fldMSSTEXT	fldBLOB/fldstMEMO	fldPDXMEMO	fldDBMEMO
fldMSSTIMESTAMP	fldVARBYTES	fldPDXBINARYBLOB	fldDBMEMO
fldMSSTINYINT	fldINT16	fldPDXSHORT	fldDBNUM {6.0}
fldMSSVARBINARY	fldVARBYTES	fldPDXBINARYBLOB	fldDBMEMO

fldMSSVARCHAR	fldZSTRING	fldPDXCHAR	fldDBCHAR

From InterBase physical type	**To BDE logical type**	**To Paradox physical type**	**To dBASE physical type**
fldIBBLOB	fldBLOB	fldPDXBINARYBLOB	fldDBMEMO
fldIBBLOB/1	fldBLOB/fldstMEMO	fldPDXMEMO	fldDBMEMO
fldIBCHAR			
iUnits 1 <=255	fldZSTRING	fldPDXCHAR	fldDBCHAR
iUnits1 > 255	fldBLOB/fldstMEMO	fldPDXMEMO	fldDBMEMO
fldIBDATE	fldTIMESTAMP	fldPDXDATETIME	fldDBDATETIME
fldIBDOUBLE	fldFLOAT	fldPDXNUM	if dBASE table Level < 7 fldDBFLOAT {20.4} else fldDBDOUBLE
fldIBFLOAT	fldFLOAT	fldPDXNUM	if dBASE table Level < 7 fldDBFLOAT {20.4} else fldDBDOUBLE
fldIBLONG	fldINT32	fldPDXLONG	fldDBLONG
fldIBSHORT	fldINT16	fldPDXSHORT	fldDBNUM {6.0}
fldIBVARYING			
iUnits1 <= 255	fldSTRING	fldPDXCHAR	fldDBCHAR
iUnits1 >255	fldBLOB/fldstMEMO	fldPDXMEMO	fldDBMEMO

From Informix physical type	**To BDE logical type**	**To Paradox physical type**	**To dBASE physical type**
fldINFBYTE	fldBLOB/fldstBINARY	fldPDXBINARYBLOB	fldDBMEMO
fldINFCHAR			
iUnits1 <=255	fldZSTRING	fldPDXCHAR	fldDBCHAR
iUnits1 > 255	fldBLOB/fldstMEMO	fldPDXMEMO	fldDBMEMO
fldINFDATE	fldDATE	fldPDXDATE	fldDBDATE
fldINFDATETIME	fldTIMESTAMP	fldPDXDATETIME	fldDBDATETIME
fldINFDECIMAL	fldFLOAT	fldPDXNUM	if dBASE table Level < 7 fldDBFLOAT {20.4} else fldDBDOUBLE
fldINFFLOAT	fldFLOAT	fldPDXNUM	if dBASE table Level < 7 fldDBFLOAT {20.4} else fldDBDOUBLE
fldINFINTEGER	fldINT32	fldPDXLONG	fldDBLONG
fldINFINTERVAL	fldZSTRING	fldPDXCHAR	fldDBCHAR

fldINFMONEY	fldFLOAT/fldstMONEY	fldPDXMONEY	if dBASE table Level < 7 fldDBFLOAT {20.4} else fldDBDOUBLE
fldINFSERIAL	fldINT32	fldPDXLONG	fldDBLONG
fldINFSMALLFLOAT	fldFLOAT	fldPDXNUM	if dBASE table Level < 7 fldDBFLOAT {20.4} else fldDBDOUBLE
fldINFSMALLINT	fldINT16	fldPDXSHORT	fldDBNUM {6.0}
fldINFTEXT	fldBLOB/fldstMEMO	fldPDXMEMO	fldDBMEMO
fldINFVARCHAR	fldZSTRING	fldPDXCHAR	fldDBCHAR

{button ,AL(`types')} Other data type topics

International compatibility

The following sections describe considerations that may be encountered for international applications:

- Character Sets
- Sorting and Uppercasing Rules
- Language Drivers
- Date, Time, and Number Formats

Character sets

The shapes of characters that appear onscreen depend on an operating system's conventions for associating these shapes to internal binary values. Such conventions are called character sets, or code pages. The 8-bit code pages supported by BDE have 256 characters, numbered from 0 to 255 (using decimal values).

While most code pages use exactly the same numeric values (code points) for characters that are important in the United States, many of the symbols that are important to non-English-speaking countries map to different code points, depending on the particular code page. For example, the accented letter ‘á' maps to 160 on many DOS code pages, but in the Windows (ANSI) character set the same letter maps to code point 225. If an attempt is made to pass this character from an environment that uses the ANSI character set (used by most Windows programs) to a DOS environment, without translating the internal code point, the character appears under DOS as 'ß' (the German double-s) and may be misinterpreted in indexing, sorting, and so on. Character set identification and translation is therefore a very important issue if data loss is to be avoided internationally.

Characters whose code points are less than 128 are said to fall in the standard ASCII range; all the special international characters, located above code point 127, are known as extended characters.

BDE does not have a native character set. Usually, it operates with the binary values of characters. Strings should be passed to BDE in their default character set. The following table summarizes the default character sets for different character strings:.

Use	For
DOS code page	Local file names and pathnames, local user names and database aliases, names for table lookup and referential integrity, non-maintained index names
SQL server's character set	SQL data and metadata (table, field and index names, passwords and user names)
Table's character set	Table field names, data, validity checks, and secondary and maintained index names
ANSI	All SQL scripts (for local or SQL tables)

For QBE scripts, use the DOS character set for local table names and aliases. Use the ANSI character set for keywords and the table's character set for remaining characters in the script.

To translate character data between a table's native character set and Windows ANSI, use the functions DbiNativeToAnsi and DbiAnsiToNative. BDE returns error messages in the Windows ANSI character set.

Sorting and uppercasing rules

When character data is sorted in English-speaking countries, the sort sequence is usually based on the numeric values of the characters defined by the code page. This kind of sorting is known as binary collation. The approach is reasonable for English because most code pages define English letters in a neat, ascending numeric order.

However, binary sorting is not reasonable for other languages, because most code pages assign higher, fairly arbitrary values for their special characters (that is, the characters occur out of sequence with the standard ASCII characters among which they must be sorted). For similar reasons, uppercasing can be based on binary values for English, but not for other languages. To provide support for country-, code page-, and language-specific sorting and uppercasing rules, BDE uses information stored in language drivers.

Language drivers

A language driver (LD) specifies a particular primary (or native) character set, as well as a country/language-dependent set of rules for character manipulation, such as sorting, upper- and lowercasing, and the set of characters that are considered alphabetic. A language driver's primary character set is the character set in which its rules are defined. It specifies sorting and uppercasing in terms of the code points used by that particular code page. It also defines the character translation mapping between its primary character set and the ANSI code page, when necessary. (For a complete list of available language drivers and their primary character sets, use DbiOpenLdList.)

Long name	Short name	Character set	Collation sequence
'ascii' ANSI	DBWINUS0	1252 (ANSI)	Binary
'Spanish' ANSI	DBWINES0	1252 (ANSI)	Spanish
'WEurope' ANSI	DBWINWE0	1252 (ANSI)	Multilingual Western Europe
Access General	ACCGEN	1252 (ANSI)	Access Western Europe
Access Nord/Danish	ACCNRDAN	1252 (ANSI)	Access Norwegian/Danish
Access Swed/Finnish	ACCSWFIN	1252 (ANSI)	Access Swedish/Finnish
Access Japanese	ACCJAPAN	DOS 932	Access Japanese
Borland DAN Latin-1	BLLT1DA0	1252 (ANSI)	Danish
Borland DEU Latin-1	BLLT1DE0	1252 (ANSI)	German
Borland ENG Latin-1	BLLT1UK0	1252 (ANSI)	English/UK
Borland ENU Latin-1	BLLT1US0	1252 (ANSI)	Binary
Borland ESP Latin-1	BLLT1ES0	1252 (ANSI)	Spanish
Borland FIN Latin-1	BLLT1FI0	1252 (ANSI)	Finnish
Borland FRA Latin-1	BLLT1FR0	1252 (ANSI)	French
Borland FRC Latin-1	BLLT1CA0	1252 (ANSI)	French Canadian
Borland ISL Latin-1	BLLT1IS0	1252 (ANSI)	Isalandic
Borland ITA Latin-1	BLLT1IT0	1252 (ANSI)	Italian
Borland NLD Latin-1	BLLT1NL0	1252 (ANSI)	Dutch
Borland NOR Latin-1	BLLT1NO0	1252 (ANSI)	Norwegian
Borland PTG Latin-1	BLLT1PT0	1252 (ANSI)	Portogese
Borland SVE Latin-1	BLLT1SV0	1252 (ANSI)	Swedish
dBASE CHS cp936	DB936CN0	DOS CODE PAGE 936	dBASE China
dBASE CHT cp950	DB950TW0	DOS CODE PAGE 950	dBASE Taiwan
dBASE CSY cp852	DB852CZ0	DOS CODE PAGE 852	dBASE Czech852
dBASE CSY cp867	DB867CZ0	DOS CODE PAGE 867	dBASE Czech867
dBASE DAN cp865	DB865DA0	DOS CODE PAGE 865	dBASE Danish
dBASE DEU cp437	DB437DE0	DOS CODE PAGE 437	dBASE German
dBASE DEU cp850	DB850DE0	DOS CODE PAGE 850	dBASE German850
dBASE ELL GR437	DB437GR0	DOS CODE PAGE 737	dBASE Greek
dBASE ENG cp437	DB437UK0	DOS CODE PAGE 437	dBASE English/UK
dBASE ENG cp850	DB850UK0	DOS CODE PAGE 850	dBASE English850/UK

dBASE ENU cp437	DB437US0	DOS CODE PAGE 437	dBASE English/US
dBASE ENU cp850	DB850US0	DOS CODE PAGE 850	dBASE English/US
dBASE ESP cp437	DB437ES1	DOS CODE PAGE 437	dBASE Spanish
dBASE ESP cp850	DB850ES0	DOS CODE PAGE 850	dBASE Spanish850
dBASE FIN cp437	DB437FI0	DOS CODE PAGE 437	dBASE Finnish
dBASE FRA cp437	DB437FR0	DOS CODE PAGE 437	dBASE French
dBASE FRA cp850	DB850FR0	DOS CODE PAGE 850	dBASE French850
dBASE FRC cp863	DB863CF1	DOS CODE PAGE 863	dBASE Canadian-French863
dBASE HUN cp852	DB852HDC	DOS CODE PAGE 852	dBASE Hungarian
dBASE ITA cp437	DB437IT0	DOS CODE PAGE 437	dBASE Italian
dBASE ITA cp850	DB850IT1	DOS CODE PAGE 850	dBASE Italian850
dBASE JPN cp932	DB932JP0	DOS CODE PAGE 932	dBASE Japan932
dBASE JPN Dic932	DB932JP1	DOS CODE PAGE 932	dBASE JapanDic932
dBASE KOR cp949	DB949KO0	DOS CODE PAGE 949	dBASE Korea
dBASE NLD cp437	DB437NL0	DOS CODE PAGE 437	dBASE Dutch
dBASE NLD cp850	DB850NL0	DOS CODE PAGE 850	dBASE Dutch850
dBASE NOR cp865	DB865NO0	DOS CODE PAGE 865	dBASE Norwegian
dBASE PLK cp852	DB852PO0	DOS CODE PAGE 852	dBASE Polish852
dBASE PTB cp850	DB850PT0	DOS CODE PAGE 850	dBASE Brazilian Portuguese 850
dBASE PTG cp860	DB860PT0	DOS CODE PAGE 860	dBASE Brazilian Portuguese 860
dBASE RUS cp866	DB866ru0	DOS CODE PAGE 866	dBASE Russian
dBASE SLO cp852	DB852SL0	DOS CODE PAGE 852	dBASE Slovac
dBASE SVE cp437	DB437SV0	DOS CODE PAGE 437	dBASE Swedish
dBASE SVE cp850	DB850SV1	DOS CODE PAGE 850	dBASE Swedish850
dBASE THA cp874	DB874TH0	DOS CODE PAGE 874	dBASE Thai
dBASE TRK cp857	DB857TR0	DOS CODE PAGE 857	dBASE Turkish
DB2 SQL ANSI	DB2ANDEU	1252 (ANSI)	Dictionary
FoxPro German 437	FOXDE437	DOS CODE PAGE 437	FoxPro German
FoxPro Nordic 437	FOXNO437	DOS CODE PAGE 437	FoxPro Nordic
FoxPro Nordic 850	FOXNO850	DOS CODE PAGE 850	FoxPro German
FoxPro German 1252	FOXDEWIN	1252 (ANSI)	FoxPro German
FoxPro Nordic	FOXNOWIN	1252 (ANSI)	
Hebrew dBASE	DBHEBREW		dBASE Hebrew
Oracle SQL WE850	ORAWE850	DOS CODE PAGE 850	Multilingual Western Europe
Paradox 'ascii'	ascii	DOS CODE PAGE 437	Binary
Paradox 'hebrew'	hebrew	DOS CODE PAGE 862	Hebrew
Paradox 'intl'	intl	DOS CODE PAGE 437	Multilingual Western Europe

Paradox 'intl' 850	intl850	DOS CODE PAGE 850	Brazilian Portuguese, French Canadian
Paradox 'nordan'	nordan	DOS CODE PAGE 865	Norwegian/Danish (Paradox 3.5)
Paradox 'nordan40'	nordan40	DOS CODE PAGE 865	Norwegian/Danish (Paradox 4.0, 5.0, 5.5, 7.0)
Paradox 'japan'	japan	DOS CODE PAGE 932	Japanese
Paradox 'swedfin'	swedfin	DOS CODE PAGE 437	Swedish/Finnish
Paradox 'turk'	turk		Turkish
Paradox ANSI HEBREW	ANHEBREW	1255(ANSI)	HebrewAnsi
Paradox China 936	china	DOS CODE PAGE 936	China
Paradox Cyrr 866	cyrr	DOS CODE PAGE 866	Cyrillic
Paradox Czech 852	czech	DOS CODE PAGE 852	Czech852
Paradox Czech 867	cskamen	DOS CODE PAGE 867	Czech867
Paradox ESP 437	SPANISH	DOS CODE PAGE 437	Spanish
Paradox Greek GR437	grcp437	DOS CODE PAGE 737	Greek
Paradox Hun 852 DC	hun852dc	DOS CODE PAGE 852	Hungarian
Paradox ISL 861	iceland	DOS CODE PAGE 861	Icelandic
Paradox Korea 949	korea	DOS CODE PAGE 949	Korea
Paradox Polish 852	polish	DOS CODE PAGE 852	Polish
Paradox Slovene 852	slovene	DOS CODE PAGE 852	Slovene
Paradox Taiwan 950	taiwan	DOS CODE PAGE 950	Taiwan
Paradox Thai 874	thai	DOS CODE PAGE 874	Thai
Pdox ANSI ISO L_2 CZ	ANIL2CZW	1250 (ANSI)	
Pdox ANSI Cyrillic	ancyrr	1251 (ANSI)	Compatible with Paradox "cyrr"
Pdox ANSI Czech	anczech	1250 (ANSI)	Compatible with Paradox "czech"
Pdox ANSI Greek	angreek1	1253 (ANSI)	Compatible with Paradox "greek"
Pdox ANSI Hun. DC	anhundc	1250 (ANSI)	Compatible with Paradox "hung"
Pdox ANSI Intl	ANSIINTL	1252 (ANSI)	Compatible with Paradox "intl"
Pdox ANSI Intl850	ANSII850	DOS CODE PAGE 850	Compatible with Paradox "intl850"
Pdox ANSI Nordan4	ANSINOR4	1252 (ANSI)	Compatible with Paradox "nordan40"
Pdox ANSI Polish	anpolish	1250(ANSI)	Compatible with Paradox "polish"
Pdox ANSI Slovene	ansislov	1250(ANSI)	Compatible with Paradox "slovene"
Pdox ANSI Spanish	ANSISPAN	1252(ANSI)	Compatible with Paradox

			"SPANISH"
Pdox ANSI Swedfin	ANSISWFN	1252(ANSI)	Compatible with Paradox "swedfin"
Pdox ANSI Turkish	ANTURK	1254(ANSI)	Compatible with Paradox "turk"
SQL Link ROMAN8	BLROM800	Roman-8	Binary
Sybase SQL Dic437	SYDC437	DOS CODE PAGE 437	Sybase 437 dict. with case-sensitivity
Sybase SQL Dic850	SYDC850	DOS CODE PAGE 850	Sybase 850 dict. with case-sensitivity
pdx Czech 852 'CH'	czechw	DOS CODE PAGE 852	
pdx Czech 867 'CH'	cskamenw	DOS CODE PAGE 867	
pdx ANSI Czech 'CH'	anczechw	1250 (ANSI)	
pdx ISO L_2 Czech	il2czw	ISO8859-2	

Default language driver settings are defined in the BDE configuration file (IDAPI.CFG). You can change these defaults using the BDE Administrator. If you can be certain that your application will not need to support character sets other than Windows ANSI, you can reduce the need for extra processing, such as character translation, by changing your language driver defaults to ANSI-based ones. Additionally, if your application will be working exclusively with data from a particular SQL server, it may be advantageous to reset local language driver defaults to the driver you have associated with the SQL database alias.

When a Paradox, dBASE, or FoxPro table is created, the default language driver's identification is stored in the table file header. The default language driver setting can be overridden at creation by specifying optional parameters to DbiCreateTable. The table's language driver will be used by BDE functions that manipulate character data, such as DbiSortTable, DbiAddIndex, and a variety of other functions such as DbiGetNextRecord, DbiGetPriorRecord, DbiSetRange, DbiSetToKey, DbiInsertRecord, and so on. A table's language driver can be changed after creation by using DbiDoRestructure. DbiDoRestructure does not translate table data or metadata to the character set of the new language driver, in cases where the character sets of the old and new language drivers differ. However, table data is transliterated between differing character sets by DbiBatchMove.

For SQL table driver types, such as Sybase or Oracle, language driver settings are defined with the database alias in the BDE configuration file (IDAPI.CFG). All of the above operations when applied to SQL tables are governed by this setting.

To obtain the name of a table's language driver or the name of the default LD for a specific table driver, use the function DbiGetLdName.

The following table summarizes the default settings for language drivers.

Language driver for	**Default Setting**
System	System language driver setting current in IDAPI.CFG.
Access driver	Access language driver setting current in IDAPI.CFG.
Paradox driver	Paradox language driver setting current in IDAPI.CFG.
dBASE driver	dBASE language driver setting current in IDAPI.CFG.

Text driver	System language driver.
SQL database	LANGDRIVER setting for this database current in IDAPI.CFG.
SQL drivers	LANGDRIVER setting in DB OPEN section of IDAPI.CFG for this driver.
Table cursor	Language driver associated with this table at the time it was created.
Database handle	Language driver of the database this handle represents.

Note: You can override all defaults by using DbiSetProp.

Date, time, and number formats

Default settings for date, time, and number formats are defined in the Registry. (See the Date, Time, and Number pages in the BDE Administrator.) These settings are used by BDE anywhere conversion must be performed between strings (such as "15/12/94") and internal representations of dates, times, and numbers (for example, when parsing a date found in a query string). For best results, the BDE default settings should be kept in synchronization with the Windows Control Panel. The default settings can be overridden at any time with DbiSetDateFormat, DbiSetTimeFormat, and DbiSetNumberFormat.

Credits

Sara Anderson
Richard Army
Gretel Bailey
Gareth Bowles
Laxman Chinnakotla
Cliff Cormier
Nick Derpich
Mike Destein
Susanne Edgerton
Anne Fletcher
Scott Frolich
Rajamohan Gandhasri
Barbara Gentry
Micael Gomez
Kurt Hansen
Brian Henry
Niel Henry
Sarah Huang
Jonathan Lin
John Keegan
Robin Kennedy
Jan Kraski
Klaus Krietsch
Marilyn Lem
Emeli Marcondes
William Morris
Michael Morrison
Rick Nadler
Chris Ohlsen
Don Phan
Kris Ramberg
Eric Roth
Aparna Srikanth
Ramesh Theivendran
Steve Todd
Devendra Vamathevan
Narayanan Vijaykumar
Ken Vodicka
Ginger Wilsbacher