Contents

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

Installation
Installation instructions, requirements, and how to contact Borland for technical support.

Basic Concepts
Concepts fundamental to the Borland Database Engine.

Configuration Management
Settings for the BDE configuration file and additional hints for configuring your system.

Application Development
An introduction to the basics of programming with BDE and a guide to the general procedures for developing applications.

Using the Function Reference
The elements and syntax conventions used in each function definition. Includes the definitions of variable names, constants, #defines, and typedefs, as used throughout this reference.

Function Reference, Categorical
Functions grouped according to the tasks they perform. Use this list to find and display definitions for all functions associated with a particular task.

Function Reference, Alphabetical
Functions listed in alphabetical order. Use this list to to quickly display a particular function's definition.

Data Structures
BDESDK constants, type definitions, data types, descriptor structures, and properties. Use this list to display a definition for each data structure.

International Compatibility
Considerations for international applications.

Installation

This section provides the prerequisites and procedures for installing the Borland Database Engine.

- Hardware and Software Requirements
- Installation Procedure
- BDE Software Components
- Software Registration and Technical Support

Borland Database Engine

The Borland® Database Engine (BDE) is a software package designed to give developers of Microsoft Windows applications access to multiple data sources with a consistent application program interface (API). This is exactly the same engine used by Delphi®, Paradox® for Windows, Visual dBASE® for Windows, and Quattro® Pro for Windows. The API for this database engine is called Borland Database Engine Software Development Kit (BDESDK). (Note: You might occasionally encounter references to the older name for BDESDK: the "Integrated Database Application Program Interface" or "IDAPI".)

Hardware and Software Requirements

This list describes the hardware and software requirements of the Borland Database Engine.

- DOS 3.1 or higher
- Windows version 3.1
- 6 MB RAM required, 8 MB RAM recommended
- 25 MB free disk space

Installation Procedure

The following sections describe the steps used to install the Borland Database Engine from diskettes or CD-ROM.

Installing from diskettes

Follow these steps to install the Borland Database Engine:

1. Insert Diskette 1 in a floppy drive. (The following instructions assume you are using drive A. Substitute your drive letter if necessary.)
2. Choose File|Run from the Program Manager. The Run dialog box appears.
3. Enter a:setup.exe in the Command Line text box.
4. Follow the instructions on the screen. Change the default installation directory to the directory you chose for the BDESDK section of the Borland Database Engine.

Installing from CD-ROM

Follow these steps to install the Borland Database Engine:

1. Insert the CD-ROM in your CD-ROM drive. (The following instructions assume you are using drive E. Substitute your drive letter if necessary.)
2. Choose File|Run from the Program Manager. The Run dialog box appears.
3. Enter e:setup.exe in the Command Line text box.
4. Follow the instructions on the screen. Change the default installation directory to the directory you chose for the BDESDK section of the Borland Database Engine.

BDE Software Components

This table describes Borland Database Engine software components.

Component	Description
BDE	Core .DLL files.
SnipIt Code Viewer	Allows you to display and run precompiled and linked code segments that demonstrate the use of BDESDK functions.
BDE Configuration Utility	Allows you to configure BDE.
Database Desktop	Allows you to view, create, and restructure tables and run queries with a graphic interface.
Query Manager	Allows you to create and run SQL and QBE queries.
DBPing	Allows you to connect to SQL databases.
Sample applications	Demonstrates the use of BDE functions.

Software Registration and Technical Support

The Borland Assist program offers a range of technical support plans to fit the different needs of individuals, consultants, large corporations, and developers. To receive help with this product, send in the registration card and select the Borland Assist plan that best suits your needs.

North American customers can register by phone 24 hours a day by calling 1-800-845-0147.

For additional details on these and other Borland services, see the Borland Assist Support and Services Guide included with this product.

Basic Concepts

This section introduces concepts basic to the Borland Database Engine (BDE), including a product description, architectural overview, a description of each of the BDE objects, and information about database entities, transactions, callbacks, and cross-database operations. Terms used throughout the BDE documentation are defined.

Features	A list of product capabilities and benefits
Core BDE Files	A list of the essential files that make up the Borland Database Engine.
Tools and Examples	A list of supplemental tools and files containing sample code
Components	A list of the major software components of the BDE package.
Architectural Overview	A visual high-level overview of the BDE components.
Shared Services	The infrastructure or internal organization of BDE.
BDESDK	Overview of BDESDK, the API for the Borland Database Engine.
BDE Objects	Overview of the various run-time objects you can create to manipulate database entities.
Database Entities	Overview of persistent objects common to most database systems.

Core Borland Database Engine Files

The core BDE 2.5 files include:

Core File	Description
IDAPI01.DLL	Primary BDE DLL
ILD01.DLL	International Language Driver support functions
IDBAT01.DLL	Contains the batch operations
IDQRY01.DLL	Local Query Engine
IDASCI01.DLL	ASCII Text driver
IDPDX01.DLL	Paradox Driver
IDDBAS01.DLL	dBASE driver
IDODBC01.DLL	ODBC Socket Driver (allows the use of any ODBC 2.0 driver)
IDR10009.DLL	Resource file for error messages
ODBC.NEW	ODBC 2.0 core driver (can rename to ODBC.DLL)
ODBCINST.NEW	ODBC 2.0 installation driver (can rename to ODBCINST.DLL)
BDECFG.EXE	BDE configuration utility
BDECFG.HLP	Help file for the configuration utility
IDAPI.CFG	File containing BDE configuration information
*.ld	International Sort order information

Tools and Examples

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

Tool	Description
Database Desktop	Simple user interface for viewing and creating tables
Configuration Utility	Main BDE tool for managing driver, alias, and system configuration in the BDE configuration file (IDAPI.CFG)
DLL Swap	Tool to switch between the debug ("debug layer") and non-debug versions of BDESDK. The Debug Layer is the version of BDESDK that can output trace information and perform extra parameter checking.
IDAPI.TOK	BDE syntax highlighting file for the BC 4.x IDE.
BDE.HLP	BDE Help File.
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.
Pascal Examples	Simple examples written in Pascal
INVENTORY	Simple inventory example, works on Paradox tables. Written in C.
EMPLOYEE	Simple personnel example, works on dBASE tables. Written in Pascal. Basically a port of Inventory.
ADDRESS	Simple AddressBook example. Works with any table type. Written in C.
SEEK	Simple example of using the BDE in a DLL Searches dBASE expression indexes. Paradox for Windows form provided for testing. DLL written in C.
RESTRUCT	Another example of the BDE in a DLL. Shows how to do a restructure from OPAL in Paradox for Windows. DLL written in C.
FILTER	Using filters with Delphi controls. Written in Object Pascal for use with Delphi.
TABLEENH	Using BDE to expand the Delphi environment. Creates a new component to add to the component library which adds additional functionality to a table-derived class.
TABLES	Sample tables

Database Desktop

(\BDE\DBD\DBD.EXE)

The 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 configuration utility: \IDAPI\BDECFG.EXE

The BDE configuration file: \IDAPI\IDAPI.CFG

The BDE Configuration utility is a visual tool for managing the information in the BDE configuration file (IDAPI.CFG). This information includes all driver and alias information, the location of the network control directory for PARADOX tables, the size of the Swap Buffer (Database Data cache), the amount of lower DOS memory to use, and various other system information. The configuration utility is currently the only way outside of Paradox for Windows to add an alias to the system.

You can modify system information and existing aliases by using the function DbiOpenCfgInfoList.

DLL Swap

\BDE\BIN\DLLSWAP.EXE

The DLL swap utility is used to swap between the debug and non-debug versions of the BDE. The debug layer is useful during application development. When enabled, the debug layer will do additional parameter checking for BDE functions, as well as providing trace information. In its normal state, the core BDE DLLs do not do much parameter validation. While this provides a speed improvement if the parameters are correct, it usually involves a GP fault if the parameters are incorrect.

The debug layer switches the DLL that is to be used. By default, the BDE ships the files IDAPI01.DLL and DBG.DLL. IDAPI01.DLL is the non-debug version of the BDE, while DBG.DLL is the debug version. The DLL swap utility will copy IDAPI01.DLL to NODBG.DLL, and then copy DBG.DLL to IDAPI01.DLL. Or, if DLLSwap has already been run, it will go the other way, restoring the non-debug version of the BDE.

Note that the BDESDK function DbiDebugLayerOptions must be called with the proper parameters to enable the debug layer. Calling this function with the non-debug version of the BDE will not result in any harm, so this code can be left in the application regardless of which DLL is being used.

IDAPI.TOK

(\BDE\DOC\IDAPI.TOK)

This file is used by the BC 4.x IDE to provide syntax highlighting for BDESDK functions and types. The IDAPITOK.TXT file in the same directory provides information on using this file.

BDE.HLP

(\BDE\DOC\BDE.HLP)

This WinHelp file contains the complete user's guide and BDESDK function reference. It requires the presence of WINHELP.EXE. Note that this file can be linked into the BC 4.x IDE by using the OpenHelp machanism (\BC45\BIN\OHELPCFG.EXE to configure).

DBPING

(\BDE\EXAMPLES\C\DBPING\DBPING.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

(\BDE\EXAMPLES\QUERY\QUERY.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 and QBE statements and see the results of the operation. Note that the output Window is limited to 64Kb of data and it is a multi-line edit control. However it does provide a method of testing SQL and QBE statements, so it can be used to determine if it is the query itself that is failing, or if the offending application is calling the BDE query functions incorrectly.

SNIPIT

(\BDE\EXAMPLES\SNIPIT\SNIPIT.EXE)

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

Pascal Examples

(EXAMPLES\PASCAL*.pas)

These examples were written with the same intention as the SNIPIT example: to provide simple examples showcasing as many aspects of BDE as possible. Note that the Pascal examples are ports from SNIPIT (C), and that not all examples have been ported.

INVENTORY

(\BDE\EXAMPLES\C\INVENTRY\INVENTRY.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.

EMPLOYEE

\BDE\EXAMPLES\PASCAL\EMPLOYEE\EMPLOYEE.EXE

This example is a standalone Pascal application for Windows using BDE (No Delphi Controls). It is basically a port of the Inventory example to Pascal, although it does use the dBASE table format.

ADDRESS

\BDE\EXAMPLES\C\ADDRESS\ADDRESS.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).

SEEK

\BDE\EXAMPLES\PDOXWIN\SEEK

Simple example of using the BDE in a DLL. Searches dBASE expression indexes. Paradox for Windows form provided for testing. DLL written in C.

RESTRUCT

\BDE\EXAMPLES\PDOXWIN\RESTRUCT

Another example of the BDE in a DLL. Shows how to do a restructure from OPAL in Paradox for Windows. DLL written in C.

FILTER

\BDE\EXAMPLES\DELPHI\FILTER

Using filters with Delphi controls. Written in Object Pascal for use with Delphi.

TABLEENH

\BDE\EXAMPLES\DELPHI\TABLEENH

Uses BDE to expand the Delphi environment. Creates a new component to add to the component library which adds additional functionality to a TTable derived class.

TABLES

\BDE\EXAMPLES\TABLES

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

Features

BDE offers these features:

- BDE provides application developers with BDESDK, a uniform and consistent API to access multiple database formats including dBASE, Paradox, Text, InterBase, Oracle, Sybase, Informix, 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 for upsizing to client/server applications.
- BDE gives applications direct and live access to data sources without the need for importing and exporting. This includes the ability to easily copy data from one format to another, as well as linking and querying data across formats.
- BDE is the highest performance database engine for Paradox and dBASE file formats.
- BDE provides a consistent query language (SQL or QBE) and navigational-based access mechanism to define and access data in both SQL-based servers and file-based databases.
- 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 drivers. 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.

Components

The BDE package consists of the following components:

- The Borland Database Engine Software Development Kit (BDESDK, a set of function calls.
- A common database query engine that supports both Structured Query Language (SQL) and Query by Example (QBE) languages.
- Three core database drivers (Paradox, dBASE, and Text).
- Optional native SQL drivers to Oracle, Sybase, Informix, and InterBase.
- ODBC connectivity that allows access to any data source for which an ODBC driver is available. (BDESDK applications get the benefits of BDESDK even when using an ODBC driver.)
- A collection of tools and sample programs to ease the task of application development.

Architectural Overview

BDE has a driver-based architecture. Each distinct database format or data source usually requires a separate BDESDK 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 some FoxPro™ (v. 2.5, 2.6) file formats.

BDE is object-oriented in design, making it easy to extend and customize. To extend BDE so that it can access an additional database system, simply install the appropriate BDESDK 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, since it provides transparent access to both server databases and local databases on PCs.

Shared Services

BDE is based upon the software components 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 architecture of BDE.

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 common 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 are not required 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 arbitrarily 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.

SQL Generator

The common query engine supports QBE as an alternate query language, which is more intuitive to end users tha 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 and dBASE 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 most 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 through the same BDESDK function calls used for accessing persistent tables.

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.

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 configuration file (IDAPI.CFG). It reads the configuration file at startup time to get the information it needs to customize the BDE environment.

The BDE function DbiOpenCfgInfoList gives the application access to the configuration file. The BDE configuration utility (BDECFG.EXE) enables the application developer to register drivers and aliases, set date format options, and customize BDE drivers.

Common Query Engine

A common query engine supports both SQL and QBE query languages. The query engine supports a subset of ANSI92 standard SQL on Paradox and dBASE tables. The common query engine also supports cross-database joins.

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. BDE is bundled with nearly ninety language drivers.

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.

BDESDK

The Borland Database Engine Software Development Kit (BDESDK) is the API for BDE. It consists of a set of functions that can be called from any programming language capable of calling Windows DLLs. BDESDK is optimized for calling from C/C.

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, Informix, 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

BDESDK unifies access to both PC-based or ISAM databases and server-based SQL databases with a consistent cursor-based API. BDESDK supports the basic APIs for both types of databases, extending powerful features of each type to the other. For example, BDESDK's navigational features are influenced by ISAM databases, and are extended to support server-based databases. Similarly, the Query portion of BDESDK 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 BDESDK unique. For example, Paradox for Windows and Visual dBASE for Windows 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 transaction processing for SQL databases. For this reason, BDESDK is not a least-common-denominator API.

Purposes

For all supported databases for which an BDESDK native driver or an ODBC driver is available, BDESDK 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

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 BDESDK 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

System

One system object controls the resources common to all applications running on the same machine. The BDESDK 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 BDE configuration file (IDAPI.CFG).

Clients

A new client object is created when an application calls the BDESDK initialization function. This first call to DbiInit is necessary before any other BDESDK call can be made. The client object is maintained automatically by BDESDK and exists mainly as a context for all the system resources used by BDESDK 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.

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 BDESDK run-time objects that can be created:

- Database Drivers
- Databases
- Cursors
- Query statements

Any object created in the context of one session might not be used in the context of another session. 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.

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 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.

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

dBASE, Paradox, and Text drivers

The drivers for Paradox, dBASE, and Text databases are shipped with BDE.

SQL Drivers

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

ODBC Drivers

Any ODBC driver can be used with BDE, because BDESDK has an ODBC connectivity socket. The rich features of BDESDK, 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.

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 supports the notion of a standard database to deal with file-based databases such as Paradox, dBASE, and Text. 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 path name, 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.

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 configuration utility BDECFG.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.

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.

Query Statements

A query 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.

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 opening a cursor.

Database Entities

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

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

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 BDESDK 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.

Indexes

An index determines the order of the records in a table. Paradox, dBASE, 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, 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 databases 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 databases mentioned, only dBASE uses subset indexes.
- Unique indexes cannot have duplicate key values.
- Indexes can be maintained or non-maintained.

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 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 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. 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 index.
- Paradox supports maintained and non-maintained secondary indexes.
- Paradox does not support expression indexes.
- Paradox does not support subset indexes.
- Paradox supports case-sensitive/insensitive secondary indexes.

Primary indexes

A Paradox primary key is defined as a field or group of fields when 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 field(s) 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 upper- and lowercase letters as it sorts fields. 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.

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

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 translates all data types as they are stored by the data source back to the generic logical data types. BDE's logical data types are compatible with standard C language data types.

When accessing a table, the application can override the translation mechanism, opting to receive data in the physical format used by the data source.

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.

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

- 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.

For both QBE and SQL executed on Paradox and dBASE tables, a query can be executed as a live result set, resulting in an updatable cursor on the original table.

SQL

BDE allows access to Paradox or dBASE data through a convenient subset of the SQL language.

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 do not result in updatable cursors.

Query by Example (QBE)

BDESDK 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 updatable.

Transactions

SQL database servers support transactions. A transaction is a group of related operations that must all be performed successfully or no change to the database takes place.

BDE supports transactions on SQL servers with three BDESDK function calls:

- DbiBeginTran
- DbiEndTran
- DbiGetTranInfo

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

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 the engine. 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 the engine can get the application's response without interrupting the normal client process flow.

To inspect the callback structures, see Data Structures

Cross-database Operations

BDESDK query and batch functions can operate on heterogenous 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 Online User 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.

Configuration Management

The Borland Database Engine Configuration Utility is a redistributable application that sets up and manages your application's configuration. The configuration parameters are stored in a binary file with the extension .CFG (for example, IDAPI.CFG) that the application reads at startup.

BDE Configuration Reference

The BDE Configuration Utility is displayed in notebook format, with tabbed pages that govern different configuration tasks. The following sections describe the menu commands and each task page in the BDE Configuration Utility.

- Managing Your BDE Configuration File

Overview and menu commands for managing the BDE configuration file (IDAPI.CFG).

- Drivers Page

Setting the default parameters for any database driver your Borland Database Engine application will access, including dBASE, Paradox. SQL, and ODBC.

- Aliases Page

Adding, deleting, or modifying database aliases.

- System Page

Setting your Borland Database Engine initialization parameters.

- Date, Time, and Number Pages

Setting the format for displaying date, time, and numeric data.

BDE Configuration Guide

The following sections explain how to modify the appropriate configuration files when configuring drivers.

- Configuring Microsoft Open Database Connectivity (ODBC)

Supplemental help with configuring ODBC drivers and datasources. AutoODBC enables the BDE to automatically read the names of drivers and data sources from the ODBC configuration files, odbc.ini and odbcinst.ini.

Note: Before using this utility to change your BDE configuration file, be sure to close any open BDE applications. Your changes take effect the next time you start your Borland Database Engine application.

Managing Your BDE Configuration File

The BDE Configuration Utility (BDECFG.EXE) and a generic configuration file (IDAPI.CFG) are 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 Configuration Utility it will display the parameters stored in IDAPI.CFG.

The default configuration file is listed in the IDAPI section of your WIN.INI file as CONFIGFILE01.

The BDE Configuration Utility provides the following menu commands for managing the BDE configuration file:

Command	Function
File | Open	Displays the Open dialog box, which enables you to select a .CFG file to view or edit.
File | Save	Saves any changes made to the current BDE configuration file.If the current file is not the default BDE configuration file, the Configuration Utility displays the Non-system Configuration File dialog box. If you want this file to become the new default BDE configuration file, choose Yes in that dialog box. Choose No to leave your current default configuration file unchanged.The changes take effect the next time you re-start all open BDE applications.
File | Save As	Saves the current .CFG settings under a different .CFG file name. You can name your BDE configuration file anything as long as it ends in .CFG and is no more than 12 characters long.If the file name already exists, the Configuration Utility displays the Overwrite Existing File dialog box. If you want to over-write the existing file (erasing any unique aliases or ODBC driver connections it may contain), click Yes. To cancel this operation, click No.
File | Merge	Merges another configuration file with the one already in use.
File | Exit	Exits the BDE Configuration Utility. If you made changes and did not save them, a warning appears. You can save your changes or exit without saving.

Drivers Page

The Drivers page controls how your application creates, sorts, and handles different kinds of tables. It is also used to add or remove SQL ODBC driver connections.

Driver Name lists the types of drivers installed at your workstation. STANDARD drivers are Paradox and dBASE; other drivers are for use with SQL servers, and are installed separately.

New Driver and Delete Driver enable you to add an ODBC driver connection to the list of available drivers. (For further information, see the BDE Configuration Utility Online Help.)

Parameters lists all the parameters tracked by the BDE Configuration Utility for the selected driver type, and their current settings.

To modify a setting, select the driver name and highlight the desired configuration parameters. Delete the old value and enter a new one in the appropriate text box.

- dBASE Settings
- Paradox Settings
- SQL Settings
- ODBC Settings

dBASE Settings

dBASE settings determine the way dBASE tables are created, sorted, and handled.

Settings	Description
VERSION	Internal version number of the dBASE driver.Do not modify.
TYPE	Type of server to which this driver helps you connect. FILE represents a standard, file-based server.Do not modify.
LANGDRIVER	Language driver used to determine table sort order and character set. U.S. default: DBASE ENU CP437.
LEVEL	Type of table format used to create dBASE temporary tables. Can be 3 (dBASE III and dBASE III Plus table format), 4 (dBASE IV table format), or 5 (dBASE 5.0 table format). Default: 5.
MDX BLOCK SIZE	Size of disk blocks dBASE allocates for .MDX files, in bytes. Can be any integer that is a multiple of 512. Default: 1024.
MEMO FILE BLOCK	Size of disk blocks dBASE allocates for memo (.DBT) files, in bytes. Can be any integer that is a multiple of 512. Default: 1024.

Paradox Settings

Paradox driver settings determine the way Paradox tables are created, sorted, and handled.

Settings	Description
VERSION	Internal version number of the Paradox driver.Do not modify.
TYPE	Type of server to which this driver helps you connect. FILE represents a standard, file-based server.Do not modify.
NET DIR	Directory containing the network control file (PDOXUSRS.NET).
LANGDRIVER	Language driver used to determine table sort order and character set. U.S. default: ascii.
LEVEL	Type of table format used to create temporary Paradox tables. Can be 3 (Paradox 3.5 and earlier table formats), 4 (Paradox 4.0 table format), or 5 (Paradox 5.0 table format). Default: 4. Note: Paradox level 4 tables are required to use BLOB fields, secondary indexes, and strict referential integrity.
BLOCK SIZE	Size of disk blocks used to store Paradox table records. Can be 1024, 2048, 3072, or 4096. Default: 2048.
FILL FACTOR	Percentage of current disk block which must be filled before Paradox will allocate another disk block for index files. Can be any integer ranging from 1 to 100. Default: 95. Note: Smaller values offer better performance but increase the size of indexes. Larger values give smaller index files but increase the time needed to create an index.
STRICTINTEGRTY	Specifies whether Paradox tables can be modified using applications that do not support referential integrity (Paradox 4.0). For example, if TRUE you will be unable to change a table with referential integrity using Paradox 4.0; if FALSE, you can change the table, but you risk the integrity of your data. Default: TRUE.

SQL Settings

If you have installed any Borland SQL Link drivers or created any ODBC driver connections, the list of drivers on the Drivers page will include more than just dBASE and Paradox.

SQL driver settings determine the way the Borland Database Engine connects to the target SQL server, and how the SQL databases are opened. The settings are slightly different for each driver. For further information on SQL Link driver settings, see the *Borland SQL Links Getting Started* manual. (Note: You may need to widen this help window to view the entire chart.)

All SQL Link drivers use at least the following common parameters:

Settings	Description
VERSION	Internal version number of the SQL Link driver.Do not modify.
TYPE	Type of server to which this driver helps you connect. SERVER represents a SQL server.Do not modify.
DLL	SQL Link dynamic link library name. For internal use only.Do not modify.
DRIVER FLAGS	Internal product-specific flag. Do not modify unless directed to do so by Borland support personnel.Default: NULL.
SERVER NAME	Name of the target SQL server. Use format appropriate for your server; for example, the InterBase server name takes the form: IB_SERVER:/PATH/DATABASE.GDB.
USER NAME	Default user name for accessing the target SQL server.
OPEN MODE	Default mode in which to open the target SQL database. Can be READ ONLY or READ/WRITE. Default: READ/WRITE.
SCHEMA CACHE SIZE	Number of SQL tables whose schema information will be cached. Possible values are 0 - 32. Default: 8.
LANGDRIVER	Language driver used to manipulate all data that originates from the SQL server. When the cursor is in the LANGDRIVER field, a scroll bar appears at the right side of the text box; use the scroll bar to display a list of language driver long names that work with your driver. U.S. default: NULL.
SQLQRYMODE	Method for handling queries to SQL data. Possible modes and their meanings are listed below. Default: NULL.
	NULL (blank setting) — Server-local. (Default mode) In server-local query mode, the query goes first to the SQL server. If the server is unable to perform the query, the query is performed locally.For a discussion of how Borland language drivers affect the processing of SQL queries, see the *Borland SQL Links Getting Started* manual.
	SERVER — Server-only. In server-only query mode, the query is sent to the SQL server. If the server is unable to perform the query, no local processing is performed.
	LOCAL — Local-only. In local-only query mode, the query is always performed locally.
SQLPASSTHRU MODE	Specifies whether or not the application will be able to access the SQL server via both desktop commands and pass-through SQL in the same alias connection. Possible modes and their meanings are listed below. Default: SHARED AUTOCOMMIT for Informix; NOT SHARED for all other SQL Link drivers.
	NOT SHARED(blank setting) — (InterBase, Oracle, Sybase default) Pass-through SQL and non-pass-through SQL do NOT share the same connection.
	SHARED AUTOCOMMIT — (Informix default) Pass-through SQL and non-pass-through SQL will share the same connection, and pass-through SQL will behave in a similar fashion to non-pass-through. This means that, as long as the user is not in an explicit client transaction or batch mode, pass-through SQL statements will be automatically committed.
	SHARED NOAUTOCOMMIT — Pass-through SQL and non-pass-through SQL will share the same connection, but the SQL driver will not automatically commit pass-through statements. In this mode, pass-through behavior is server-dependent.
SCHEMA CACHE TIME	Specifies how long table list information will be cached. (In BDE this happens when you call either DbiOpenTableList or DbiOpenFileList.) Possible modes and their meanings are listed below. Default: -1.
	-1 — (Default) The table list is cached until you close the database.

0	No table lists are cached.
1 through 2147483647	The table list is cached for the number of seconds specified in the setting.

ODBC Settings

All ODBC driver connections created through the BDE Configuration Utility use the common parameters shown in the following table. Also see Configuring Microsoft Open Database Connectivity (ODBC)

Settings	Description
VERSION	Internal version number of the ODBC driver.Do not modify.
TYPE	Uniquely identifies this ODBC driver connection. Can include any combination of 12 alphanumeric characters; the BDE Configuration Utility automatically prepends the characters ODBC_. For example, if the ODBC data source resides on a Sybase server whose servername is Silver, you might name the ODBC drive connection sysilver. The Configuration Utility converts this to ODBC_sysilver.
DLL	ODBC driver dynamic link library name. For internal use only.Do not modify.
ODBC DRIVER	ODBC driver used to connect the workstation to the target ODBC server.
DRIVER FLAGS	Internal product-specific flag. Do not modify unless directed to do so by Borland support personnel.Default: NULL.
USER NAME	Default user name for accessing the target ODBC server.
ODBC DSN	The name of the target ODBC data source.
OPEN MODE	Default mode in which to open the target ODBC database. Can be READ ONLY or READ/WRITE. Default: READ/WRITE.
SCHEMA CACHE SIZE	Number of SQL tables whose schema information will be cached. Possible values are 0 - 32. Default: 8.
SQLQRYMODE	Method for handling queries to SQL data. Possible modes and their meanings are listed in Table A.4. Default: NULL.
SQLPASSTHRU MODE	Specifies whether or not the Borland Database Engine application will be able to access the SQL server via both desktop commands and pass-through SQL in the same alias connection. Possible modes and their meanings are listed in Table A.5. Default: SHARED AUTOCOMMIT.

Aliases Page

An alias is a name and a set of parameters that describe a network resource. BDE-based applications use aliases to connect with shared databases. An alias is not required to access a local database, but it is required to access a SQL database.

The Aliases page enables you to add, delete, or modify database aliases.

Alias Names lists all the available aliases.

New Alias enables you to add a new alias.

Delete Alias enables you to delete any alias that is highlighted in the Alias Name box.

Parameters shows all the parameters of the currently-selected alias, with their current values.

Description briefly notes the purpose of the selected parameter.

Setting up an alias for a SQL database consists of assigning a name to, and customizing the access parameters for, a SQL server and database. Your SQL alias includes your user name and password on the target SQL server.

For information on adding or deleting an alias to the BDE configuration file, see DbiAddAlias and DbiDeleteAlias

For further information, see your *Borland SQL Links Getting Started* manual.

System Page

The System page controls the settings your application uses when it is initialized.

Parameters lists all the system and network parameters tracked by the BDE Configuration Utility, with their current values.

Description briefly notes the purpose of the selected parameter.

To change a setting, highlight the desired parameter. Then replace its old value with a new value.

Setting	Meaning
VERSION	Internal version of BDE.Do not modify.
LOCAL SHARE	The ability to share access to local data between an active BDE application and an active non-BDE application. Set to TRUE if you need to work with the same files through both an BDE and a non-BDE application at the same time. Default: FALSE.
MINBUFSIZE	Minimum amount of memory for database data cache, in kilobytes. Can be any integer between 32 and 65535. Must be less than the total amount of RAM available to Windows. Default: 128.
MAXBUFSIZE	Maximum amount of memory for database data cache, in kilobytes. Can be any integer greater than MINBUFSIZE and less than (or equal to) the total amount of RAM available to Windows. Must be a multiple of 128. Default: 2048.
LANGDRIVER	System language driver. Defaults to the OEM driver appropriate for a country's version of Windows; for example, ASCII for U.S. workstations.
MAXFILEHANDLES	Maximum number of file handles BDE uses. Can be any integer ranging from 5 to 256. High values improve performance but use more Windows resources. Default: 128.
SYSFLAGS	Internal BDE setting. Do not modify without specific instructions from Borland support personnel.
LOW MEMORY USAGE LIMIT	Maximum amount of low memory BDE will attempt to use, in kilobytes. Default: 32.
SQLQRYMODE	Method for handling queries to SQL data. Can be NULL (blank setting; default), SERVER, or LOCAL.Note: This parameter only appears if a Borland SQL Link driver is installed.

Date, Time, and Number Pages

The Date, Time, and Number pages control how your application converts string values into date, time and numeric values.

Parameters lists all the date, time, and number format parameters tracked by the Configuration Utility, with their current values.

Description briefly notes the purpose of the selected parameter.

- Date Settings
- Time Settings
- Number Settings

Date Settings

Date settings are used to convert string values to various date expressions. See International Compatibility

Setting	Meaning
SEPARATOR	Character used to separate the month, day, and year components of a date value; that is, the / in 12/31/96. The default is the character normally used in the country selected in the Windows Control Panel when any BDE application is installed.
MODE	Controls the order of the month, day, and year components. Can be 0 (for MDY), 1 (for DMY), or 2 (for YMD). The default is the order normally used in the country selected in the Windows Control Panel when any BDE application is installed.
FOURDIGITYEAR	Specifies the number of digits for the year value (four or two). If TRUE, years are expressed in four digits (such as, 1995). If FALSE, the default, years have two digits (93). Default: TRUE.
YEARBIASED	Tells the application whether or not it should add 1900 to years entered as two digits. For example, if TRUE and you enter 7/21/95, the date is interpreted as 7/21/1995. Default: TRUE.
LEADINGZEROM	Specifies whether or not single digit month values have a leading zero. For example, if TRUE and you enter 1/1/80, the date is interpreted as 01/1/80. Default: TRUE.
LEADINGZEROD	Controls whether or not single digit day values have a leading zero. For example, if TRUE and you enter 1/1/80, the date is interpreted as 1/01/80. Default: TRUE.Date settings are used to convert string values to date values.

Time Settings

Time settings are used to convert string values to various time expressions. See International Compatibility

Setting	Meaning
TWELVEHOUR	Specifies whether or not BDE applications express time values using a twelve-hour clock. For example, if TRUE 8:21 p.m. is expressed as 08:21 PM, and not 20:21. Default: TRUE.
AMSTRING	Character string used to indicate morning (before noon and after midnight) times, when TWELVEHOUR is TRUE. Default: AM.
PMSTRING	Character string used to indicate evening (after noon and before midnight) times, when TWELVEHOUR is TRUE. Default: PM.
SECONDS	Specifies whether or not time values include seconds. For example, if TRUE, 8:21:35 p.m. is expressed as 8:21:35 PM, and not 8:21 PM. Default: TRUE.
MILSECONDS	Specifies whether or not time values include milliseconds. For example, if TRUE, 8:21:35:54 PM. Default: FALSE.Time settings are used to convert string values to time values.

Number Settings

Number settings are used to convert string values to number values. See International Compatibility

Setting	Meaning
DECIMAL SEPARATOR	Character used to separate the decimal portion of a number from its integer portion; for example, the period (.) in 3.14. Defaults to the standard decimal separator used for your country, as specified in the Windows Control Panel (International Setting).
THOUSAND SEPARATOR	Character used to separate large numbers into their thousands components; for example, the commas (,) in 1,000,000.00. Defaults to the standard thousands separator used for your country, as specified in the Windows Control Panel (International Setting).
DECIMALDIGITS	Specifies the maximum number of decimal places to be used when converting string values to number values. Default: 2.
LEADINGZERON	Indicates whether numbers between 1 and -1 use leading zeros; for example, 0.14. instead of .14. Default: TRUE.

Configuring Microsoft Open Database Connectivity (ODBC)

This section contains sample configuration file blocks to help you understand the procedure for configuring ODBC. First configure the ODBC configuration file, then configure the BDE configuration file to support ODBC.

- Configuring the ODBC Configuration File
- Configuring the BDE Configuration File for ODBC
- ODBC Socket Configuration Entries
- AutoODBC

Configuring the ODBC Configuration File

Two files contain the configuration information for ODBC:

- The file odbcinst.ini in the windows directory lists the ODBC drivers.
- The file odbc.ini lists the ODBC data sources.

Use the administrative program ODBCADMIN to modify these files. The files are ASCII, but direct user editing is not recommended.

Here is a sample odbc.ini file:

```
[ODBC Data Sources]
    My Oracle7=VENDOR Oracle7

    [My Oracle7]
    Driver=C:\windows\system\OR706.DLL
    Description=ODBC Oracle7 Driver
    ServerName=X:ZAPPA
    Servers=
    LogonID=guest
    LockTimeOut=
    ArraySize=
    QEWSD=34480
```

The first block [ODBC Data Sources] lists the ODBC data sources and their associated drivers. Then, for each data source, there is a block that describes the datasource. One data source [My Oracle7] is shown in the example above.

Here is a sample odbcinst.ini file (the file that describes the drivers):

```
[ODBC Drivers]
    VENDOR Oracle7=Installed

[VENDOR Oracle7]
    Driver=C:\ODBC\OR706.DLL
    Setup=C:\ODBC\OR706.DLL
    APILevel=1
    ConnectFunctions=YYY
    DriverODBCVer=02.01
    FileUsage=0
    SQLLevel=1
```

The first block [ODBC Drivers] lists the installed drivers. The second block is the configuration block for the first installed drivers.

Each datasource in the odbc.ini file will have an installed driver (for example, VENDOR Oracle7) in the odbcinst.ini file.

Configuring the BDE Configuration File for ODBC

The configuration of BDE (at least as it relates to SQL) is similar to the ODBC configuration. You must specify a series DRIVERS (like the ODBC drivers) and ALIASES (much like the ODBC datasources).

Here is an example showing a Drivers section from the BDE configuration file (IDAPI.CFG):

```
ORACLE:
   INIT:
      VERSION:1.0
      TYPE:SERVER
      DLL:SQLD_ORA.DLL
      VENDOR INIT:ORA7WIN.DLL
      DRIVER FLAGS:NULL
   DB OPEN:
      SERVER NAME:NULL
      USER NAME:NULL
      NET PROTOCOL:NULL
      OPEN MODE:READ/WRITE
      SCHEMA CACHE SIZE:8
      LANGDRIVER:NULL
      SQLQRYMODE:NULL
      SQLPASSTHRU MODE:NOT SHARED
      SCHEMA CACHE TIME:NULL
```

Note that it is this entry ORACLE that gets associated with the aforementioned alias reference to a driver.

Here is an example of a Database Alias section of a BDE configuration file (IDAPI.CFG):

```
ORACLE7:
   DB INFO:
      TYPE:ORACLE
      PATH:NULL
   DB OPEN:
      SERVER NAME:ORACLE7
      USER NAME:guest
      NET PROTOCOL:SPX/IPX
      OPEN MODE:READ/WRITE
      SCHEMA CACHE SIZE:32
      LANGDRIVER:NULL
      SQLQRYMODE:NULL
      SQLPASSTHRU MODE:NOT SHARED
      SCHEMA CACHE TIME:-1
```

There are two sub-properties:

- INFO. The INFO information is used to associate the alias with the correct driver name (see TYPE: ORACLE).
- OPEN. The OPEN information is used to open the database alias.

ODBC Socket Configuration Entries

This example shows a Drivers section from the BDE configuration file (IDAPI.CFG). This section was added manually by using the Borland Database Engine Configuration Utility.

```
 ODBC_ORA7:
INIT:
   VERSION:1.0
   TYPE:SERVER
   DLL:IDODBC01.DLL
   ODBC DRIVER:VENDOR Oracle7
   DRIVER FLAGS:NULL
   LANGDRIVER:NULL
DB OPEN:
   USER NAME:guest
   ODBC DSN:My Oracle7
   OPEN MODE:READ/WRITE
   SCHEMA CACHE SIZE:8
   SQLQRYMODE:NULL
   LANGDRIVER:NULL
   SQLPASSTHRU MODE:NULL
                    SCHEMA CACHE TIME:NULL
```

Note that the DLL field for the driver is the ODBC socket .dll, NOT the ODBC .dll. The ODBC .dll is loaded implicitly, when the ODBC socket attempts to connect to a datasource. The datasource that it tries to open is "My Oracle7" (the DSN entry).

The following example shows a Database Alias section from the BDE configuration file (IDAPI.CFG). This section was added manually by using the Borland Database Engine Configuration Utility (BDECFG.EXE).

```
ODBC_ORACLE:
   DB INFO:
      TYPE:ODBC_ORA7
      PATH:NULL
   DB OPEN:
      USER NAME:guest
      ODBC DSN:My Oracle7
      OPEN MODE:READ/WRITE
      SCHEMA CACHE SIZE:8
      SQLQRYMODE:NULL
                    LANGDRIVER:NULL
      SQLPASSTHRU MODE:SHARED AUTOCOMMIT
      SCHEMA CACHE TIME:-1
```

AutoODBC

AutoODBC builds the ODBC socket datasource and driver names automatically for BDE, using the ODBC names from the ODBC configuration files, odbc.ini and odbcinst.ini.

Here is an example showing the IDAPI.CFG entries created by AutoODBC referring to the driver (VENDOR Oracle7) and datasource (My Oracle7):

```
My Oracle7:
   DB INFO:
      TYPE:VENDOR Oracle7
      PATH:NULL
   DB OPEN:
      USER NAME:guest
      ODBC DSN:My Oracle7
      OPEN MODE:READ/WRITE
      SCHEMA CACHE SIZE:8
      SQLQRYMODE:NULL
      SQLPASSTHRU MODE:SHARED AUTOCOMMIT
      SCHEMA CACHE TIME:-1

VENDOR Oracle7:
   INIT:
      VERSION:1.0
      TYPE:SERVER
      DLL:IDODBC01.DLL
      ODBC DRIVER:VENDOR Oracle7
      DRIVER FLAGS:NULL
   DB OPEN:
      USER NAME:NULL
      ODBC DSN:My Oracle7
      OPEN MODE:READ/WRITE
      SCHEMA CACHE SIZE:8
      SQLQRYMODE:NULL
      LANGDRIVER:NULL
      SQLPASSTHRU MODE:NULL
                        SCHEMA CACHE TIME:NULL
```

Application Development

This chapter describes 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

This section shows you how to get started programming with the Borland Database Engine (BDE). After following the steps and examples, you should have a simple EasyWin BDE sample application that gets a record from a table and displays the first two fields.

- 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.

Project Setup

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

1. Create the project or makefile.

MAIN.CPP	File to contain your code
DBERR.CPP	Error handling routine
IDAPI.LIB	BDE Import Library
MAIN.DEF	Module Definition file

2. Select the large memory model. By default, the large model uses far pointers for data and functions, which is what BDE expects. You can use other memory models but take care to use only far pointers with BDE functions.
3. For this simple application, set the target to be an EasyWin application. This way you don't have to deal with any Windows user interface issues.
4. Install the IDAPI.TOK file to support syntax highlighting for BDE functions and types. Directions on how to do this are included in the file \BDE\DOC\IDAPITOK.TXT. You can also incorporate the on-line help file, BDE.HLP into the BC 4.5 OpenHelp architecture, allowing context sensitive help on BDE types and functions.
5. Set the stack to a minimum of 20KB. (For Windows C applications, the stack size is set in the module definition file). This is recommended because BDE is fairly stack intensive, especially when doing batch operations and queries.
6. Increase the number of file handles available to the application. By default, BDE assumes it has access to 48 physical file handles (set by the BDE Configuration utility, System page | MAXFILEHANDLES option, 5-255). See System Page. Due to this default setting, it is usually best to set the number of file handles available to an application to a minimum of 68, by using the Windows API function SetHandleCount. See the Windows API help file for information on using SetHandleCount. This function is also used in the sample application included in this section.
7. Use the Debug Layer when developing applications. (See Using the Debug Layer.) The debug layer performs much stricter error checking than the regular DLL, resulting in fewer GP faults and less re-booting of your machine. It will also produce a trace output detailing which BDE functions were called by an application. Note that use of the debug layer requires the use of both the Debug DLL (Set using the DLLSwap utility), as well as a call to the BDESDK function DbiDebugLayerOptions.
8. Make certain to compile with Allocate enums as ints selected (In the BC 4.5 IDE, Options | Project | Compiler | Code Generation). A number of structures, such as CURProps, make use of Enumerations. Within the DLL, these are allocated as two byte values. Turning off this option will result in your code passing only one byte. This error generally manifests itself with stack corruption problems, such as GP faults when calling or returning from a function.
9. Within a module to contain BDE code, include the following header files:

   WINDOWS.H

   IDAPI.H

Basic Procedure

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

1. Increase the number of file handles available to the application
2. Initialize the Borland Database Engine
3. Enable the debug layer
4. Open a database object
5. Set the database object to point to the directory containing the table
6. Set the directory for temporary objects
7. Open a table, creating a cursor object
8. Get the properties of the table
9. Using these properties, allocate memory for a record buffer
10. Position the cursor on the desired record
11. Get the desired record from the cursor (table)
12. Get the desired fields from the record
13. Free all resources

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

Step 1: Increase the Number of File Handles Available to the Application

Make a call to the Windows API function SetHandleCount to increase the number of file handles available to an application in the Windows environment:

```
int HandlesAvail, HandlesWanted = 68;

HandlesAvail = SetHandleCount(HandlesWanted);
if (HandlesAvail < HandlesWanted)
{
    // Display message re: not enough available file handles
    return;
}
```

Note: In non-trivial cases it is recommended to determine the number of file handles that are specified in the BDE configuration file (IDAPI.CFG). Do this by using the DbiOpenCfgInfoList function. Also see Configuration Management and System Page for information on setting BDE initialization parameters.

Step 2: Initialize the Borland Database Engine

Initialize BDE by using the DbiInit function:

```
CHKERR(DbiInit(NULL));
```

The macro CHKERR is defined in the file DBERR.H, which is a part of the Error Handling module. This allocates system resources for the client.

Step 3: Enable the Debug Layer

Use this code to enable the debug layer, outputting trace information to a text file on disk:

```
DbiDebugLayerOptions(DEBUGON | OUTPUTTOFILE, "MYTRACE.INF");
```

In certain situations, you might also want to use the flag FLUSHEVERYOP, which forces output to the trace file after every operation. While this is slower, it is useful when a GP fault occurs.

Step 4: Open a Database Object

Now you are ready to open a database object.

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 modification in the future 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;        // Handle to the Database

hDb = 0;            // Initialize to zero for cleanup purposes

CHKERR_CLEANUP(
    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
                );

// At the end of the function
CleanUp:
    // Close only if open
    if (hDb != 0)
    {
        DbiCloseDatabase(&hDb);
    }
```

Step 5: Set the Database Object to Point to the Directory Containing theTable

You must set the working directory to the directory containing the table.

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 example tables):

```
CHKERR_CLEANUP(
    DbiSetDirectory(hDb,                    // Handle to the database which is being modified
        C:\\BDE\\EXAMPLES\\TABLES) // The new working directory
                );
```

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

Step 6: 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 writable.

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

```
CHKERR_CLEANUP(
    DbiSetPrivateDir(c:\\<SOMEDIR>)      // 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.

Step 7: 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 allows queries and tables to be accessed in the same method:

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

hCur = 0;                                   // Initialize to zero for cleanup purposes

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

CHKERR_CLEANUP(
    DbiOpenTable(hDb, // Handle to the database to which this cursor will be related
                 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. Optional. Paradox and SQL only
                 dbiREADWRITE,   // Open Mode - Read/Write or Read Only
                 dbiOPENSHARED, // Shared mode - SHARED or EXCL
                 xltFIELD,        // Translate mode - FIELD or NONE
                                  // FIELD: Convert data from table format to C format
                                  // NONE: Leave data in it's native state
                 FALSE, // Unidirectional - False means can navigate forward and back.
                 NULL,            // Optional Parameters.
                 &hCur)           // Handle to the cursor
               );

CleanUp:
    // Close only if open
    if (hCur != 0)
    {
        DbiCloseCursor(&hCur); // Note the use of DbiCloseCursor - there is no
                               // DbiCloseTable.
    }
```

Step 8: Get the Properties of the Table

You need to determine the size of the record buffer for the table. 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

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

curProps.iRecBufSize contains the size of the record buffer.

Step 9: 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 = NULL;          // For cleanup purposes

pRecBuf = (pBYTE)malloc(curProps.iRecBufSize * sizeof(BYTE));
if (pRecBuf == NULL)
{
    // Display some error message
    goto CleanUp;
}

CleanUp:
    if (pRecBuf)
    {
        free(pRecBuf);
    }
```

Step 10: 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 locate the current cursor position to 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.

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

Step 11: Get the Desired Record from the Cursor (Table)

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

```
CHKERR_CLEANUP(
    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
                       );
```

Step 12: 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 variable. 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. 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;

CHKERR_CLEANUP(
   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?
                     );
```

Step 13: 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 using malloc or new, all engine objects must also be cleaned up, including the cursor, database, and engine:

```
CleanUp:
    if (pRecBuf)
    {
          // Free the record buffer
          free(pRecBuf);
    }
    if (hCur !=0)
    {
          // Close the cursor
          DbiCloseCursor(&hCur);
    }
          // Close only if open
    if (hDb != 0)
    {
          // Close the database
          DbiCloseDatabase(&hDb);
    }

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

Accessing and Updating Tables

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

Phase	Task	BDESDK 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
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
Exit	Close the cursor	Call DbiCloseCursor
Exit	Close the database	Call DbiCloseDatabase
Exit	Exit the database engine	Call DbiExit

Preparing to Access a Table

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

- Initializing the Engine
- Opening a Database
- Opening a Table
- Preparing the Record Buffer and Retrieving Field Descriptors

Initializing the Engine

The first call that the application makes to BDESDK 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 pointer is normally passed as NULL, which forces the Borland Database Engine to search for the configuration file (IDAPI.CFG), and to use the configuration file's 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 WIN.INI file located in your Windows directory for a configuration file defined by an entry of [IDAPI] 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);
```

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 BDESDK 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, 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 DbiSetDirector :

```
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 can 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

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 has already been done at database open time.

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.

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

- Use the *pszIndexName* parameter in the form <tablename>.MDX if the index is within a production index.
- Use the *pszIndexTagName* parameter to specify the tag name of the index in an MDX 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.

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);
        ...
        }
```

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); < 0 (-1, -2. . .): None (SQL)
BOOL	bSoftDeletes	TRUE, if the cursor supports soft deletes (dBASE 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 defaults 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 ID, 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 an 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 *ixltFIELD*, 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.

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 BDESDK 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 BDESDK functions can simultaneously position the cursor and fetch a record into the record buffer.

Positioning The Cursor On A Crack

Some BDESDK 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 BDESDK 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 BDESDK 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 BDESDK 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 (\BDE\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.

Field-level Access

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

BDESDK type	C language 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

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 BDESDK 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

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

For dBASE 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

BDESDK 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).

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. But there is a set of BDESDK functions 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.

The Windows 3.1 version can read only up to 64K with each call to DbiGetBlob. For larger BLOBs, multiple calls must be made.

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 which 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);
}
```

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 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).

Sorting Tables

The BDESDK 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.

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:

Session Layer

Database Handle Layer

Table Cursor Layer

Acquired Locks

Table Lock Coexistence

Locking Strategy

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.

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.

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.

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 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.

Acquired Table Locks

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

DbiAcqTableLock can place a WRITE lock on the table, which prevents other users from updating a table. It can also place a READ lock on the table, which prevents other users from placing a WRITE lock on the table (which would keep your application from updating the table and guarantees that the table is not modified by another user while you read.)

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.

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.

Record Locks

Applications can acquire record locks at record retrieval time. Most BDESDK 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 BDESDK 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.

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.

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**	**Succeed**
Opens the table in share mode	**Fail**	**Succeed**	**Succeed**	**Succeed**

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 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.

SQL Optimistic Locking

With dBASE and Paradox, a record lock prevents another user from updating the record. BDE SQL drivers (and some ODBC drivers), however, 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. 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 the time that it takes to complete a single users'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.

SQL 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 committeed 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).

Querying Databases

Through the BDESDK interface, the Borland Database Engine (BDE) enables the client to use SQL or Query by Example (QBE) to access dBASE and Paradox tables on the PC (standard databases), as well as server-based SQL tables.

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

- Querying Paradox and dBASE Tables
- Querying Different Databases
- Executing Queries Directly
- Executing Queries in Stages

SQL Queries

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

If the appropriate BDE driver is installed, the BDESDK 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
QBE allows uniform access to data in Paradox or dBASE tables and tables in server-based databases.

Querying Paradox and dBASE 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 are supported for tables in standard databases:

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

Naming Conventions

When writing SQL statements to be used with dBASE 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 BDESDK style aliases. For example,

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

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

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

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, and HAVING

The following aggregates are supported:

SUM, AVG, MIN, MAX, COUNT

The following operators are supported:

, -, *, /, =, < >, IS NULL

UPDATE, INSERT, DELETE operations are allowed.

For example:

```
Select part_no
   from parts
   where part_no > 543
```

Data Definition Language

The DDL syntax for Paradox and dBASE 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 BDESDK:

```
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* Help.

Querying Different Databases

Through the BDESDK 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 "Heterogeneous Joins" in the Local SQL Online User Guide.

The following SQL statement shows a join of three tables from different platforms, 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
```

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);
```

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 SQL queries on servers). The application uses a separate function call for each stage:

1 To prepare the query and get a statement handle, call DbiQPrepare.
2 To change properties in the statement handle, call DbiSetProp.
3 To execute the prepared query, call DbiQExec.
4 To free resources bound to the query, call DbiQFree.

DbiQPrepare

This function is used to prepare a SQL or QBE query for subsequent execution. The application passes the database handle, specifies whether the query language is QBE or SQL, and passes the formulated query string. The function returns a statement handle for the prepared query.

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 DbiQPrepare. The property to be set can be the result table type, degree of liveness, or query mode for binding parameters on SQL servers. The following examples show how values are set for these properties:

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

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. 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 SQL statements sent to a SQL server, 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).

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 sesCFGMODE. 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 BDESDK 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:

Properties	System	Session	Database	Driver	Cursor	Statement
sysMAXPROPS	X	X	X	X	X	X
sysLOWMEUSAGE	X	X	X	X	X	X
sesMAXPROPS		X	X		X	X
sesSESSIONNAME		X	X		X	X
sesNETFILE		X	X		X	X
sesCFGMODE		X	X		X	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			
drvMAXPROPS				X	X	
drvDRIVERTYPE				X	X	
drvDRIVERVERSION				X	X	
curMAXPROPS					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

Retrieving Schema and System Information

A set of BDESDK functions return schema or system information. Some functions, in the format DbiOpenXXXList, can be used to return a cursor to an in-memory table whose records contain the requested information. Other functions in the format DbiGetXXXDescs 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

Inquiry functionStructures are supplied by the application. These function calls return descriptive information. 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 DbiOpenXXXList, 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 and DbiPutField. 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
DbiOpenDriverList	The virtual table contains only one CHAR field.
DbiOpenLdList	LDDesc
DbiOpenDatabaseList	DBDesc
DbiOpenUserList	USERDesc
DbiOpenLockList	LOCKDesc
DbiOpenFieldList	FLDDesc
DbiOpenFieldTypesList	FLDType
DbiOpenIndexTypesList	IDXType
DbiOpenTableList	TBLBaseDesc, TBLExtDesc, TBLFullDesc
DbiOpenTableTypesList	TBLType
DbiOpenFileList	FILEDesc
DbiOpenFamilyList	FMLDesc
DbiOpenIndexList	IDXDesc
DbiOpenRintList	RINTDesc
DbiOpenSecurityList	SECDesc
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 index.
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 functionStructures are supplied by the application. These function calls return descriptive information.

List function	Record structure of the virtual table returning the information
DbiGetIndexDesc	IDXDesc structure
DbiGetIndexDescs	Array of IDXDesc structures
DbiGetIndexTypeDesc	IDXType structure
DbiGetTableTypeDesc	TBLType structure
DbiGetDatabaseDesc	DBDesc structure
DbiGetFieldDescs	Array of FLDDesc structures
DbiGetFieldTypeDesc	FLDType structure
DbiGetDriverDesc	DRVType 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 using the BDESDK 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 tables only, 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 or dBASE 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 paramerter 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 and dBASE 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 BDESDK logical types are supported.

Modifying Table Structure

After a table has been created, the application can modify it using BDESDK 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 or Paradox indexes up to date by calling either of two BDESDK functions. DbiRegenIndex regenerates a single out-of-date index; the application specifies the index name. DbiRegenIndex regenerates out-of-date indexes on a table.

Restructuring a Table

Currently, for Paradox and dBASE 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 same table descriptor structure, CRTblDesc, used to create the table, but much of the information specified in the descriptor is different.

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 the engine 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 BDESDK 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.

ecbType	Event description
cbBATCHRESULT	Batch processing results
cbRESTRUCTURE	Restructure
cbTABLECHANGED	Table has changed
cbGENPROGRESS	Generic Progress report
cbINPUTREQ	Input request when a BDE driver needs to communicate with the end user
cbDBASELOGIN	Access encrypted dBASE files

Callback function declarations and associated parameter lists, function return types, and callback data types are defined in the file IDAPI.H, which is the application interface to the 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 BDESDK function call in progress by returning *cbrABORT*.

Data Source Independence

You can use these techniques to achieve data source independence:

- Qualify tablenames 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 common query engine.

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

Filtering Records

This section explains how to create an expression tree used in DbiAddFilter.

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 example, you may 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 Borland Database Engine (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.

Defining a Filter

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 application can use an 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.

Using an Expression Tree

An expression tree is a block of memory arranged as a series of nodes, which define the conditions of the filter. Each expression tree structure has the following parts:

- Expression Tree Header
- Expression Tree Node Area
- Literal Pool Area

For example, to define a filter to display only those records where CUST_NO>1500. An expression tree is created to pass to DbiAddFilter

This following chart represents the expression tree: CUST_NO > 1500.00

Binary node: GT (Offset 0)

Constant & Field Nodes: Field (Offset 8) Constant (Offset 16)

Literal & Constant Pool: Cust_No (Offset 0) 1500.0 (Offset 8)

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;                // Handle to the Database
        hDBICur         hCur;               // Handle to the table
        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
        UINT16          uSizeLiterals;      // Size of the literals
        UINT16          uTotalSize;         // Total size of filter expression
        UINT32          uNumRecs = 10;      // Number of records to display
        CANExpr         canExp;             // Contains the header information
        UINT16          Nodes[ ] =          // Nodes of the filter tree
   {
   // Offset 0
                nodeBINARY,             // canBinary.nodeClass
                canGT,                  // canBinary.canOp
                8,                      // canBinary.iOperand1
                16,                     // canBinary.iOperand2
                                        // Offsets in the Nodes array
   // Offset 8
                nodeFIELD,              // canField.nodeClass
                canFIELD,               // canField.canOp
                1,                      // canField.iFieldNum
                0,                      // canField.iNameOffset: szField is the
                                        // literal at offset 0
   // Offset 16
                nodeCONST,              // canConst.nodeClass
                fldFLOAT,       // canConst.iType
                8,                      // canConst.iSize
                8,                      // canConst.iOffset: fconst is the literal
                                        // at   offset strlen(szField)   1
};
CHAR    szField[] = CUST_NO;// Field name of third node of tree
FLOAT    fConst     = 1500.0;       // Value of constant for second node
```

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;
        UINIT16   iTotalSize;
        UINT16    iNodes;
        UINT16    iNodeStart;
        UINT16    iLiteralStart;
    } CANExpr;
    typedef CANExpr far *pCANExpr;
    typedef pCANExpr far *ppCANExpr;
```

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 of a field name containing a literal. A constant node points to a constant value within the literal pool.

Operator nodes are of different types:

- Relational
- Logical
- Arithmetic
- Miscellaneous

Operator Nodes, Relational

Enumerated type	Description
canISBLANK	Unary; blank operand
canNOTBLANK	Unary; non-blank operand
canEQ	Binary; equal to
can NE	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
can DIV	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:

1000 > FLD1 AND FLD2 <=2022.22

Expression Tree

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

```
UINT16 iFTEST [ ] =
   {
        1,                          //iVer
        85,                         //iTotalSize of pCANExpr structure passed in
        -1,                         //iNodes
        sizeof (CANExpr),   iNodeStart - offset into structure
        60   sizeof (CANExpr),      // iLiteralStart- offset into structure
//nodeStart :
        nodeBINARY,                 //0
        canAND,
        8,                          // iOperand1 offset
        34,                         // iOperand2 offset
        nodeBINARY,         // 8
        canGT,
        16,                         // iOperand1 offset
        26,                         // iOperand2 offset
        nodeCONST,                  // 16
        canCONST ,
        fldFLOAT,
        2,
        0,
        nodeFIELD ,                 // 26
        canFIELD,
        1,
        16,
        nodeBINARY,                 // 34
        canLE ,
        42,                         // iOperand1 offset
        50,                         // iOperand2 offset
        nodeFIELD,                  // 42
        canFIELD,
        2,
        21,
        nodeCONST,                  //50
        canCONST                    ,
        fldFLOAT ,
        sizeof (FLOAT),
        8,
};
//Literal pool start (offset 60)
FLOAT  XBLits0[ ] =         {1000};                    // 0
FLOAT  XBLits1[ ] =         {2022.22};                 // 8
CHAR            szXBFld1[ ] =       "FLD1" ;   // 16
CHAR            szXBFld2[ ] =       "FLD2" ;
```

Chart

The chart below represents the same Boolean expression: 1000 > FLD1 AND FLD2 < 2022.22
(Note that the offsets are shown in parentheses.)

Header:	- -					
Binary node:			AND (0)			
Binary nodes:		GT (8)		LE (34)		
Constant & field nodes:	CONST (16)	FIELD (26)		FIELD (42)	CONST (50)	
Literal / constant pool:	1000 (0)	FLD1 (16)			FLD2 (21)	2022.22 (8)

Database Driver Characteristics

The Borland Database Engine (BDE) requires a separate BDESDK driver to support each database format or datasource. To extend BDE to support an additional database system, you must install the appropriate driver. This section provides additional information about specific driver types that you may use.

- SQL Drivers
- Text Driver

SQL Drivers

All BDESDK 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.

- Passthrough SQL
- SQL Transaction Control
- SQL Connection
- SQL Record Caching
- SQL Record Modification Requirements
- SQL Record Modification Behavior
- SQL Record-locking Behavior
- SQL Table-locking Behavior
- SQL Asynchronous Queries
- SQL Performance Tips

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 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

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 2 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.

SQL Transaction Control

To control explicit transactions, use DbiBeginTran and DbiEndTran. Except for explicit transactions, the BDESDK 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.

Transaction Isolation Levels

BDE 2.5 supports extended transaction isolation levels. If an unsupported isolation level is specified in DbiBeginTran, the next-highest supported isolation level is used. If the requested iso 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

Informix
Not changed for BDE 2.5.

InterBase
Supports Repeatable Read and Read Committed.The wait mode has been changed 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.

The following table shows the changes from previous versions of BDE.

REQUESTED Isolation Level	ACTUAL Isolation Level Used 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).

SQL Connection

BDESDK connects to the SQL server database by using the following guidelines:

- BDESDK uses the server authorization scheme. The password is used in DbiOpenDatabase to connect to the server.
- Most BDESDK 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.

Live Caching

Live caching provides fuller BDESDK 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:

```
DbiGetPriorRecord (...)
```

The SQL driver uses a cache, rather than the server:

ID	Name
12321	Harry
12345	Beth
12666	Joe

The SQL driver terminates the query and clears the cache:

DbiSetToBegin (...)

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.
- 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.

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:

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 not a unique index.

```
DbiModifyRecord (hCursor, &myRecBuff, TRUE);
UPDATE Customer
SET Name = "Harold"
WHERE Id = 12321 AND Name = "Harry"
```

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 and Informix, 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.

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.

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 drivers that support this feature (currently, only Sybase SQL Link).

You can use the BDE Configuration Utility 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 cancelled. When a query is successfully cancelled, DBIERR_CANCEXCEPT Query cancelled is returned.

SQL Performance Tips

The following tips are suggested to help reduce unnecessary processing, and speed up performance:

- Use pass-through SQL for complex queries or stored procedures.
- 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.

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 quote character (") 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.

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. Visual dBASE for Windows and Paradox for Windows 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, DATE) 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 or dBASE) 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.

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

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 xxx.txt will be xxx.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 languagedriver to use. This is the base filename of the .LDfile used for localization purposes.

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.
- Datatype. 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 datatypes (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 format only.

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
DATE        - Date field. Format specified by IDAPI.CFG
TIME        - Time field. Format specified by IDAPI.CFG
TIMESTAMP - Date  Time field. Format specified by IDAPI.CFG
```

Note: You can specify Date and time formats in the BDE configuration utility

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.

Error Handling

BDESDK 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 an BDESDK call, any of the BDESDK subsystems may push an error context onto the common BDESDK error stack. This error context allows the application to examine potentially more detailed information about the cause of any error.

Several BDESDK 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 and debugging, see the following topics:

- Using DbiGetErrorEntry to Access the Error Stack
- Using DbiGetErrorString to Get a Detailed Error Message
- Using DbiGetErrorContext to Get More Specific Information

Includes a table showing error context types.

- Using DbiGetErrorInfo to Get Immediate Information

Includes a chart showing the DBIErrInfo structure.

- Using the Debug Layer

Using DbiGetErrorEntry to Access the Error Stack

Every error generated as a result of an BDESDK 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 BDESDK calls reset the error stack.

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.

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.

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 an BDESDK 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 occured. Retrieve the error string.
    DbiGetErrorString(rslt, dbiStatus);
 }
```

Using the Debug Layer

BDE provides a debug layer that allows you to examine the state of function parameters as you access functions. The debug layer is composed of the following files:

- Two BDESDK .DLL files containing all entry points for the BDESDK functions are shipped with the BDE: IDAPI01.DLL and DBG.DLL.
- IDAPI01.DLL does not contain the debug layer; it is recommended that you ship your application with this .DLL file. IDAPI01.DLL is installed by default.
- DBG.DLL contains the contents of IDAPI01.DLL and also includes the debug layer. It is recommended that you use this .DLL file during your development process.

Preparing to Use the Debug Layer

Before using the debug layer, follow these two steps:

1 Use the DLLSWAP utility to rename DBG.DLL to IDAPI01.DLL (and to rename IDAPI01.DLL to NODBG.DLL).

2 Call DbiInit.

Turning the Debug Layer On and Off

Important: The debug layer must be turned on after calling DbiInit and turned off before calling DbiExit.

To turn on the debug layer, make the following call:

```
DbiDebugLayerOptions (DEBUGON, NULL);
```

To turn off the debug layer, make the following call:

```
DbiDebugLayerOptions (0, NULL);
```

Specifying Debug Layer Options

The following options can be specified with DbiDebugLayerOptions:

Option	Result
DEBUGON	If specified, the debug layer is activated. (Note: The debug layer .DLL must be in place). If this option is not specified, the debug layer is deactivated.
OUTPUTTOFILE	If specified, debug layer trace information is directed to the file specified by pDebugFile. If pDebugfile is not specified, a default TRACE file is created.
FLUSHEVERYOP	If specified, flushes trace information to the TRACE file every time an BDESDK function is called. (Note: This option is expensive and markedly slows processing.) If not specified, trace information is flushed periodically.
APPENDTOLOG	If specified, the trace output is appended to the end of the existing pDebugFile file. If not specified, the trace output overwrites the existing pDebugFile.

Methods of Using the Debug Layer

To turn on the debug layer and output trace information to a default file, overwriting any existing information in the file:

```
DbiDebugLayerOptions(DEBUGON | OUTPUTTOFILE, NULL);
```

To turn on the debug layer and output trace information to the TRACE.INF file, overwriting any existing information in TRACE.INF:

```
DbiDebugLayerOptions(DEBUGON | OUTPUTTOFILE, "trace.inf");
```

To turn on the debug layer and output trace information to the TRACE.INF file, overwriting trace information after every BDESDK call:

```
DbiDebugLayerOptions(DEBUGON | OUTPUTTOFILE | FLUSHEVERYOP, "trace.inf");
```

To turn on the debug layer and output trace information to the TRACE.INF file, appending information to TRACE.INF while leaving existing information intact:

```
DbiDebugLayerOptions(DEBUGON | OUTPUTTOFILE | APPENDTOLOG,"trace.inf");
```

To turn off the debug layer:

```
DbiDebugLayerOptions(NULL, NULL);
```

Debug Layer Options

The following options are defined for the debug layer:

Option	Description
DEBUGON	Turns the debug layer on.
OUTPUTTOFILE	Trace information output to a file.
FLUSHEVERYOP	Trace information is overwritten after every BDESDK call.
APPENDTOLOG	New information goes to the end of the trace file.

Examining Trace Files

A sample trace file is shown here:

```
ENTERING ***** DbiOpenDatabase
   variable name = pszDbName      type name = pCHAR
      Pointer = 0000:0000
      String = (null)
   variable name = pszDbType      type name = pCHAR
      Pointer = 2767:2F78
      String = STANDARD
   variable name = eOpenMode      type name = DBIOpenMode
      code name = READWRITE, integer value = 0
   variable name = eShareMode     type name = DBIShareMode
      code name = OPENSHARED, integer value = 0
   variable name = pszPassword    type name = pCHAR
      Pointer = 0000:0000
      String = (null)
   variable name = iOptFlds       type name = UINT16
      Uint = 0
   variable name = pOptFldDesc    type name = pFLDDesc
      Pointer = 0000:0000
   variable name = pOptParams     type name = pBYTE
      Pointer = 0000:0000
   variable name = phDb type name = phDBIDb
      Pointer = 2767:BC56
```

Trace Information Output

If the debug layer determines that a parameter you have passed is invalid, the following information is conveyed:

- An "ERROR:" message is echoed to the trace file.
- The return value DBIERR_INVALIDPARAM is returned.

These error messages provide clues for determining the cause of a crash or corruption that occurred while running without the debug layer.

Examples: Error conditions and resulting error messages

The table below shows some possible error conditions and their resulting error messages:

Example of error condition	Resulting trace ERROR message
bDefault == 999	ERROR: NOT BOOLEAN
pszTableName == NULL	ERROR: NULL STRING POINTER
hCursor == <junk value>	ERROR: BAD POINTER OR BAD HANDLE
hCursor == NULL	ERROR: NULL POINTER OR NULL HANDLE

Using the Function Reference

You can find a complete description of each BDESDK function by looking in the task-related tables in Function Reference, Categorical or searching the complete list in Function Reference, Alphabetical. Each BDESDK 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 BDESDK function that retrieves information about the client application environment.

The following general conventions and definitions will assist you in understanding the DBESDK function reference:

- Syntax Conventions
- Variable Names
- Constants
- #Defines
- Typedefs
- Object Definitions
- Buffer Typedefs

Syntax Conventions

The syntax for BDESDK 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
Description	Summary description of function
Syntax	Diagram of the function and parameters
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	Parameters that are 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*]

Variable Names

Each variable name used in this reference begins with a standard prefix. 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.

Constants

The following table lists the constants used to define maximum limits throughout this reference:

Constant	Limit	Description
DBIMAXNAMELEN	31	Maximum object name limit (such as, table, field)
DBIMAXTBLNAMELEN	127	Maximum table name length
DBIMAXFLDSINKEY	16	Maximum number of fields in a key
DBIMAXKEYEXPLEN	220	Maximum key expression length
DBIMAXPATHLEN	81	Maximum path file name length (allocate 82)
DBIMAXEXTLEN	3	Maximum file extension length, not including the extension delimiter "."
DBIMAXDRIVELEN	2	Maximum drive length
DBIMAXMSGLEN	127	Maximum message length (allocate 128)
DBIMAXVCHKLEN	255	Maximum validity check length
DBIMAXPICTLEN	175	Maximum picture length
DBIMAXFLDSINSEC	256	Maximum fields in security specification
DBIMAXSCFIELDS	16	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
DBIMAXUSERNAMELEN	14	Maximum user name (general)
DBIMAXXBUSERNAMELEN	12	Maximum user name length for xBASE
DBIMAXBOOKMARKLEN	4104	Maximum bookmark length

#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
FLOAT	double
DATE	long
TIME	long
TIMESTAMP	double
DBIFN	pascal far
UINT16	DBIResult

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
FLOAT	far *pFLOAT
DATE	far *pDATE
TIME	far *pTIME
BOOL	far *pBOOL
TIMESTAMP	far *pTIMESTAMP
pBYTE	far *ppBYTE
pCHAR	far *ppCHAR
pBOOL	far *ppBOOL
DBIResult	far *pDBIResult

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

Buffer Typedefs

The following typedefs for buffers of various common sizes are defined:

Type	typedef	Description
DBIPATH	CHAR[DBIMAXPATHLEN1]	Holds a DOS path
DBINAME	CHAR[DBIMAXNAMELEN1]	Holds a name
DBIEXT	CHAR[DBIMAXEXTLEN1]	Holds a file extension
DBIDOTEXT	CHAR[DBIMAXEXTLEN2]	Holds a file extension including "."
DBIDRIVE	CHAR[DBIMAXDRIVELEN1]	Holds a drive name
DBITBLNAME	CHAR[DBIMAXTBLNAMELEN1]	Holds a table name
DBIUSERNAME	CHAR[DBIMAXUSERNAMELEN1]	Holds a user name
DBIKEY	UINT16[DBIMAXFLDSINKEY]	Holds a list of fields in a key
DBIKEYEXP	CHAR[DBIMAXKEYEXPLEN1];	Holds a key expression
DBIVCHK	BYTE[DBIMAXVCHKLEN1]	Holds a validity check
DBIPICT	CHAR[DBIMAXPICTLEN1]	Holds a picture clause
DBIMSG	CHAR[DBIMAXMSGLEN1]	Holds an error message

Function Reference, Categorical

Each of the BDESDK functions documented in this reference fall into one of the categories listed in the table below:

Function Type	Purpose
Environment	Returns information or affects the client application environment.
Session	Returns information or affects a session.
Error handling	Returns information or performs related tasks.
Locking	Returns information or affects locks.
Cursor	Returns information or affects cursors and bookmarks.
Index	Returns information or affects indexes.
Query	Performs query tasks.
Database	Returns information or performs related tasks.
Table	Returns information or performs table-wide operations.
Data access	Performs specific data access operations.
Capability or schema	Returns information about database schema.
Date/time/number	Handles formats for the session.
Transaction	Returns information or performs related tasks.

Environment Functions

Each BDESDK 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. Or the function performs 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).
DbiAnsiToNative	Multipurpose translate function.
DbiDebugLayerOptions	Activates, deactivates, or sets options for the BDESDK debug layer.
DbiDeleteAlias	Deletes an alias from the BDE configuration file (IDAPI.CFG).
DbiExit	Disconnects the client application from BDESDK.
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 BDESDK system configuration information.
DbiGetSysInfo	Retrieves system status and information.
DbiGetSysVersion	Retrieves the system version information, including the engine version number, date, and time, and the client interface version number.
DbiInit	Initializes the BDESDK environment.
DbiLoadDriver	Loads a given driver.
DbiNativeToAnsi	Translates an OEM string 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.
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.
DbiUseIdleTime	Allows BDESDK to accomplish background tasks during times when the client application is idle.

Session Functions

Each BDESDK 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 BDESDK 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 BDESDK 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 BDESDK 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 BDESDK 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).
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.
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.
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.
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 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 BDESDK 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).
DbiSwitchToIndex	Allows the user to change the active index order of the given cursor.

Query Functions

Each BDESDK function listed below performs a Query task.

Function	Description
DbiGetProp	Returns a property of an object.
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.
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.
DbiQPrepareExt	Prepares a SQL or QBE query for execution, and returns a handle to a statement containing the prepared query. In addition, provides an option that allows the user to modify or update the resulting record set.
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.

Database Functions

Each BDESDK function listed below returns information about a specific database, available databases, or performs 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 BDESDK 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.
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.
DbiQInstantiateAnswer	Creates a permanent table from a cursor handle.
DbiPackTable	Optimizes table space by rebuilding the table associated with the cursor and releasing any free space.
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 BDESDK 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 (tuple) 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 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 BDESDK function listed below returns information about capabilties, or about the 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.
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 BDESDK 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 DATE 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

Each BDESDK function listed below begins, ends, or returns information about a transaction.

Function	Description
DbiBeginTran	Begins a transaction.
DbiEndTran	Ends a transaction.
DbiGetTranInfo	Retrieves the transaction state.

Function Reference, Alphabetical

DbiGetBlobHeading
DbiGetBlobSize
DbiGetBookMark
DbiGetCallBack
DbiGetClientInfo
DbiGetCurrSession
DbiGetCursorForTable
DbiGetCursorProps
DbiGetDatabaseDesc
DbiGetDateFormat
DbiGetDirectory
DbiGetDriverDesc
DbiGetErrorContext
DbiGetErrorEntry
DbiGetErrorInfo
DbiGetErrorString
DbiGetField
DbiGetFieldDescs
DbiGetFieldTypeDesc
DbiGetFilterInfo
DbiGetIndexDesc
DbiGetIndexDescs
DbiGetIndexForField
DbiGetIndexSeqNo
DbiGetIndexTypeDesc
DbiGetLdName
DbiGetLdObj
DbiGetLinkStatus
DbiGetNetUserName
DbiGetNextRecord
DbiGetNumberFormat
DbiGetObjFromName
DbiGetObjFromObj
DbiGetPriorRecord
DbiGetProp
DbiGetRecord
DbiGetRecordCount
DbiGetRecordForKey
DbiGetRelativeRecord
DbiGetRintDesc
DbiGetSeqNo
DbiGetSesInfo
DbiGetSysConfig
DbiGetSysInfo
DbiGetSysVersion
DbiGetTableOpenCount
DbiGetTableTypeDesc
DbiGetTimeFormat
DbiGetTranInfo

DbiGetVchkDesc
DbiInit
DbiInitRecord
DbiInsertRecord
DbiIsRecordLocked
DbiIsTableLocked
DbiIsTableShared
DbiLinkDetail
DbiLinkDetailToExp
DbiLoadDriver
DbiMakePermanent
DbiModifyRecord
DbiNativeToAnsi
DbiOpenBlob
DbiOpenCfgInfoList
DbiOpenDatabase
DbiOpenDatabaseList
DbiOpenDriverList
DbiOpenFamilyList
DbiOpenFieldList
DbiOpenFieldTypesList
DbiOpenFieldXlt
DbiOpenFileList
DbiOpenIndex
DbiOpenIndexList
DbiOpenIndexTypesList
DbiOpenLdList
DbiOpenLockList
DbiOpenRintList
DbiOpenSecurityList
DbiOpenSPList
DbiOpenSPParamList
DbiOpenTable
DbiOpenTableList
DbiOpenTableTypesList
DbiOpenUserList
DbiOpenVchkList
DbiPackTable
DbiPutBlob
DbiPutField
DbiQExec
DbiQExecDirect
DbiQExecProcDirect
DbiQFreeDbiQInstantiateAnswer
DbiQPrepare
DbiQPrepareExt
DbiQPrepareProc
DbiQSetParams
DbiQSetProcParams

DbiReadBlock
DbiRegenIndex
DbiRegenIndexes
DbiRegisterCallBack
DbiRelPersistTableLock
DbiRelRecordLock
DbiRelTableLock
DbiRenameTable
DbiResetRange
DbiSaveChanges
DbiSetCurrSession
DbiSetDateFormat
DbiSetDirectory
DbiSetFieldMap
DbiSetLockRetry
DbiSetNumberFormat
DbiSetPrivateDir
DbiSetProp
DbiSetRange
DbiSetTimeFormat
DbiSetToBegin
DbiSetToBookMark
DbiSetToCursor
DbiSetToEnd
DbiSetToKey
DbiSetToRecordNo
DbiSetToSeqNo
DbiSortTable
DbiStartSession
DbiSwitchToIndex
DbiTimeDecode
DbiTimeEncode
DbiTimeStampDecode
DbiTimeStampEncode
DbiTranslateField
DbiTranslateRecordStructure
DbiTruncateBlob
DbiUndeleteRecord
DbiUnlinkDetail
DbiUseIdleTime
DbiVerifyField
DbiWriteBlock

DbiAcqPersistTableLock

Syntax
DBIResult DBIFN DbiAcqPersistTableLock (*hDb*, *pszTableName*, [*pszDriverType*]);

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 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.

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.

dBASE: This function is not supported for dBASE 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 tables.

See also
DbiOpenLockList

DbiAcqTableLock

Syntax
DBIResult DBIFN DbiAcqTableLock (*hCursor*, *eLockType*);

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: If a READ lock is attempted, it is automatically upgraded to a WRITE lock.

Paradox: Both READ locks and WRITE locks can be acquired.

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 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.

DbiActivateFilter

Syntax
DBIResult DBIFN DbiActivateFilter (*hCursor*, [*hFilter*]);

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

DbiAddAlias

Syntax
DBIResult DbiAddAlias([*hCfg*], *pszAliasName*, *pszDriverType*, *pszParams*, *bPersistent*);

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 passed in, it must reference a driver name that exists in the configuration file being modified.

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).

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"

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 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.

Prerequisites
DbiInit must be called prior to calling DbiAddAlias.

DbiResult return values

DBIERR_INVALIDPARAM	Null alias name, or one of the following was encountered as in *pszDriverType*: *szASCII*, *szDBASE*, *szPARADOX*. In the case of the latter, use a NULL *pszDriverType* to indicate a STANDARD database.
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

DbiAddFilter

Syntax
DBIResult DBIFN DbiAddFilter (*hCursor*, [*iClientData*], [*iPriority*], [*bCanAbort*], *pcanExpr*, [*pfFilter*], *phFilter*);

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. 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.

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

DbiAddIndex

Syntax
DBIResult DBIFN DbiAddIndex (*hDb*, *hCursor*, *pszTableName*, [*pszDriverType*], *pIdxDesc*, [*pszKeyviolName*]);

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.

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 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 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 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)
Optional. Specifies key violation table name.

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.

SQL: The client application must have the appropriate privileges to add indexes.

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.

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 maintained or primary, 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 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

DbiAddPassword

Syntax
DBIResult DBIFN DbiAddPassword (*pszPassword*);

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

DbiAnsiToNative

Syntax
DBIResult DBIFN DbiAnsiToNative (*pLdObj*, *pOemStr*, *pAnsiStr*, *iLen*, *pbDataLoss*);

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 Translation completed successfully.

See also
DbiNativeToAnsi, DbiGetLdObj

DbiAppendRecord

Syntax
DBIResult DBIFN DbiAppendRecord (*hCursor*, *pRecBuf*);

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: 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).
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.

DbiBatchMove

Syntax

DBIResult DBIFN DbiBatchMove (*pSrcTblDesc*, *hSrcCur*, *pDstTblDesc*, *hDstCur*, *ebatMode*, *iFldCount*, *pSrcFldMap*, *pszIndexName*, *pszIndexTagName*, *iIndexId*, [*pszKeyviolName*], [*pszProblemsName*], [*pszChangedName*], *p1ProbRecs*, *p1KeyvRecs*, *p1ChangedRecs*, *bAbortOnFirstProb*, *bAbortOnFirstKeyviol*, *p1RecsToMove*, *bTransliterate*);

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, BDESDK 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 DBIMAXPATHLEN1 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, BDESDK 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 DBIMAXPATHLEN1 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. All records that are updated or subtracted from the source table are placed here. 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, BDESDK 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 DBIMAXPATHLEN1 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. (Records from the source table that don't match are not added.)
batAPPENDUPDATE	Adds non-matching records to the destination table and overwrites matching records.
batSUBTRACT	Deletes matching records from the destination table.
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.

Note: In-memory tables are not supported as source tables.

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.

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

DbiBcdFromFloat

Syntax
DBIResult DBIFN DbiBcdFromFloat (*piVal*, *iPrecision*, *iPlaces*, *pBcd*);

Description
DbBcdFromFloat converts a number in the BDE logical FLOAT format into the BDE logical binary coded decimal (BCD) format.

Parameters
piVal Type: Double (Input)
Specifies the FLOAT data to convert.

iPrecision Type: Word (Input)
Specifies the precision of the BCD number. This number must be 32.

iPlaces Type: Word (Input)
Specifies the number of decimals of the BCD number.

pBcd Type: FMTBcd (Output)
Pointer to the client buffer that receives the BCD number. The BDE logical BCD format has a length which equals (*iPrecision* 2).

DbiBcdToFloat

Syntax
DBIResult DBIFN DbiBcdToFloat (*pBcd*, p*iVal*);

Description
DbiBcdToFloat converts a number in the BDE logical binary coded decimal (BCD) format to the BDE FLOAT format.

Parameters
pBcd Type: FMTBcd (Output)
Specifies the binary coded decimal (BCD) data to convert.

piVal Type: Double (Input)
Pointer to the client buffer that receives the FLOAT number.

DbiBeginLinkMode

Syntax
DBIResult DBIFN DbiBeginLinkMode (*phCursor*);

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 to *phCursor*) will result in an error.

DbiResult return values
DBIERR_NONE The cursor was successfully converted to a linked cursor.

See also
DbiEndLinkMode, DbiLinkDetail, DbiLinkDetailToExp, DbiUnlinkDetail, DbiGetLinkStatus

DbiBeginTran

Syntax
DBIResult DBIFN DbiBeginTran (*hDb*, *eXIL*, *phXact*);

Description
DbiBeginTran begins a transaction for a SQL server.

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.

Prerequisites
A valid database handle must be obtained from a SQL server.

DbiResult return values
DBIERR_NONE	The transaction has begun successfully.
DBIERR_ACTIVETRAN	There is already an active transaction.

DbiCheckRefresh

Syntax
DBIResult DBIFN DbiCheckRefresh (VOID);

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

DbiCloneCursor

Syntax
DBIResult DBIFN DbiCloneCursor (*hCurSrc*, *bReadOnly*, *bUniDirectional*, *phCurNew*);

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. [MORE]

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 only be Unidirectional. [MORE]

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: All indexes open on the source cursor are open on the clone.

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 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

DbiCloseCursor

Syntax
DBIResult DBIFN DbiCloseCursor (*phCursor*);

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

DbiCloseDatabase

Syntax
DBIResult DBIFN DbiCloseDatabase (*phDb*);

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, Paradox, and Text tables 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

DbiCloseFieldXlt

Syntax
DBIResult DBIFN DbiCloseFieldXlt (*hXlt*);

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

DbiCloseIndex

Syntax
DBIResult DBIFN DbiCloseIndex (*hCursor*, *pszIndexName*, *iIndexId*);

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 maintained index.

iIndexId Type: UINT16 (Input)
Currently not used.

Usage
DbiCloseIndex is applicable only with dBASE tables. It is used primarily to manipulate non-maintained indexes. DbiCloseIndex cannot close a current index, or a maintained 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 an 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 maintained 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

DbiCloseSession

Syntax
DBIResult DBIFN DbiCloseSession (*hSes*);

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 BDESDK 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

DbiCompareBookMarks

Syntax
DBIResult DBIFN DbiCompareBookMarks (*hCur*, *pBookMark1*, *pBookMark2*, *pCmpBkmkResult*);

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

pCmpBlmkResult

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.

DbiCompareKeys

Syntax
DBIResult DBIFN DbiCompareKeys (*hCursor*, *pKey1*, [*pKey2*], *iFields*, *iLen*, *piResult*);

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

DbiCopyTable

Syntax
DBIResult DBIFN DbiCopyTable (*hDb*, *bOverwrite*, *pszSrcTableName*, *pszSrcDriverType*, *pszDestName*);

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 and dBASE 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:

dBASE: For dBASE tables, default family members include

- The table (usually ends with a .DBF extension)
- BLOB file (usually <*tablename*>.DBT)
- Production index (usually <*tablename*>.MDX)

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.

SQL: The DbiCopyTable function copies only the table itself. The indexes are not copied.

Prerequisites
A read lock is required on source dBASE 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

DbiCreateInMemTable

Syntax
DBIResult DBIFN DbiCreateInMemTable (*hDb*, *pszName*, *iFields*, *pfldDesc*, *phCursor*);

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 BDESDK 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.

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

DbiCreateTable

Syntax
DBIResult DBIFN DbiCreateTable (*hDb*, *bOverWrite*, *pcrTblDsc*);

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.

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.

Prerequisites
If the client chooses to overwrite an existing table; the existing table must be closed.

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_UNKNOWNTABLETYPE	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 indexname 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

DbiCreateTempTable

Syntax
DBIResult DBIFN DbiCreateTempTable (*hDb*, *pcrTblDsc*, *phCursor*);

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, 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

DbiDateDecode

Syntax
DBIResult DBIFN DbiDateDecode (*dateD*, *piMon*, *piDay*, *piYear*);

Description
DbiDateDecode decodes DATE into separate month, day, and year components.

Parameters
dateD Type: DATE (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

DbiDateEncode

Syntax
DBIResult DBIFN DbiDateEncode (*iMon*, *iDay*, *iYear*, *pdateD*);

Description
DbiDateEncode encodes separate date components into DATE 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: pDATE (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

DbiDeactivateFilter

Syntax
DBIResult DBIFN DbiDeactivateFilter (*hCursor*, [*hFilter*]);

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

DbiDebugLayerOptions

Syntax
DBIResult DBIFN DbiDebugLayerOptions (*iOption*, *pDebugFile*);

Description
DbiDebugLayerOptions is used to activate, deactivate, or set options for the BDESDK debug layer.

Parameters
iOption Type: UINT16 (Input)
Specifies a union of debug layer options.

pDebugFile Type: pCHAR (Input)
Specifies a trace file into which trace information is written. (This parameter is valid only if the option OUTPUTTOFILE is specified.)

Usage
The debug layer has two purposes:

- It provides a more advanced level of parameter checking.
- It enables the application to output trace information, giving a detailed breakdown of the parameters passed into BDESDK functions. The trace includes error messages that are generated when variables contain invalid data.

The trace file name is specified in *pDebugFile* as *<filename.ext>*. If that file reaches 500K, then the contents are rolled over to a *<filename.old>* file. The trace file capacity is limited to 1MB. When the new trace file reaches 500K, its contents are rolled over, overwriting the *<filename.old>* file.

Prerequisites
Important: Before calling DbiDebugLayerOptions, the proper dynamic link library must be made available to the core file, IDAPI01.DLL. To do this, run the standalone utility DLLSWAP.EXE, which is used to swap between DBG.DLL and NODBG.DLL. DLLSWAP.EXE makes it known which DLL is currently available.

To accomplish the same task without the use of DLLSWAP.EXE, locate the directory where the BDESDK DLLs are installed. IDAPI01.DLL is the main entry point to BDESDK.

If DBG.DLL is listed in the directory, the debug layer is not enabled. To enable the debug layer, rename IDAPI01.DLL to NODBG.DLL, and rename DBL.DLL to IDAPI01.DLL.

If NODBG.DLL is listed in the directory, the debug layer is enabled. To disable the debug layer, rename IDAPI01.DLL to DBG.DLL, and rename NODBG.DLL to IDAPI01.DLL.

Take special care when using the debug layer in a multi-session environment. Debug layer state is shared by all concurrent sessions. For example, if one session has set the debug layer on for tracing, all sessions are traced.

Tracing imposes a considerable overhead. For this reason, when you are trying to isolate a problem, avoid tracing within long loops.

DbiResult return values
DBIERR_NONE The debug layer options have been successfully specified.

iOption

The possible debug layer options are:

Option	Result
DEBUGON	If specified, the debug layer is activated. (Note: The debug layer .DLL must be in place.) If this option is not specified, the debug layer is deactivated.
OUTPUTTOFILE	If specified, debug layer trace information is directed to the file specified by pDebugFile. If pDebugfile is not specified, a default trace file is created.
FLUSHEVERYOP	If specified, flushes trace information to the trace file every time an BDESDK function is called. (Note: This option is expensive, and markedly slows processing.) If not specified, trace information is flushed periodically.
APPENDTOLOG	If specified, the trace output is appended to the end of the existing pDebugFile file. If not specified, the trace output overwrites the existing pDebugFile.

DbiDeleteAlias

Syntax
DBIResult DbiDeleteAlias ([*hCfg*], *pszAliasName*);

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

DbiDeleteIndex

Syntax

DBIResult DBIFN DbiDeleteIndex (*hDb*, *hCursor*, *pszTableName*, [*pszDriverType*], *pszIndexName*, *pszIndexTagName*, *iIndexId*);

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.

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 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 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 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 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: 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.

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

DbiDeleteRecord

Syntax
DBIResult DBIFN DbiDeleteRecord (*hCursor*, [*pRecBuf*]);

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: 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.

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 only), DbiUndeleteRecord (dBASE only)

DbiDeleteTable

Syntax
DBIResult DBIFN DbiDeleteTable (*hDb*, *pszTableName*, [*pszDriverType*]);

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 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 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 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

DbiDoRestructure

Syntax

DBIResult DBIFN DbiDoRestructure (*hDb*, *iTblDescCount*, *pTblDesc*, *pszSaveAs*, [*pszKeyviolName*], [*pszProblemsName*], *bAnalyzeOnly*);

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 DBIMAXPATHLEN1 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 DBIMAXPATHLEN1 bytes).

bAnalyzeOnly Type: BOOL (Input)
Not currently used.

Usage

Paradox: For Paradox only, 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: Not currently supported for SQL.

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*

DbiDropFilter

Syntax
DBIResult DBIFN DbiDropFilter (*hCursor*, [*hFilter*]);

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

DbiDropPassword

Syntax
DBIResult DBIFN DbiDropPassword (*pszPassword*);

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

DbiEmptyTable

Syntax
DBIResult DBIFN DbiEmptyTable (*hDb*, *hCursor*, *pszTableName*, [*pszDriverType*]);

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 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 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 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

DbiEndLinkMode

Syntax
DBIResult DBIFN DbiEndLinkMode (*phCursor*);

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

DbiEndTran

Syntax
DBIResult DBIFN DbiEndTran (*hDb*, *hXact*, *eEnd*);

Description
DbiEndTran ends a transaction for a SQL server.

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

SQL: This function is supported with SQL server databases only.

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.

DbiExit

Syntax
DBIResult DBIFN DbiExit (VOID);

Description
DbiExit disconnects the client application from BDESDK.

Usage
DbiExit uninitializes the engine for use by this client and releases all resources allocated by the client application. DbiExit should be the last DBI/BDESDK 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 the engine has been removed, the user must reinitialize the engine before any BDESDK functions can be called.

DbiResult return values

DBIERR_NONE	The connection to the engine has been successfully removed.

See also
DbiInit

DbiExtractKey

Syntax
DBIResult DBIFN DbiExtractKey (*hCursor*, [*pRecBuf*], *pKeyBuf*);

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

DbiForceReread

Syntax
DBIResult DBIFN DbiForceReread (*hCursor*);

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 only ensures that the buffered data is current at the time of the call. Use DbiForceReread or DbiCheckRefresh periodically to ensure current data. Use record locking to prevent other users from updating records being modified by this cursor.

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

DbiFormFullName

Syntax
DBIResult DBIFN DbiFormFullName (*hDb*, *pszTableName*, *pszDriverType*, *pszFullName*);

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.

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.

DbiResult return values
DBIERR_NONE	The table name has been successfully returned.
DBIERR_INVALIDFILENAME	The specified table name is invalid.

See also
DbiSetDirectory

DbiFreeBlob

Syntax
DBIResult DBIFN DbiFreeBlob (*hCursor*, *pRecBuf*, *iField*);

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

Syntax
DBIResult DBIFN DbiGetBlob (*hCursor*, *pRecBuf*, *iField*, *iOffSet*, *iLen*, *pDest*, *piRead*);

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* must also be less 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.

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

DbiGetBlobHeading

Syntax
DBIResult DBIFN DbiGetBlobHeading (*hCursor*, *iField*, *pRecBuf*, *pDest*);

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: This function is not supported for dBASE 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 or dBASE.

See also
DbiPutBlob, DbiTruncateBlob, DbiFreeBlob, DbiGetBlob, DbiGetBlobSize

DbiGetBlobSize

Syntax
DBIResult DBIFN DbiGetBlobSize (*hCursor*, *pRecBuf*, *iField*, *piSize*);

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

Syntax
DBIResult DBIFN DbiGetBookMark (*hCur*, *pBookMark*);

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: For dBASE 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 is NULL.

See also
DbiSetToBookMark, DbiCompareBookMarks, DbiGetCursorProps

DbiGetCallBack

Syntax
DBIResult DBIFN DbiGetCallBack (*hCursor*, *ecbType*, *piClientData*, *piCbBufLen*, *ppCbBuf*, *ppfCb*);

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 pass-through 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

DbiGetClientInfo

Syntaax
DBIResult DBIFN DbiGetClientInfo (*pclientInfo*);

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

DbiGetCurrSession

Syntax
DBIResult DBIFN DbiGetCurrSession (*phSes*);

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 of the default session if no sessions have been started explicitly (with DbiStartSession) by the client application.

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

DbiGetCursorForTable

Syntax
DBIResult DBIFN DbiGetCursorForTable ([*hDb*], *pszTableName*, [*pszDriverType*], *phCursor*);

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

DbiGetCursorProps

Syntax
DBIResult DBIFN DbiGetCursorProps (*hCursor*, *pcurProps*);

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 using DbiGetProp. This function can be called immediately after DbiOpenTableto retrieve information necessary to allocate the record buffer and the array for the field descriptors in the table.

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

DbiGetDatabaseDesc

Syntax
DBIResult DBIFN DbiGetDatabaseDesc (*pszName*, *pdbDesc*);

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

DbiGetDateFormat

Syntax
DBIResult DBIFN DbiGetDateFormat (*pfmtDate*);

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

DbiGetDirectory

Syntax
DBIResult DBIFN DbiGetDirectory (*hDb*, *bDefault*, *pszDir*);

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)
Specifies whether to retrieve the default directory or the current working directory. [MORE]

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 standard 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

DbiGetDriverDesc

Syntax
DBIResult DBIFN DbiGetDriverDesc (*pszDriverType*, *pdrvType*);

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.

See also
DbiOpenDriverList

DbiGetErrorContext

Syntax
DBIResult DBIFN DbiGetErrorContext (*eContext*, *pszContext*);

Description
After receiving an error code back from a call, DbiGetErrorContext allows the client to probe the 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
No other calls (other than error handling functions) can be made after the call that produced the error.

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

DbiGetErrorEntry

Syntax
DBIResult DBIFN DbiGetErrorEntry (*uEntry*, *pulNativeError*, *pszError*);

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 BDESDK 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, DbiGetErrorEntry, DbiGetErrorString

DbiGetErrorInfo

Syntax
DBIResult DBIFN DbiGetErrorInfo (*bFull*, *pErrInfo*);

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 BDESDK 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

DbiGetErrorString

Syntax
DBIResult DBIFN DbiGetErrorString (*rslt*, *pszError*);

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. The engine 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 engine calls. In contrast, DbiGetErrorInfo returns information only on the last error logged by the engine.

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

DbiGetField

Syntax
DBIResult DBIFN DbiGetField (*hCursor*, *iField*, *pRecBuf*, [*pDest*], [*pbBlank*]);

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 an BDESDK 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 an BDESDK 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

DbiGetFieldDescs

Syntax
DBIResult DBIFN DbiGetFieldDescs (*hCursor*, *pfldDesc*);

Description
DbiGetFieldDescs retrieves a list of descriptors for all the fields in the table associated with *hCursor*.

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

DbiGetFieldTypeDesc

Syntax
DBIResult DBIFN DbiGetFieldTypeDesc (*pszDriverType*, *pszTableType*, *pszFieldType*, *pfldType*);

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

DbiGetFilterInfo

Syntax
DBIResult DBIFN DbiGetFilterInfo (*hCursor*, *hFilter*, *iFilterId*, *iFilterSeqNo*, *pFilterinfo*);

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 (Input)
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.

DbiGetIndexDesc

Syntax
DBIResult DBIFN DbiGetIndexDesc (*hCursor*, *iIndexSeqNo*, *pidxDesc*);

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.

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

DbiGetIndexDescs

Syntax
DBIResult DBIFN DbiGetIndexDescs (*hCursor*, *pidxDesc*);

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.

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

DbiGetIndexForField

Syntax
DBIResult DBIFN DbiGetIndexForField (*hCursor*, *iFld*, *bProdTagOnly*, [*pidxDesc*]);

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 tables 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: For dBASE tables, only simple indexes are considered because there are no composite indexes. Expression indexes are not considered.

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

DbiGetIndexSeqNo

Syntax
DBIResult DBIFN DbiGetIndexSeqNo (*hCursor*, *pszIndexName*, *pszTagName*, *iIndexId*, *piIndexSeqNo*);

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 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: For dBASE 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: 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

DbiGetIndexTypeDesc

Syntax
DBIResult DBIFN DbiGetIndexTypeDesc (*pszDriverType*, *pszIndexType*, *pidxType*);

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

DbiGetLdName

Syntax
DBIResult DBIFN DbiGetLdName (*pszDriver*, *pObjName*, *pLdName*);

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

DbiGetLdObj

Syntax
DBIResult DBIFN DbiGetLdObj (*hCursor*,**ppLdObj*);

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

DbiGetLinkStatus

Syntax
DBIResult DBIFN DbiGetLinkStatus (*hCursor*, *phCursorMstr*, *phCursorDet*, *phCursorSib*);

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

DbiGetNetUserName

Syntax
DBIResult DBIFN DbiGetNetUserName (*pszNetUserName*);

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.

DbiGetNextRecord

Syntax
DBIResult DBIFN DbiGetNextRecord (*hCursor*, [*eLock*], [*pRecBuf*], [*precProps*]);

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 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 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: If the *precProps* argument is supplied, the record number can be retrieved for the record (via the iPhyRecNum field of *precProps*). dBASE does 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. 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 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

DbiGetNumberFormat

Syntax
DBIResult DBIFN DbiGetNumberFormat (*pfmtNumber*);

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

DbiGetObjFromName

Syntax
DBIResult DBIFN DbiGetObjFromName (*eObjType*, [*pszObjName*], *phObj*);

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

DbiGetObjFromObj

Syntax
DBIResult DBIFN DbiGetObjFromObj (*hObj*, *eObjType*, *phObj*);

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.

DbiGetPriorRecord

Syntax
DBIResult DBIFN DbiGetPriorRecord (*hCursor*, [*eLock*], [*pRecBuf*], [*precProps*]);

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 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 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: If the *precProps* argument is supplied, the record number can be retrieved for the prior record (the *iPhyRecNum* field of the RECProps structure). dBASE does 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 and dBASE only).
DBIERR_NA	Cursor is unidirectional.

See also
DbiGetRecord, DbiGetNextRecord, DbiGetPriorRecord, DbiGetRelativeRecord, DbiGetField, DbiModifyRecord

DbiGetProp

Syntax
DBIResult DBIFN DbiGetProp (*hObj*, *iProp*, *pPropValue*, *iMaxLen*, *piLen*);

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.

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

Native Handles

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	--
ODBC Socket	HDBC 4	

stmtNATIVEHNDL, curNATIVEHNDL

	*ppropValue	*pilen
InterBase	gds_stmt_handle	4
Sybase	DBIERR_NOTSUPPORTED	--
Oracle	CDA 64	
Informix	DBIERR_NOTSUPPORTED	--
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. Currently, 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 Link 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

Syntax
DBIResult DBIFN DbiGetRecord (*hCursor*, [*eLock*], [*pRecBuf*], [*precProps*]);

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 dBASE and Paradox 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: If *precProps* is supplied, the record number can be retrieved for the current record (via the iPhyRecNum field of *precProps*). dBASE does 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.

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 and BASE only).

See also
DbiGetField, DbiGetNextRecord, DbiGetPriorRecord, DbiGetRelativeRecord

DbiGetRecordCount

Syntax
DBIResult DBIFN DbiGetRecordCount (*hCursor*, *piRecCount*);

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.

DbiGetRecordForKey

Syntax
DBIResult DBIFN DbiGetRecordForKey (*hCursor*, *bDirectKey*, *iField*s, *iLen*, *pKey*, [*pRecBuf*]);

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 (Input)
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 partial key to be matched.

pRecBuf Type: pBYTE (Input)
Pointer to the record buffer where the new current record is returned. Optional.

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

DbiGetRelativeRecord

Syntax
DBIResult DBIFN DbiGetRelativeRecord (*hCursor*, *iPosOffset*, [*eLock*], [*pRecBuf*], [*precProps*]);

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 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 dBASE 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 returns an error if the requested lock cannot be acquired. See the following section for SQL-specific locking behavior information.

dBASE: If the *precProps* argument is supplied, the record number can be retrieved for the record (the *iPhyRecNum* field of the RECProps structure). dBASE does 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.

Usage
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

DbiGetRintDesc

Syntax
DBIResult DBIFN DbiGetRintDesc (*hCursor*, *iRintSeqNo*, *printDesc*);

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

DbiGetSeqNo

Syntax
DBIResult DBIFN DbiGetSeqNo (*hCursor*, *piSeqNo*);

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.

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

DbiGetSesInfo

Syntax
DBIResult DBIFN DbiGetSesInfo (*psesInfo*);

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

DbiGetSysConfig

Syntax
DBIResult DBIFN DbiGetSysConfig (*psysConfig*);

Description
DbiGetSysConfig retrieves BDESDK 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

DbiGetSysInfo

Syntax
DBIResult DBIFN DbiGetSysInfo (*psysInfo*);

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

DbiGetSysVersion

Syntax
DBIResult DBIFN DbiGetSysVersion (*psysVersion*);

Description
DbiGetSysVersion retrieves the system version information, including the engine 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

DbiGetTableOpenCount

Syntax

DBIResult DBIFN DbiGetTableOpenCount (*hDb*, *pszTableName*, [*pszDriverType*],
piOpenCount);

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 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 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 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 BDESDK, 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.

This 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

DbiGetTableTypeDesc

Syntax
DBIResult DBIFN DbiGetTableTypeDesc (*pszDriverType*, *pszTableType*, *ptblType*);

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

DbiGetTimeFormat

Syntax
DBIResult DBIFN DbiGetTimeFormat (*pfmtTime*);

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

DbiGetTranInfo

Syntax
DBIResult DBIFN DbiGetTranInfo (*hDb*, *hXact*, *pxInfo*);

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 SQL database.

Completion state
Information function only; does not affect transaction processing.

DbiResult return values
DBIERR_NONE

DbiGetVchkDesc

Syntax
DBIResult DBIFN DbiGetVchkDesc (*hCursor*, *iValSeqNo*, *pvalDesc*);

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

DbiInit

Syntax
DBIResult DBIFN DbiInit (*pEnv*);

Description
DbiInit initializes the BDESDK 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 BDESDK with the client name.

Usage
Initializes the engine environment. Default settings can be overwritten by supplying the appropriate settings. If *pEnv* is NULL, then BDESDK 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 engine environment was initialized successfully.
DBIERR_MULTIPLEINIT	Illegal attempt to initialize the engine more than once.

See also
DbiExit

DbiInitRecord

Syntax
DBIResult DBIFN DbiInitRecord (*hCursor*, *pRecBuf*);

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 BDESDK 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

DbiInsertRecord

Syntax
DBIResult DBIFN DbiInsertRecord (*hCursor*, [*eLock*], *pRecBuf*);

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: For dBASE 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

DbiIsRecordLocked

Syntax
DBIResult DBIFN DbiIsRecordLocked (*hCursor*, *pbLocked*);

Description
DbiIsRecordLocked is used to check if the session for this cursor has the current record lock.

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 anybody.

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.
DBIERR_EOR	There is no current record.
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

DbiIsTableLocked

Syntax
DBIResult DBIFN DbiIsTableLocked (*hCursor*, *edbiLock*, *piLocks*);

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: For dBASE 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

DbiIsTableShared

Syntax
DBIResult DBIFN DbiIsTableShared (*hCursor*, *pbShared*);

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

DbiLinkDetail

Syntax
DBIResult DBIFN DbiLinkDetail (*hMstrCursor*, *hDetlCursor*, *iLnkFields*, *piMstrFields*, *piDetlFields*);

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

DbiLinkDetailToExp

Syntax
DBIResult DBIFN DbiLinkDetailToExp (*hCursorMstr*, *hCursorDetl*, *iKeyLen*, *pszMstrExp*);

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: 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 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

DbiLoadDriver

Syntax
DBIResult DBIFN DbiLoadDriver (*pszDriverType*);

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_NONE The driver has been loaded successfully.

See also
DbiOpenDriverList

DbiMakePermanent

Syntax
DBIResult DBIFN DbiMakePermanent (*hCursor*, [*pszName*], *bOverWrite*);

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

DbiModifyRecord

Syntax
DBIResult DBIFN DbiModifyRecord (*hCursor*, *pRecBuf*, *bFreeLock*);

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

DbiNativeToAnsi

Syntax
DBIResult DBIFN DbiNativeToAnsi (*pLdObj*, *pAnsiStr*, *pOemStr*, *iLen*, *pbDataLoss*);

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

DbiOpenBlob

Syntax
DBIResult DBIFN DbiOpenBlob (*hCursor*, *pRecBuf*, *iField*, *eOpenMode*);

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

Syntax
DBIResult DBIFN DbiOpenCfgInfoList (*hCfg*, *eOpenMode*, *eConfigMode*, *pszCfgPath*, *phCur*);

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.

phCur Type: phDBICur (Output)
Pointer to the client-allocated CFGDesc structure.

Usage
This function can be used to retrieve information from the configuration file about BDESDK drivers, internal buffers, and aliases by supplying a known path in *pszConfigPath*.

DbiOpenCfgInfoList accesses the same configuration file that was used when BDESDK 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 BDESDK, 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.

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.

See also
DbiInit, DbiOpenDatabaseList, DbiOpenDriverList,DbiOpenCfgInfoList

DbiOpenDatabase

Syntax

DBIResult DBIFN DbiOpenDatabase (*pszDbName*, *pszDbType*, *eOpenMode*, *eShareMode*, [*pszPassword*], *iOptFlds*, *pOptFldDesc*, *pOptParams*, *phDb*);

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.

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.

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
- INFORMIX

DbiOpenDatabaseList

Syntax
DBIResult DBIFN DbiOpenDatabaseList (*phCur*);

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.

Completion state
A cursor on a list of accessible databases is returned. The cursor is positioned before the first record.

DbiResult return values
DBIERR_NONE	The table was created successfully.
DBIERR_INVALIDHNDL	*phCur* is NULL.

See also
DbiGetDatabaseDesc

DbiOpenDriverList

Syntax
DBIResult DBIFN DbiOpenDriverList (*phCur*);

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 BDE configuration file and can be used as input to other functions. If no configuration file was available at initialization time, or if no drivers were configured, an error is returned. The table contains only one CHAR field.

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

DbiOpenFamilyList

Syntax
DBIResult DBIFN DbiOpenFamilyList (*hDb*, *pszTableName*, [*pszDriverType*], *phFmlCur*);

Description
DbiOpenFamilyList creates a table listing the family members associated with a specified table. See FMLDesc

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 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: For dBASE tables, the table can include maintained index files (.MDX files), BLOBs (.DBT 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: This function is not supported with SQL tables. With SQL 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

DbiOpenFieldList

Syntax
DBIResult DBIFN DbiOpenFieldList (*hDb*, *pszTableName*, [*pszDriverType*], *bPhyTypes*, *phCur*);

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 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 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 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 BDESDK 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 BDESDK 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.

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

DbiOpenFieldTypesList

Syntax
DBIResult DBIFN DbiOpenFieldTypesList (*pszDriverType*, [*pszTblType*], *phCur*);

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.

pszTblType Type: pCHAR (Input)
Pointer to the table type. Use *DbiOpenTableTypesList* to retrieve table type information. 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 Descriptors (FLDDesc) without creating a table. If *pszTblType* is not specified, the default table type is used.

DbiResult return values
DBIERR_NONE The table with the list of field types was created successfully.

See also
DbiGetFieldTypeDesc

DbiOpenFieldXlt

Syntax

DBIResult DBIFN DbiOpenFieldXlt (*pszSrcDriverType*, *pszSrcLangDrv*, *pfldSrc*, *pszDesDriverType*, *pszDstLangDrv*, *pfldDest*, *pbDataLoss*, *phXlt*);

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 an BDESDK 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 DATE 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.

See also

DbiTranslateField, DbiCloseFieldXlt

DbiOpenFileList

Syntax
DBIResult DBIFN DbiOpenFileList (*hDb*, [*pszWild*], *phCur*);

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
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.

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
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.

DbiOpenIndex

Syntax
DBIResult DBIFN DbiOpenIndex (*hCursor*, *pszIndexName*, *iIndexId*);

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 tables.

Usage
dBASE: This function is used to open non-production dBASE 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 a .MDX 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_ALREADYOPENED	The index is already opened, either implicitly or explicitly.
DBIERR_NOSUCHINDEX	No such index exists for the table.

See also
DbiAddIndex, DbiCloseIndex, DbiSwitchToIndex

DbiOpenIndexList

Syntax
DBIResult DBIFN DbiOpenIndexList (*hDb*, *pszTableName*, [*pszDriverType*], *phCur*);

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 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 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 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.

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

DbiOpenIndexTypesList

Syntax
DBIResult DBIFN DbiOpenIndexTypesList (*pszDriverType*, *phCur*);

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_UNKNOWNTABLETYPE	The specified driver type is unknown.
DBIERR_INVALIDHNDL	The specified handle is invalid.

See also
DbiGetIndexDesc

DbiOpenLDList

Syntax
DBIResult DBIFN DbiOpenLdList (*phCur*);

Description
DbiOpenLdList creates a table containing a list of available language drivers.

Parameters
phCur Type: phDBICur (Output)
Pointer to the cursor handle.

Completion state
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.

DbiOpenLockList

Syntax
DBIResult DBIFN DbiOpenLockList (*hCursor*, *bAllUsers*, *bAllLockTypes*, *phLocks*);

Description
DbiOpenLockList creates a 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 and SQL tables, *bAllUsers* must be set to FALSE. For dBASE, only users for the current session are listed. For SQL, only locks for the current database connection are listed.

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.

Usage
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: For dBASE 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

DbiOpenRintList

Syntax
DBIResult DBIFN DbiOpenRintList (*hDb*, *pszTableName*, [*pszDriverType*], *phChkCur*);

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
Currently, this function is supported only with Paradox tables.

Completion state
Each of the referential integrity 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 RINTDesc type definition, and used like a RINTDesc C language structure.

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

DbiOpenSecurityList

Syntax
DBIResult DBIFN DbiOpenSecurityList (*hDb*, *pszTableName*, [*pszDriverType*], *phSecCur*);

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

DbiOpenSPList

Syntax
DBIResult DBIFN DbiOpenSPList (*hdb*, *bExtended*, *bSystem*, *pszQual*, *phCur*);

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.

DbiOpenSPParamList

Syntax
DBIResult DBIFN DbiOpenSPParamList (*hdb*, *pszSPName*, *bPhyTypes*, *uOverload*, *phCur*);

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

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.

DbiOpenTable

Syntax

DBIResult DBIFN DbiOpenTable (*hDb*, *pszTableName*, [*pszDriverType*], *pszIndexName*, *pszIndexTagName*, *iIndexId*, *eOpenMode*, *eShareMode*, *exltMode*, [*bUniDirectional*], [*pOptParams*], *phCursor*);

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 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, szPARADOX, or szASCII (for importing or exporting data to/from text files; see the Usage section).

For Paradox 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 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 file used to order the records in the result set. Optional; used for dBASE 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 and SQL tables only.

Paradox: or 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.

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.

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.[MORE]

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.

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

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 dBASE and Paradox 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

Syntax
DBIResult DBIFN DbiOpenTableList (*hDb*, *bExtended*, *bSystem*, *pszWild*, *phCur*);

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)
Specifies whether to return only the standard table information, or to return extended table information as well. (The default is standard information only). [MORE]

bSystem Type: BOOL (Input)
Specifies whether to include system tables or not. SQL only. [MORE]

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.

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

bExtended

***bExtended* value**	**Type of table info returned**
TRUE	Extended
FALSE	Standard

bSystem

***bSystem* value**	**System table included?**
TRUE	Yes
FALSE	No

DbiOpenTableTypesList

Syntax
DBIResult DBIFN DbiOpenTableTypesList (*pszDriverType*, *phCur*);

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

DbiOpenUserList

Syntax
DBIResult DBIFN DbiOpenUserList (*phUsers*);

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.

DbiOpenVchkList

Syntax
DBIResult DBIFN DbiOpenVchkList (*hDb*, *pszTableName*, [*pszDriverType*], *phChkCur*);

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: This function is not supported for dBASE tables.

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

DbiPackTable

Syntax
DBIResult DBIFN DbiPackTable (*hDb*, *hCursor*, *pszTableName*, [*pszDriverType*], *bRegenIdxs*);

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: dBASE allows users to 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: This function is not valid for SQL 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

DbiPutBlob

Syntax
DBIResult DBIFN DbiPutBlob (*hCursor*, *pRecBuf*, *iField*, *iOffSet*, *iLen*, *pSrc*);

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* should be less 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*.

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

DbiPutField

Syntax
DBIResult DBIFN DbiPutField (*hCursor*, *iField*, *pRecBuf*, *pSrc*);

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 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 BDESDK 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.

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

DbiQExec

Syntax
DBIResult DBIFN DbiQExec (*hStmt*, *phCur*);

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.

SQL: For SQL, 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.

See also
DbiQPrepare, DbiQExecDirect, DbiQFree, DbiQSetParams

DbiQExecDirect

Syntax
DBIResult DBIFN DbiQExecDirect (*hDb*, *eQryLang*, *pszQuery*, *phCur*);

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 BDESDK 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 BDESDK SQL dialect and referencing tables qualified with database alias names.

QBE: For QBE language queries, the BDESDK 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.

See also
DbiQExec, DbiQFree, DbiQPrepare, DbiQSetParams

DbiQExecProcDirect

Syntax
DBIResult DBIFN DbiQExecProcDirect (*hDb*, *pszProc*, *uParamDescs*, *paParamDescs, pRecBuff, phCur*);

Description
DbiQExecProcDirect executes a stored procedure and returns a cursor to the result set, if one is generated.

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.

pRecBuff 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.

See also
DbiQPrepareProc, DbiQSetProcParams, DbiOpenSPList, DbiOpenSPParamList

DbiQFree

Syntax
DBIResult DBIFN DbiQFree (*phStmt*);

Description
DbiQFree frees the resources associated with a previously prepared 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 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.

DbiResult return values
DBIERR_NONE The query's resources were released successfully.

See also
DbiQExec, DbiQExecDirect, DbiQPrepare

DbiQInstantiateAnswer

Syntax
DBIResult DBIFN DbiQInstantiateAnswer (*hStmt*, *hCursor*, *pszAnswerName*,*pszAnswerType*, *bOverWrite*, *phCur*)

Description
DbiQInstantiateAnswer creates an ANSWER table of type *PARADOX*. 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.

pszAnswerName Type: pCHAR (Input)
Pointer to the name of the permanent table.

pszAnswerType Type: pCHAR (Input)
Pointer to the name of the permanent table.

bOverWrite Type: BOOL (Input)
If set to TRUE, overwrites the existing file.

hDstCursor Type: hDBICur (Output)
Specifies 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*.

SQL: This function is not supported by SQL drivers.

Prerequisites
Either a statement handle or a cursor handle must first be generated with DbiQPrepare.

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

Syntax
DBIResult DBIFN DbiQPrepare (*hDb*, *eQryLang*, *pszQuery*, *phStmt*);

Description
DbiQPrepare prepares a SQL or QBE query for execution, and returns a handle to a statement containing the prepared query.

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.

phStmt Type: phDBIStmt (Output)
Pointer to the statement handle.

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 BDESDK 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 BDESDK SQL dialect and referencing tables qualified with database alias names.

QBE: For QBE language queries, the BDESDK 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
DbiQExec, DbiQExecDirect, DbiQFree, DbiQSetParams, DbiQPrepareExt

DbiQPrepareExt

Syntax
DBIResult DBIFN DbiQPrepareExt (*hDb*, *eQryLang*, *pszQuery*, *upropBits, phStmt*);

Description
DbiQPrepareExt prepares a SQL or QBE query for execution, and returns a handle to a statement containing the prepared query, just as does DbiQPrepare. In addition, DbiQPrepareExt includes an option that allows the user to modify or update the resulting record set. If updateable queries are not required, DbiQPrepare should be used instead.

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.

upropBits Type: UINT16 (Input)
Detemines whether query is updateable. Two possible values:
qprepNONE Behaves like *DbiQPrepare*
qprepFORUPDATE Query is updateable.

phStmt Type: phDBIStmt (Input)
Pointer to the statement handle.

Usage
If the preparation for update succeeds, a statement handle is returned (as with DbiQPrepare). Successful execution of the statement produces a cursor handle that supports the cursor-oriented BDE write operations: DbiModifyRecord, DbiDeleteRecord, DbiInsertRecord, and so on.

The modification operations are handled through the normal Borland SQL Link optimistic locking strategy. All other existing operations work exactly as they do for a normal passthrough query.

Note that the record size of a resulting row for an updateable query is larger than for a non-updateable query. The updateable query contains information used to locate records and provide optimistic lock checking.

The updateable query feature is available with either unidirectional or bidirectional cursors for all supported servers. The resultant cursor of a statement execution is treated as a passthrough result set. A write request requires a premature read of the result set for servers that do not support multiple-statement connections (for example, SQL Server 1.x-4.x).

Note: An updateable query requires more resources than a non-updateable query. A query should be made updateable only when necessary; not as a default situation.

Prerequisites
* The query must be parseable by the local SQL parser, which is a derivative of the InterBase parser. This means that server-specific features might not be available in an updateable query. For example, if the current parser does not support the built-in server functions, they cannot be used in updateable queries. (See the *Local SQL* Help)
* SELECT DISTINCT and GROUP BY clauses are not allowed. Note that an ORDER BY clause is allowed.
* No expressions can exist in the projection list. This includes AS expressions.
* If a column is projected, it may be projected only once in the project list.
* The updateable query can reference only one table.
* The updateable query does not support self-joins.
* Passthrough queries are dead tables. This has implications that affect record modification behavior and API availability. For example, as with all passthrough, no indexes are available for switching to, a range cannot be set, and so on.
* The local SQL parser accepts BDE database alias references in queries, but these are not allowed for an updateable query. The query must be issued directly against the BDE SQL database. For example, the query:

  SELECT * FROM :SQLALIAS:mytable
  is not allowed. First, the alias SQLALIAS must be opened. Then the query (without alias) can be issued

against the database.

Completion state

BDE supports an INSERT through a cursor, but the servers do not. An insert into a dead table remains visible only as long as the cursor remains on that record. When the cursor moves off the record, the record disappears and may or may not re-appear later as the result set is read.

An unqualified table reference is assumed to be owned by the database user, as with BDE table open operations. Therefore, a query such as:

```
SELECT mycolumn FROM mytable
```

for user sqluser, results in the equivalent query:

```
SELECT mycolumn FROM sqluser.mytable.
```

While this would exhibit the same behavior as normal passthrough (if sqluser owned a table called mytable, and if the user did not own that table) the server would find it based on server name-lookup rules. For an updateable query, the name must be qualified properly to find an unowned table.

If no unique record location information is available for a table, (such as a unique index or row id) update, delete, and insert is still allowed, exactly as with a BDE table open operation. The operation succeeds provided that the server allows the operation and multiple records would not be affected when only one is targeted.

DbiResult return values

DBIERR_NONE	The query was successfully prepared for execution.
DBIERR_TABLEREADONLY	The query could not be made updateable.
DBIERR_CONNECTNOTSHARED	The SQLPASSTHRU_MODE parameter is set to NOT_SHARED. It must be set to either SHARED AUTOCOMMIT or SHARED NOAUTOCOMMIT.

See also

DbiQExec, DbiQExecDirect, DbiQFree, DbiQSetParams

DbiQPrepareProc

Syntax
DBIResult DBIFN DbiQPrepareProc (*hDb*, *pszProc*, *uParamDescs*, *paParamDescs*, *pRecBuff*, *phStmt*);

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.

pRecBuff 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 *pRecBuff* 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

DbiQSetParams

Syntax
DBIResult DBIFN DbiQSetParams (*hStmt*, *uFldDescs*, *paFldDescs*, *pRecBuf*);

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. It is supported only for passthrough SQL queries processed on SQL server tables at present.

The field descriptor array and record buffer is constructed by the client and passed to BDESDK, which uses each specified field, along with the record buffer, to locate the data and set the specified parameter. Each field may be either an BDESDK type or a driver type for the database that the query is prepared for.

Parameter markers are either ? or :name. 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.

dBASE: This function is not supported for dBASE tables.

Paradox: This function is not supported for Paradox tables.

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
DbiQExec, DbiQExecDirect, DbiQFree, DbiQPrepare

DbiQSetProcParams

Syntax
DBIResult DBIFN DbiQSetProcParams (*hStmt*, *uParamDescs, paParamDescs, pRecBuff*);

Description
DbiQSetProcParams binds parameters for a stored procedure prepared with DbiQPrepareProc.

Parameters
hStmt Type: phDBIStmt (Output)
Specifies the returned statement handle.

uParamDescs Type: UINT16 (Input)
Specifies the number of parameter descriptors.

paParamDescs Type: pSPParamDesc (Input)
Pointer to the array of parameter descriptors.

pRecBuff 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 *pRecBuff*. If the output parameter value is NULL or TRUNCATED, then *indNULL* or *indTRUNC* is placed in the *iNulloffset* of the client-supplied *pRecBuff*. 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

DbiReadBlock

Syntax
DBIResult DBIFN DbiReadBlock (*hCursor*, *piRecords*, *pBuf*);

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.

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

DbiRegenIndex

Syntax

DBIResult DBIFN DbiRegenIndex (*hDb*, [*hCursor*], [*pszTableName*], [*pszDriverType*], *pszIndexName*, *pszIndexTagName*, *iIndexId*);

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 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 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 file. Used for dBASE tables only. This parameter is ignored if the index given by *pszIndexName* is not a .MDX 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: A SQL index cannot be regenerated.

dBASE: DbiRegenIndex is normally used to update a non-maintained dBASE 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, the engine 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

DbiRegenIndexes

Syntax
DBIResult DBIFN DbiRegenIndexes (*hCursor*);

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: All open indexes are regenerated.

Paradox: All maintained and non-maintained indexes are regenerated.

SQL: SQL 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

DbiRegisterCallBack

Syntax
DBIResult DBIFN DbiRegisterCallBack (*hCursor*, *ecbType*, *iClientData*, *iCbBufLen*, *pCbBuf*, *pfCb*);

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, or cbDBASELOGIN. (See Usage below.)

iClientData Type: UINT32 (Input)
Pass-through 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 engine function before completing an operation or to return information to the client. DbiRegisterCallBack allows the client to instruct the database engine about what further actions should be taken by the engine upon the occurrence of an event. The engine calls the client-registered function when the pertinent event occurs, and the client responds to the callback by telling the engine 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 the engine 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 the engine.

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:

- cbgenprogress
- cbRESTRUCTURE
- cbBATCHRESULT
- cbTABLECHANGED
- cbINPUTREQ
- cbDBASELOGIN

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

DbiRelPersistTableLock

Syntax
DBIResult DBIFN DbiRelPersistTableLock (*hDb*, *pszTableName*, [*pszDriverType*]);

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: This function is not supported with dBASE 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

DbiRelRecordLock

Syntax
DBIResult DBIFN DbiRelRecordLock (*hCursor*, *bAll*);

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

DbiRelTableLock

Syntax
DBIResult DBIFN DbiRelTableLock (*hCursor*, *bAll*, *eLockType*);

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 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: 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

DbiRenameTable

Syntax
DBIResult DBIFN DbiRenameTable (*hDb*, *pszOldName*, [*pszDriverType*], *pszNewName*);

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 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 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: szPARADOX or szDBASE.

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: The following files are renamed:

- The table (.DBF extension)
- BLOB files (.DBT extension)
- The production index (.MDX extension)

SQL: All indexes are renamed with the table. 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

DbiResetRange

Syntax
DBIResult DBIFN DbiResetRange (*hCursor*);

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

DbiSaveChanges

Syntax
DBIResult DBIFN DbiSaveChanges (*hCursor*);

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

DbiSetCurrSession

Syntax
DBIResult DBIFN DbiSetCurrSession (*hSes*);

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 4.
DBIERR_INVALIDSESHANDLE	The specified session handle is invalid.

See also
DbiGetCurrSession, DbiStartSession, DbiCloseSession, DbiGetSysInfo, DbiGetSesInfo

DbiSetDateFormat

Syntax
DBIResult DBIFN DbiSetDateFormat (*pfmtDate*);

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 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

DbiSetDirectory

Syntax
DBIResult DBIFN DbiSetDirectory (*hDb*, *pszDir*);

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: DbiSetDirectory is not applicable to SQL 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

DbiSetFieldMap

Syntax
DBIResult DBIFN DbiSetFieldMap (*hCur*, *iFields*, *pFldDesc*);

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 dBASE and Paradox, 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

DbiSetLockRetry

Syntax
DBIResult DBIFN DbiSetLockRetry (*iWait*);

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 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
- DbiAcqPersistTableLock

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 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

DbiSetNumberFormat

Syntax
DBIResult DBIFN DbiSetNumberFormat (*pfmtNumber*);

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

DbiSetPrivateDir

Syntax
DBIResult DBIFN DbiSetPrivateDir (*pszDir*);

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.

Prerequisites
The directory must be available for exclusive access. No other BDESDK 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

DbiSetProp

Syntax
DBIResult DBIFN DbiSetProp (*hObj*, *iProp*, *iPropValue*);

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.

See also
DbiOpenTable, DbiGetProp

DbiSetRange

Syntax
DBIResult DBIFN DbiSetRange (*hCursor*, *bKeyItself*, [*iFields1*], [*iLen1*], [*pKey1*], *bKey1Incl*, *iFields2*, *iLen2*, [*pKey2*], *bKey2Incl*);

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

DbiSetTimeFormat

Syntax
DBIResult DBIFN DbiSetTimeFormat (*pfmtTime*);

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

DbiSetToBegin

Syntax
DBIResult DBIFN DbiSetToBegin (*hCursor*);

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. (DbiGetRecordreturns 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

DbiSetToBookMark

Syntax
DBIResult DBIFN DbiSetToBookMark (*hCur*, *pBookMark*);

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

DbiSetToCursor

Syntax
DBIResult DBIFN DbiSetToCursor (*hDest*, *hSrc*);

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.

See also
DbiGetBookMark, DbiSetToBookMark, DbiCloneCursor, DbiOpenTable

DbiSetToEnd

Syntax
DBIResult DBIFN DbiSetToEnd (*hCursor*);

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

DbiSetToKey

Syntax
DBIResult DBIFN DbiSetToKey (*hCursor*, *eSearchCond*, *bDirectKey*, [*iFields*], [*iLen*], *pBuf*);

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.

The key can be specified 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
The search always results in the cursor being positioned on the crack just prior to 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

DbiSetToRecordNo

Syntax
DBIResult DBIFN DbiSetToRecordNo (*hCursor*, *iRecNo*);

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 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

DbiSetToSeqNo

Syntax
DBIResult DBIFN DbiSetToSeqNo (*hCursor*, *iSeqNo*);

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

DbiSortTable

Syntax

DBIResult DBIFN DbiSortTable (*hDb*, *pszTableName*, *pszDriverType*, *hSrcCur*, *pszSortedName*, *phSortedCur*, *hDstCur*, *iSortFields*, *piFieldNum*, [*pbCaseInsensitive*], [*pSortOrder*], [**ppfsortFn*], *bRemoveDups*, [*hDuplicatesCur*], [*plRecsSort*]);

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

DbiStartSession

Syntax
DBIResult DBIFN DbiStartSession ([*pszName*], *phSes*, [*pNetDir*]);

Description
DbiStartSession starts a new session for the client application.

Parameters
pszName Type: pCHAR (Input)
Pointer to the session name. 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. Optional.

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

DbiSwitchToIndex

Syntax
DBIResult DBIFN DbiSwitchToIndex (*phCursor*, *pszIndexName*, *pszTagName*, *iIndexId*, *bCurrRec*);

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 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 and SQL 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. 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

DbiTimeDecode

Syntax
DBIResult DBIFN DbiTimeDecode (*timeT*, *piHour*, *piMin*, *piMilSec*);

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

DbiTimeEncode

Syntax
DBIResult DBIFN DbiTimeEncode (*iHour*, *iMin*, *iMilSec*, *ptimeT*);

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

DbiTimeStampDecode

Syntax
DBIResult DBIFN DbiTimeStampDecode (*tsTS*, *pdateD*, *ptimeT*);

Description
DbiTimeStampDecode extracts separate encoded DATE and TIME components from the TIMESTAMP.

Parameters
tsTS Type: TIMESTAMP (Input)
Specifies the encoded DATETIME timestamp.

pdateD Type: pDATE (Output)
Pointer to the client variable that receives the encoded DATE 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

DbiTimeStampEncode

Syntax
DBIResult DBIFN DbiTimeStampEncode (*dateD*, *timeT*, *ptsTS*);

Description
DbiTimeStampEncode encodes the encoded DATE and encoded TIME into a TIMESTAMP.

Parameters
dateD Type: DATE (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

DbiTranslateField

Syntax
DBIResult DBIFN DbiTranslateField (*hXlt*, *pSrc*, *pDest*);

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 (Input)
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. The translation object must be built using an BDESDK-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.

See also
DbiOpenFieldXlt, DbiCloseFieldXlt

DbiTranslateRecordStructure

Syntax
DBIResult DBIFN DbiTranslateRecordStructure (*pszSrcDriverType*, *iFlds*, *pfldsSrc*, *pszDstDriverType*, *pszLangDriver*, *pfldsDst*);

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 physical.

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 physical.

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.

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.

DbiTruncateBlob

Syntax
DBIResult DBIFN DbiTruncateBlob (*hCursor*, *pRecBuf*, *iField*, *iLen*);

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: 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

DbiUndeleteRecord

Syntax
DBIResult DBIFN DbiUndeleteRecord (*hCursor*);

Description
DbiUndeleteRecord undeletes a dBASE record that has been marked for deletion (a soft delete).

Parameters
hCursor Type: hDBICur (Input)
Specifies the dBASE cursor handle.

Usage
dBASE: This function is supported with dBASE tables only.

Paradox: This function is not supported with Paradox 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 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 tables.
DBIERR_NOCURRREC	The cursor is not positioned on a valid record.

See also
DbiDeleteRecord, DbiPackTable

DbiUnlinkDetail

Syntax
DBIResult DBIFN DbiUnlinkDetail (*hDetlCursor*);

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

DbiUseIdleTime

Syntax
DBIResult DBIFN DbiUseIdleTime (VOID);

Description
DbiUseIdleTime allows the BDE to accomplish background tasks during times when the client application is idle.

Usage
An interactive BDE client application typically operates by interpreting messages from the Windows Message Queue™. If this queue is found to be empty (such as between keystrokes), the client application can call this function to allow BDESDK to accomplish background tasks. This call returns quickly, so the user does not experience any delay when typing.

DbiUseIdleTime is primarily used for writing dirty buffers to disk. This makes the client application more reliable during power failure or machine lockups in another application (recoverable by CtrlAltDelete). Likewise, client applications can set up a timer and call this function periodically when the timer event is generated.

Completion state
This function writes one dirty buffer to disk, or returns if there are none.

DbiResult return values
DBIERR_NONE This function always returns DBIERR_NONE.

DbiVerifyField

Syntax
DBIResult DBIFN DbiVerifyField (*hCursor*, *iField*, *pSrc*, [*pbBlank*]);

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: For dBASE 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

DbiWriteBlock

Syntax
DBIResult DBIFN DbiWriteBlock (*hCursor*, *piRecords*, *pBuf*);

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*.

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

Data Structures

The data structures used in BDESDK are listed in the following table. Two charts of data translations are available at the end of the table.

Structure	Description
BATTblDesc	Batch table definition
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
DBIEnv	Defines the BDESDK environment
DBIErrInfo	Provides error information
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 standard procedure
SPParamDesc	Describes the parameters to a standard procedure
SYSConfig	Provides basic system configuration information
SYSInfo	Provides BDESDK system status
SYSVersion	Provides BDESDK 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

Data Type Translations Chart showing how data is translated between different database formats.

Logical Types and Driver-specific Physical Types
Chart showing relationships between logical and physical types.

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)

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

canOPType: 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
canLIKE	Case-insensitive partial field search
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

Arithmetic operators	
canMINUS	Unary; minus
canADD	Binary; addition
canSUB	Binary; subtraction

canMUL	Binary; multiplication
canDIV	Binary; division
canMOD	Binary; modulo division
canREM	Binary; remainder of division

Aggregate type operators

canSUM	Binary, accumulate sum of
canCOUNT	Binary, accumulate count of
canMIN	Binary, find minimum of
canMAX	Binary, find maximum of
canAVG	Binary, find average of

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 BDESDK 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	Not currently used.
szUserName	DBINAME	Not currently used.
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 only).
printDesc	pRINTDesc	Specifies the array of referential integrity specifications (Paradox 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; used if the record structure is changing. Must be crADD if a field is added, crMODIFY if a field is modified, or crCOPY if a 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 BDESDK 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). < 0 (-1, -2. . .): None (SQL)
bSoftDeletes	BOOL	This value is set to TRUE if this cursor supports soft deletes (dBASE 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 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	Not currently used.
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
dbiREADWRITE	Read and write (default)
dbiREADONLY	Read-only

eShareMode

The following share modes are valid:

Share Mode	Description
dbiOPENSHARED	Open shared (default)
dbiOPENEXCL	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
xltNONE	No translation; use physical types
xltFIELD	Field-level translation; use logical types

eprvRights

The table-level rights supported are:

Privilege	Description
prvNONE	No privileges
prvREADONLY	Read-only table or field
prvMODIFY	Read and modify fields
prvINSERT	Insert all of above
prvINSDEL	Delete all of above
prvFULL	Full rights
prvUNKNOWN	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.

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 pass-through 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, 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	DATE	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 following properties:

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 ordinal number of the field (1 to n). For Paradox, it is the invariant field ID.
szName	DBINAME	Specifies the name of the 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 an 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	NT16	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.

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.
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	f 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.

```
typedef FMLDesc far *pFMLDesc;
#define DBIOEM_CP                1        // (dos)
#define DBIANSI_CP               2        // (win)
#define DBIOS2_CP                3
/* UNIX etc. */
#define DBISUNOS_CP              4
#define DBIVMS_CP                5
#define DBIHPUX_CP               6
#define DBIULTRIX_CP             7
#define DBIAIX_CP                8
#define DBIAUX_CP                9
#define DBIXENIX_CP              10
#define DBIMAC_CP                11
#define DBINEXT_CP               12
```

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.

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	NT8	If TRUE, on input add 1900 to year.
bMonthLeadingZero	INT8	If TRUE, the month is displayed with a leading zero.
bDayLeadingZero	NT8	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 a.m. time (only for 12-hour time).
szPmString[6]	CHAR	Specifies the string to use for designating p.m. time (only for 12-hour time).
bSeconds	INT8	If TRUE, show seconds.
bMilSeconds	INT8	If TRUE, show milli-seconds.

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
dBASE	.NDX style: *szName* alone identifies the index.
	.MDX 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 only.
szFormat	DBINAME	Currently, for information only. Describes the physical index format type (for example, BTREE or HASH).
bPrimary	BOOL	TRUE, if the key is primary.
bUnique	BOOL	TRUE, if the key is unique.
bDescending	BOOL	TRUE, if the key is descending.
bMaintained	BOOL	TRUE, if the key is maintained.
bSubset	BOOL	TRUE, if the index is a subset index. Supported for dBASE only.
bExpIdx	BOOL	TRUE, if the index is an expression index. Supported for dBASE only.
iCost	UINT16	Not currently used.
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	BOOL	Not specified while index is created; TRUE, if the index is out-of-date.
iKeyExpType	UINT16	Specifies the type of the key expression (dBASE 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 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 only). This field is used only if bSubset = TRUE. The expression is stated as a dBASE expression.
bCaseInsensitive	BOOL	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*

szName

The following table describes how to name Paradox indexes:

Index ID Param	Non-composite index	Composite index
szName	Same as the field name	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

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 only).
bKeyExpr	BOOL	If TRUE, the key can be an expression (dBASE 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	Specifies the platform type to which the driver's character set corresponds. For example, DOS or Windows.
AlternateCpPlatform	UINT16	Specifies the alternate platform. For internal use only.

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 BDESDK session number, if BDESDK 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).

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 only).
bRecChanged	BOOL	Not currently used.
bSeqNumChanged	BOOL	Not currently used.
bDeleteFlag	BOOL	Specifies if the record is deleted. Applicable only when soft delete is supported (dBASE only).

RINTDesc (referential integrity)

The RINTDesc structure describes the referential integrity options for a table (currently Paradox 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	Specifies the field numbers that make up this referential integrity constraint in this table.
aiOthTabFld	DBIKEY	Specifies the number of fields in the other table.

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	DATE	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-BDESDK 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 BDESDK 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 BDESDK system version information, using the following fields:

Field	Type	Description
iVersion	UINT16	Specifies the engine version.
iIntfLevel	UINT16	Specifies the client interface level.
dateVer	DATE	Specifies the version date.
timeVer	TIME	Specifies the version time.

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	DATE	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).

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, BDESDK 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 and SQL tables only), using the following fields (*bRequired* is the only option supported by the SQL drivers):

Field	Type	Description
iFldNum	UINT16	Specifies the field number (1 to n).
bRequired	BOOL	Specifies wether 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.
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

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 form 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 by the Paradox driver only.

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 backto 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;
```

SQL-specific Locking Behavior

SQL deals with record locking differently than Paradox or dBASE. 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.

All record locking on SQL tables is optimistic. An optimistic record lock is basically a notification tool; it does not prevent another user from modifying the locked record. The operation is said to be optimistic because it assumes that no other client will change the record. However, if the record has been changed, and the client tries to modify the record (DbiModifyRecord), the client is notified that the requested operation cannot be performed because someone else has modified the data. The client can then inspect the new data, and decide whether or not to submit the changes. 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 users's modifications. At the same time, the client is protected from inadvertently changing data that has never been inspected.

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.

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 might need to widen this Help window to display the full width of the chart.)

From Paradox	To dBASE	To Oracle	To Sybase	To InterBase	To Informix
Alpha	Character	Character	VarChar	Varying	Character
Number	Float {20.4}	Number	Float	Double	Float
Money	Number {20.4}	Number	Money	Double	Money {16.2}
Date	Date	Date	DateTime	Date	Date
Short	Number {6.0}	Number	SmallInt	Short	SmallInt
Memo	Memo	Long	Text	Blob/1	Text
Binary	Memo	LongRaw	Image	Blob	Byte
Formatted memo	Memo	LongRaw	Image	Blob	Byte
OLE	OLE	LongRaw	Image	Blob	Byte
Graphic	Binary	LongRaw	Image	Blob	Byte
Long	Number {11.0}	Number	Int	Long	Integer
Time	Character {>8}	Character {>8}	Character {>8}	Character {>8}	Character {>8}
DateTime	Character {>8}	Date	DateTime	Date	DateTime
Bool	Bool	Character {1}	Bit	Character {1}	Character
AutoInc	Number{11.0}	Number	Int	Long	Integer
Bytes	Memo	LongRaw	Image	Blob	Byte
BCD	N/A	N/A	N/A	N/A	N/A

From dBASE	To Paradox	To Oracle	To Sybase	To InterBase	To Informix
Character	Alpha	Character	VarChar	Varying	Character
Number iUnits2=0 && iUnits1<5	Short	Number	SmallInt	Short	SmallInt
others	Number	Number	Float	Double	Float
Float	Number	Number	Float	Double	Float
Date	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	Blob	Byte (temp tables only)

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	Float {20.4}	Float	Double	Float
Long	Memo	Memo	Text	Blob/1	Text
LongRaw	Binary	Memo	Image	Blob	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	Float {20.4}	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	Date	DAte	Date	DateTime
TimeStamp	Binary	Memo	Raw	Varying	VarChar
Float4	Number	Number	Number	Double	Float
Money4	Money	Number {20.4}	Number	Double	Money {16.2}
DateTime4	DateTime	Date	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
Date	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

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.)

From Paradox physical type	To BDESDK logical type	To dBASE physical type
fldPDXCHAR	fldZSTRING	fldDBCHAR
fldPDXNUM	fldFLOAT	fldDBFLOAT {20.4}
fldPDXMONEY	fldFLOAT/fldstMONEY	fldDBNUM {20.4}
fldPDXDATE	fldDATE	fldDATE
fldPDXSHORT	fldINT16	fldDBNUM {6.0}
fldPDXMEMO	fldBLOB/fldstMEMO	fldDBMEMO
fldPDXBINARYBLOB	fldBLOB/fldstBINARY	fldDBMEMO
fldPDXFMTMEMO	fldBLOB/fldstFMTMEMO	fldDBMEMO
fldPDXOLEBLOB	fldBLOB/fldstOLEOBJ	fldDBOLEBLOB
fldPDXGRAPHIC	fldBLOB/fldstGRAPHIC	fldDBBINARY
fldPDXBLOB	fldPDXMEMO	fldDBMEMO
Paradox level 5 data types:		
fldPDXLONG	fldINT32	fldDBNUM {11.0}
fldPDXTIME	fldTIME	fldDBCHAR {>8}
fldPDXDATETIME	fldTIMESTAMP	fldDBCHAR {30}
fldPDXBOOL	fldBOOL	fldDBBOOL
fldPDXAUTOINC	fldINT32	fldDBNUM {11.0}
fldPDXBYTES	fldBYTES	fldDBMEMO
fldPDXBCD	fldBCD	fldDBCHAR

From dBASE physical type	To BDESDK logical type	To Paradox physical type
fldDBCHAR	fldZSTRING	fldPDXCHAR
fldDBNUM	if (iUnits2=0 && iUnits1<5) fldINT16	fldPDXSHORT
	else fldFLOAT	fldPDXNUM
fldDBMEMO	fldBLOB	fldPDXMEMO
fldDBBOOL	fldBOOL	fldPDXBOOL
fldDBDATE	fldDATE	fldPDXDATE
fldDBFLOAT	fldFLOAT	fldPDXNUM
fldDBLOCK	fldLOCKINFO	fldPDXCHAR {24}
fldDBBINARY	fldBLOB/fldstTYPEDBINARY	fldPDXBINARYBLOB
fldDBOLEBLOB	fldBLOB/fldstDBSOLEOBJ	fldPDXOLEBLOB
fldDBBYTES	fldBYTES	fldPDXBYTES (only for temp tables)

From Oracle physical type	To BDESDK logical type	To Paradox physical type	To dBASE physical type
fldORACHAR	fldZSTRING	fldPDXCHAR	fldDBCHAR
fldORARAW	fldVARBYTES	fldPDXBINARYBLOB	fldDBMEMO
fldORADATE	fldTIMESTAMP	fldPDXDATETIME	fldDBCHAR
fldORANUMBER	fldFLOAT	fldPDXNUM	fldDBFLOAT {20.4}
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	fldDBFLOAT {20.4}

From Sybase physical type	To BDESDK logical type	To Paradox physical type	To dBASE physical type
fldSYBCHAR	fldZSTRING	fldPDXCHAR	fldDBCHAR
fldSYBVARCHAR	fldZSTRING	fldPDXCHAR	fldDBCHAR
fldSYBINT	fldINT32	fldPDXLONG	fldDBNUM {11.0}
fldSYBSMALLINT	fldINT16	fldPDXSHORT	fldDBNUM {6.0}
fldSYBTINYINT	fldINT16	fldPDXSHORT	fldDBNUM {6.0}
fldSYBFLOAT	fldFLOAT	fldPDXNUM	fldDBFLOAT {20.4}
fldSYBMONEY	fldFLOAT/fldstMONEY	fldPDXMONEY	fldDBNUM {20.4}
fldSYBTEXT	fldBLOB/fldstMEMO	fldPDXMEMO	fldDBMEMO
fldSYBBINARY	fldVARBYTES	fldPDXBINARYBLOB	fldDBMEMO
fldSYBVARBINARY	fldVARBYTES	fldPDXBINARYBLOB	fldDBMEMO
fldSYBIMAGE	fldBLOB/fldstBINARY	fldPDXBINARYBLOB	fldDBMEMO
fldSYBBIT	fldBOOL	fldPDXBOOL	fldDBBOOL
fldSYBDATETIME	fldTIMESTAMP	fldPDXDATETIME	fldDBDATE
fldSYBTIMESTAMP	fldVARBYTES	fldPDXBINARYBLOB	fldDBMEMO
fldSYBFLOAT4	fldFLOAT	fldPDXNUM	fldDBFLOAT {20.4}
fldSYBMONEY4	fldFLOAT/fldstMONEY	fldPDXMONEY	fldDBNUM {20.4}
fldSYBDATETIME4	fldTIMESTAMP	fldPDXDATETIME	fldDBDBDATE

From InterBase physical type	To BDESDK logical type	To Paradox physical type	To dBASE physical type
fldIBSHORT	fldINT16	fldPDXSHORT	fldDBNUM {6.0}
fldIBLONG	fldINT32	fldPDXLONG	fldDBNUM {11.0}
fldIBFLOAT	fldFLOAT	fldPDXNUM	fldDBFLOAT {20.4}
fldIBDOUBLE	fldFLOAT	fldPDXNUM	fldDBFLOAT {20.4}
fldIBCHAR			
iUnits 1 <=255	fldZSTRING	fldPDXCHAR	fldDBCHAR
iUnits1 > 255	fldBLOB/fldstMEMO	fldPDXMEMO	fldDBMEMO
fldIBVARYING			
iUnits1 <= 255	fldSTRING	fldPDXCHAR	fldDBCHAR
iUnits1 >255	fldBLOB/fldstMEMO	fldPDXMEMO	fldDBMEMO
fldIBDATE	fldTIMESTAMP	fldPDXDATETIME	fldDBDATE
fldIBBLOB	fldBLOB	fldPDXBINARYBLOB	fldDBMEMO
fldIBBLOB/1	fldBLOB/fldstMEMO	fldPDXMEMO	fldDBMEMO

From Informix physical type	To BDESDK logical type	To Paradox physical type	To dBASE physical type
fldINFCHAR			
iUnits1 <=255	fldZSTRING	fldPDXCHAR	fldDBCHAR
iUnits1 > 255	fldBLOB/fldstMEMO	fldPDXMEMO	fldDBMEMO
fldINFSMALLINT	fldINT16	fldPDXSHORT	fldDBNUM {6.0}
fldINFINTEGER	fldINT32	fldPDXLONG	fldDBNUM {11.0}
fldINFSMALLFLOAT	fldFLOAT	fldPDXNUM	fldDBFLOAT {20.4}
fldINFFLOAT	fldFLOAT	fldPDXNUM	fldDBFLOAT {20.4}
fldINFMONEY	fldFLOAT/fldstMONEY	fldPDXMONEY	fldDBNUM {20.4}
fldINFDECIMAL	fldFLOAT	fldPDXNUM	fldDBFLOAT {20.4}
fldINFDATE	fldDATE	fldPDXDATE	fldDBDATE

fldINFDATETIME	fldTIMESTAMP	fldPDXDATETIME	fldDBDATE
fldINFINTERVAL	fldZSTRING	fldPDXCHAR	fldDBCHAR
fldINFSERIAL	fldINT32	fldPDXLONG	fldDBNUM {11.0}
fldINFBYTE	fldBLOB/fldstBINARY	fldPDXBINARYBLOB	fldDBMEMO
fldINFTEXT	fldBLOB/fldstMEMO	fldPDXMEMO	fldDBMEMO
fldINFVARCHAR	fldZSTRING	fldPDXCHAR	fldDBCHAR

International Compatibility

This chapter describes considerations that may be encountered for international applications. The following topics are discussed:

- 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, since 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 driver name	Short driver name	Character set	Collation sequence
Borland ENU Latin-1	BLLT1US0	ISO8859.1(ANSI)	Binary
dBASE FRA cp437	DB437FR0	DOS CODE PAGE 437	dBASE French
dBASE FIN cp437	DB437FI0	DOS CODE PAGE 437	dBASE Finnish
dBASE ENU cp437	DB437US0	DOS CODE PAGE 437	dBASE English/US
dBASE NOR cp865	DB865NO0	DOS CODE PAGE 865	dBASE Norwegian
dBASE SVE cp437	DB437SV0	DOS CODE PAGE 437	dBASE Swedish
dBASE SVE cp850	DB850SV1	DOS CODE PAGE 850	dBASE Swedish850
dBASE ESP cp437	DB437ES1	DOS CODE PAGE 437	dBASE Spanish
dBASE NLD cp437	DB437NL0	DOS CODE PAGE 437	dBASE Dutch
dBASE ESP cp850	DB850ES1	DOS CODE PAGE 850	dBASE Spanish850
dBASE ENG cp437	DB437UK0	DOS CODE PAGE 437	dBASE English/UK
dBASE ENU cp850	DB850US0	DOS CODE PAGE 850	dBASE English/US
dBASE FRC cp863	DB863CF1	DOS CODE PAGE 863	dBASE French Canadian
dBASE ENG cp850	DB850UK0	DOS CODE PAGE 850	dBASE English850/UK
dBASE ITA cp850	DB850IT1	DOS CODE PAGE 850	dBASE Italian850
dBASE DEU cp850	BD850DE0	DOS CODE PAGE 850	dBASE German850
dBASE FRA cp850	DB850FR0	DOS CODE PAGE 850	dBASE French850
dBASE ITA cp437	DB437IT0	DOS CODE PAGE 437	dBASE Italian
dBASE NLD cp850	DB850NL0	DOS CODE PAGE 850	dBASE Dutch
dBASE FRC cp850	DB850CF0	DOS CODE PAGE 850	dBASE French Canadian850
dBASE DAN cp865	DB865DA0	DOS CODE PAGE 865	dBASE Danish
dBASE DEU cp437	DB437DE0	DOS CODE PAGE 437	dBASE German
Oracle SQL WE850	ORAWE850	DOS CODE PAGE 850	ORACLE multi-lingual Western European sort order
Paradox ascii'	ascii	DOS CODE PAGE 437	Binary
Paradox intl'	intl	DOS CODE PAGE 437	Multilingual Western European
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)
Paradox swedfin'	swedfin	DOS CODE PAGE 437	Paradox 'swedfin'
Paradox ANSI INTL	ANSIINTL	ISO8859.1 (ANSI)	Compatible with Paradox 'intl'
Paradox ESP 437	SPANISH	DOS CODE PAGE 437	Spanish
Paradox ISL 861	iceland	DOS CODE PAGE 861	Icelandic
Pdox ANSI INTL850	ANSII850	ISO8859.1 (ANSI)	Compatible with Paradox 'intl' 850
Pdox ANSI NORDAN4	ANSINOR4	ISO8859.1 (ANSI)	Compatible with Paradox 'nordan40'
Pdox ANSI SWEDFIN	ANSISWFN	ISO8859.1 (ANSI)	Compatible with Paradox 'swedfin'
Pdox ESP ANSI	ANSISPAN	ISO8859.1 (ANSI)	Compatible with Paradox ESP437
SYBASE SQL Dic437	SYDC437	DOS CODE PAGE 437	SYBASE dict. with case-sensitivity
SYBASE SQL Dic850	SYDC850	DOS CODE PAGE 850	SYBASE dict. with case-sensitivity
SQL Link ROMAN8	BLROM800	ROMAN8	Binary

Default language driver settings are defined in the BDE configuration file (IDAPI.CFG). You can change these defaults using the BDE Configuration Utility. 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 or dBASE table is created, the default langauge 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 BDESDK 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.
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 BDE configuration file (IDAPI.CFG). (See Date, Time, and Number Pages.) 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 BDESDK 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
Gretel Bailey
Tracy Blank
Clement Chan
Bruce Chang
Ernest Chen
Laxman Chinnakotla
Cliff Cormier
Elvi Dalgaard
Mark Edington
Dan Eris
Freda Fine
Anne Fletcher
Rajamohan Gandhasri
Kurt Hansen
Conrad Herrmann
Renae Hester
Sarah Huang
Shabbir Khan
Gopal Kirsur
Klaus Krull
Michael Linetsky
Lisa Loud
Britta Matthews
Rick Nadler
Sunil Nair
Dung Nguyen
Inna Noten
Chris Ohlsen
Don Phan
David Raccah
Kris Ramberg
Eric Roth
Pandu Rudraraju
Richard Scannell
Max Slimmer
Bert Speelpenning
Karen Tanaka
Devendra Vamathevan
Narayanan Vijaykumar
Ken Vodicka
Nimish Vora
Jack Zoken