Go To

DataDirect Developer's Toolkit Programmer's Guide

Table of Contents

Go To

List of Tables

Go To

Using this Manual

This manual is organized to make DTK easy to learn and to provide a convenient reference guide. You don't have to read the entire manual to begin using the product.

Part I contains guidelines and specific information about using DTK to create applications. Each chapter in Part I describes a specific set of tasks and related concepts, gives sample applications to illustrate their usage, and lists the related DTK functions. Chapter 1 describes a sample DTK application that you can run to better familiarize yourself with DTK's capabilities.

Part II is a complete, alphabetical reference to the DTK functions. It begins by describing some of the parameter conventions employed in the functions.

The appendixes in Part III contain information tailored to specific tasks and users. Appendix A describes the data conversion functions that DTK provides. Appendix B contains information specific to using DTK with Microsoft Visual Basic®. Appendix C describes considerations and techniques for coding DTK applications for connection to database systems that support only one statement per connection. Appendix D lists DTK error message codes and their corresponding text. Appendix E describes compatibility issues of interest to users of version QELIB 1.0. Appendix F describes the contents and format of the QELIB.INI file.

Information on the database drivers supplied with DTK and the SQL language is provided in the INTERSOLV driver reference manual that accompanies this product.

Conventions Used in This Manual

This manual uses various conventions to aid in its usability. The typography, terminology, and callouts to environment-specific information used are intended to make this manual easy to use, regardless of the operating environment you are using. The following sections describe these conventions.

Typography

This manual uses various typefaces, fonts, and characters to indicate certain types of information, as follows:

Convention	Explanation
italics	Used to identify parameters to DTK functions, to introduce new terms that you may not be familiar with, and occasionally for emphasis.
bold	Used to emphasize important information.
bold italics	Used to identify parameters in syntax descriptions of DTK function calls. Each such parameter is preceded in the description by its declared data type.
`monospace`	Code examples or results that you receive.
UPPERCASE	Indicates the name of a file. For operating environments that use case-sensitive filenames, the correct capitalization is used in information specific to those environments.

Terminology

This manual uses the following terminology:

- The term *ODBC.INI* refers to the ODBC.INI file format as defined by the Microsoft ODBC specification.
- The suffix *.DLL* refers to a dynamic link library file. Your operating system may use shared object or shared library files instead.

Environment-Specific Information

This manual shows dialog boxes that are specific to Windows. If you are using DTK on Windows 95, Windows NT, OS/2, Macintosh, or UNIX, the dialog boxes you see may differ slightly from the Windows version.

Go To

Part 1: Using DTK

1 Getting Started

This chapter contains the following information to help you get started using the DataDirect Developer's Toolkit:

- "About the DataDirect Developer's Toolkit,"next
- "Installing DTK" on page 3
- "Building a DTK Application" on page 4
- "Solving Problems and Getting Technical Support" on page 15

About the DataDirect Developer's Toolkit

Whether you are working with spreadsheets, word processors, graphics packages, or development tools, it is often important to incorporate information from a database into your application. Unfortunately, most products provide little or no support for direct access to databases. The DataDirect Developer's Toolkit provides a powerful, flexible way for you to add database access to your applications. DTK contains functions that allow you to read, insert, update, and delete records from databases.

Many Microsoft® Windows™ or IBM® OS/2® products include a macro or script language that lets you customize the product or build your own applications. These macro languages usually provide a way for you to call functions in Dynamic Link Libraries (DLLs). DTK is packaged as a set of DLLs, so you can call DTK functions from the macro and script languages of any application that supports DLLs. Or, you can call DTK functions from programming languages such as C.

Using DTK, you can

- Use a graphics product to build a pie chart from data stored in Oracle.
- Use a spreadsheet product to analyze data stored in SQL Server.
- Use a word processor to write a memo containing last month's sales figures stored in DB2.
- Use a development tool to build a data entry screen that stores data in Paradox.

For a complete list of the databases supported by DTK, refer to the release notes that accompany the product.

All database operations are performed by sending Structured Query Language (SQL) statements to the API. SQL is a standard database language supported by many database systems. For database systems that do not support SQL, DTK operates directly on the database files to execute the SQL statements.

The advantage of supporting SQL for all database systems is that you can build one application that can access data from any database DTK supports. You don't have to rewrite your application for each database system. You can test your application on a local database system and later run it on a different, server-based database system.

Distributing INTERSOLV's Database Drivers

With your DTK applications, you are allowed to distribute, royalty free, the files your application needs to run. These are the DTK's:

- API library (for example, QELIB.DLL in Windows)
- Graphical Interface library (for example, QEGUInn.DLL in Windows)

- SQL-support library (for example, QESQLnn.DLL in Windows)
- Utilities library (for example, QEUTLnn.DLL in Windows)

For Windows, Windows 95, and Windows NT, you can also distribute, royalty free, the Query Builder library (for example, QEQRYnn.DLL in Windows) and a context-sensitive help file that is called by the Query Builder (QRYBLDR.HLP)

Please see the README file for your platform for a specific list of the files you can distribute royalty-free.

You *cannot* distribute INTERSOLV ODBC drivers included with the DataDirect Developer's Toolkit. ODBC Drivers are included only for development and testing purposes. For distributing your application and ODBC drivers, you can follow either of the following procedures:

- Purchase DataDirect ODBC Drivers from INTERSOLV for distribution with your developed application
- Distribute your application without drivers royalty free and require your customers to purchase the drivers

Either you or your customer can purchase a single driver or multiple copies of a single driver, as well as the entire DataDirect ODBC Pack from INTERSOLV. If would like to obtain a distribution license, please call 1-800-876-3101 for more information.

Installing DTK

Refer to the installation instructions that accompany this version of DTK for information on:

- System requirements
- Running the Setup program
- Setting installation options

Building a DTK Application

This section describes the workings of a DTK application and the various functions it calls. It contains the following:

- "What Can You Do with DTK?," next, lists the features that DTK provides for use in your database applications.
- "What Is a DTK Application?" on page 6 shows a sample written in C to illustrate the major parts of a DTK application.
- "Sample Programs" on page 9 describes the SAMPLE.EXE program provided with DTK and the sample routines it includes, which are reprinted throughout the chapters of Part I.

What Can You Do with DTK?

DTK provides a multi-database API that works with any database driver that complies with Microsoft's Open Database Connectivity (ODBC) standard. With DTK, you can write applications that are usable with any database system. Porting your application to another database system can be as easy as changing a single line of code.

DTK also manages all data-type conversion for you. DTK's data-retrieval functions automatically convert the native data type for a column into one of DTK's eight standard data types, removing any data conversion considerations that might affect portability. For situations where you need to use the native data types, DTK provides that capability.

DTK makes it easy to implement powerful features in your applications. With DTK, your applications can

- Be written using any development tool that can call a DLL, so you can continue to use your development tools. DTK lets you expand your capabilities while protecting your investment—don't worry about having to write new code or spend time learning new products.

- Query the system with DTK's data dictionary functions to determine what data sources, databases, tables, and stored procedures are available.
- Execute SQL statements on all database systems, even non-SQL database systems.
- Retrieve names, data types, and other information about the columns returned by Select statements.
- Scroll backward and forward through the records returned by a Select statement, even in databases that do not support backward scrolling. You can also position to a specific record by using its number.
- Update and delete records without issuing the SQL statements normally required to do so. DTK's current-record functions generate the appropriate statement for you. You can also insert a record that contains null values for columns that can be filled in later.
- Use transactions to group database operations so that they can be executed or canceled as a unit. The database drivers included with DTK let you use transactions even when connected to databases that do not support them.
- Provide your users with a Query By Example (QBE) option that lets them define retrieval conditions in the fields where those records are displayed.
- Use parameters for creating multipurpose SQL queries. DTK's parameter functions let you create queries containing parameters in their Where clauses. With these functions, you can create queries that use the results of previous queries and that can be modified by end users at runtime.
- Search the Select statement's result set to find records matching certain conditions using a single function.
- Include the Query Builder tool, which lets your users point and click to create Select statements—even if they don't know SQL.
- Parse the Where, Having, Group By, Order By, and Compute By clause, or other database-specific condition clauses from a Select statement.

- Rely on DTK's enhanced error handling functions when checking for errors and warning messages from the database system.
- Optimize the application's performance to suit the types of tasks it performs and database systems it uses.
- Call stored procedures and handle their results with functions designed specifically for that purpose.
- Take advantage of Microsoft's ODBC standard for portability among client/server database systems. All DTK applications are ODBC-compliant. Your DTK application can function with any ODBC-compliant database driver, regardless of the vendor that supplies it.

What Is a DTK Application?

A DTK application is any application that calls the DTK API to interact with a database. Such an application can be written in any programming language or environment that operates under Windows or OS/2.

The following C program shows part of a typical DTK application.

```
qeSTATUS bindfetch ()  {

/* This routine demonstrates how to use the bind functions to fetch data *    /
/* from Select statements directly into program variables. *    /

       qeHANDLE        hdbc = 0;       /* Handle to database connection *    /
       qeHANDLE        hstmt = 0;      /* Handle to SQL statement execution *    /
       qeSTATUS        res_code;       /* Result code from DTK functions *    /
       char            last_name [11]  ;
       long            last_name_len = 11  ;
       float           salary  ;
       long            salary_len = sizeof(salary)   ;

/* Call qeLibInit to initialize DTK, check for errors. *    /
       res_code = qeLibInit ()  ;
       if (res_code != qeSUCCESS) return (res_code)   ;

/* Call qeConnect to connect to a data source.  Check to see *    /
/* if hdbc == 0, which indicates that the connection failed. *    /
```

```
       hdbc = qeConnect ("DSN=QEDBF")  ;
       if (hdbc == 0) return (err_handler (hdbc, hstmt))   ;

/* Call qeExecSQL to execute the select statement. Check if hstmt == 0, *    /
/* which indicates that the statement did not execute successfully. *    /
       hstmt = qeExecSQL (hdbc, "Select last_name, salary from emp")    ;
       if (hstmt == 0) return (err_handler (hdbc, hstmt))   ;

/* Bind the result columns to program variables. *   /
       res_code = qeBindColChar (hstmt, 1, last_name, &last_name_len, "")    ;
       if (res_code != qeSUCCESS) return (err_handler (hdbc, hstmt))    ;

       res_code = qeBindColFloat (hstmt, 2, &salary, &salary_len)    ;
       if (res_code != qeSUCCESS) return (err_handler (hdbc, hstmt))    ;

/* Fetch the records from the select statement. *   /
       while (qeFetchNext (hstmt) == qeSUCCESS)    {
               MessageBox (hWnd, last_name, "Bind Fetch", MB_OK)    ;
        }

/* Check for errors, EOF is ok. *  /
       res_code = qeErr ()  ;
       if ((res_code != qeSUCCESS) && (res_code != qeEOF)    )
               return (err_handler (hdbc, hstmt))   ;
/* Close the SQL statement. *  /
       res_code = qeEndSQL (hstmt)  ;
       hstmt = 0 ;
       if (res_code != qeSUCCESS) return (err_handler (hdbc, hstmt))    ;

/* Call qeDisconnect to disconnect from a data source. *   /
       res_code = qeDisconnect (hdbc)  ;
       hdbc = 0 ;
       if (res_code != qeSUCCESS) return (err_handler (hdbc, hstmt))    ;

/* Call qeLibTerm to free memory allocated by DTK. *   /
       res_code = qeLibTerm ()  ;
       MessageBox (hWnd, "Sample succeeded.", "Bind Fetch", MB_OK)    ;
       return (res_code)  ;

}

qeSTATUS err_handler (qeHANDLE hdbc, qeHANDLE hstmt)    {

/* This routine functions as an error handler for the demonstration  *    /
/* routines. It displays an error message in a message box, terminates  *    /
```

```
/* the active statement, closes the connection, ends the DTK session,  *    /
/* and returns the result code from the most recent DTK call.  *    /

       qeSTATUS res_code;                  /* Result code from DTK functions  *    /

       res_code = qeErr ();                /* Get the DTK error number  *    /
       if (res_code > 0)                   /* Display an error message  *    /
              MessageBox (hWnd, qeErrMsg (), "Sample Failed", MB_OK)    ;
       else {              /* Display error number for negative result codes  *     /
              char     buf[10]  ;
              MessageBox (hWnd, itoa (res_code, buf, 10), "Sample Failed", MB_OK)     ;
        }
       if (hstmt != 0) qeEndSQL (hstmt);   /* End statement if active  *    /
       if (hdbc != 0)  qeDisconnect (hdbc); /* Close connection if open *    /
       qeLibTerm ();                       /* Last call to the DTK API  *    /
       return (res_code) ;
}
```

This sample illustrates the processes common to most DTK applications:

Initializing and terminating DTK tasks. This is discussed in Chapter 2, “Connecting to Databases,” on page 19.

Connecting and disconnecting with the database system. This is also discussed in Chapter 2, “Connecting to Databases,” on page 19.

Executing SQL statements. This is discussed in Chapter 3, “Executing SQL Statements,” on page 25.

Fetching records and their values. This is discussed in Chapter 4, “Retrieving and Converting Data,” on page 37.”

Other DTK capabilities and functions not covered in this sample are discussed in the following chapters:

Chapter 3, “Executing SQL Statements,” on page 25 also describes the parameter-binding functions that let you use dynamic and user-defined conditions in the Where clause of SQL statements.

Chapter 4, “Retrieving and Converting Data,” on page 37 also describes the functions that extract column values from the tables in the current SQL statement.

Chapter 5, “Modifying Data,” on page 71 describes the current-record functions that let you add, change, or delete records in the database without issuing SQL statements.

Chapter 6, “Using Transaction Functions,” on page 81 describes the functions that group database operations so that they can be executed or canceled as a unit.

Chapter 7, “Error Handling and Debugging,” on page 117 describes the functions that report errors and that trace the execution of DTK functions.

Chapter 8, “QBE and Query Builder Functions,” on page 127 describes the functions that let you implement Query By Example and the Query Builder tool in your application’s user interface.

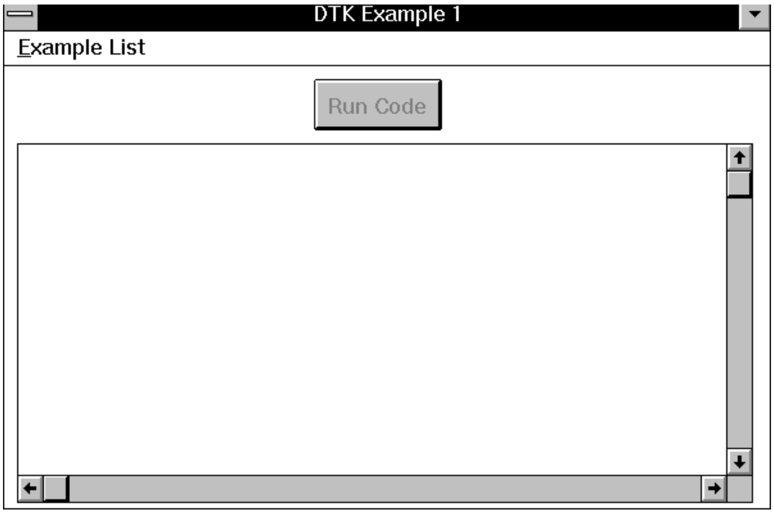
Chapter 9, “Utility Functions,” on page 143 describes DTK’s data dictionary functions, which return information on the sources to which you are connected, as well as the functions for parsing SQL statements and converting DTK’s connection handles in order to call ODBC functions directly.

Sample Programs

The DTK disks include a number of sample DTK applications, including the SAMPLE.EXE program described in the following section.

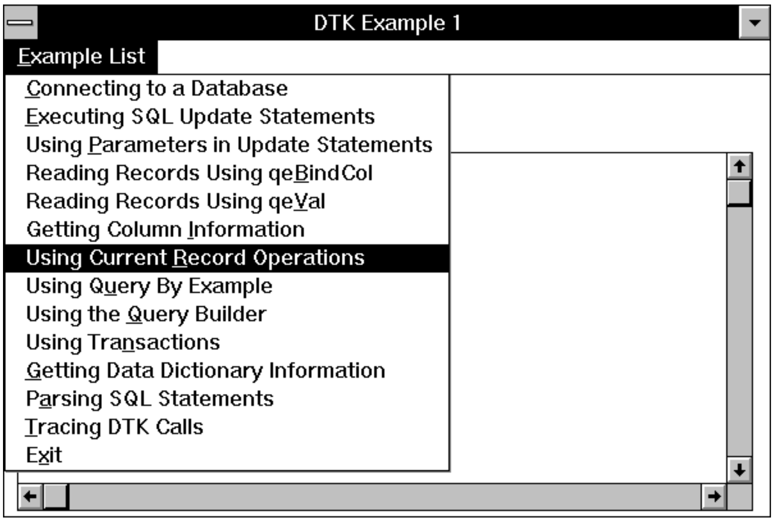
Running SAMPLE.EXE

The code samples in this book are taken from the program SAMPLE.EXE, which is included on the DTK diskette. You can run SAMPLE.EXE to both view and execute each example used in Part I. All of the samples are written in C.

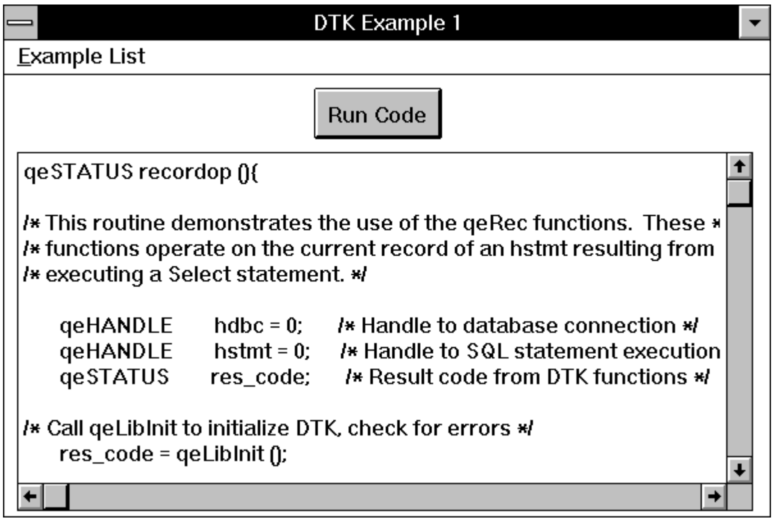


Running this sample program is a good way to get started with DTK. It provides a drop-down menu of DTK examples.

To load an example, select it from this menu.



When you've chosen an example, you can run it by clicking the Run Code button.



Note: You can copy text from this window to another Windows program that handles text as follows:

1. Drag the mouse to highlight the sample code.
2. Press CTRL + INSERT to copy the highlighted text to the clipboard.
3. Click in the application where you want to place the sample code.
4. Press SHIFT + INSERT to paste the copied sample.

If you are running Windows or Presentation Manager in a high-resolution mode, some lines of sample code displayed by SAMPLE.EXE will be longer than the display window. If you find this annoying, and don't want to change your resolution, you can use this method to copy the samples to Notepad or some other application for better viewing.

The Example List menu lets you choose among the following samples:

- Connecting to a Database
  Initializes DTK, connects and disconnects from a database system, and terminates DTK. This sample is listed in Chapter 2, “Connecting to Databases,” on page 19.

- Executing SQL Update Statements
  Issues a SQL Update statement directly via a DTK function call. This sample is listed in Chapter 3, “Executing SQL Statements,” on page 25.

- Using Parameters in Update Statements
  Uses the parameter binding functions that let you create dynamic SQL statements to which your users can supply values. This sample is listed in Chapter 3, “Executing SQL Statements,” on page 25.

- Reading Records Using qeBindCol
  Fetches and reads record values from the database using DTK’s column binding functions. This sample is listed in Chapter 4, “Retrieving and Converting Data,” on page 37.

- Reading Records Using qeVal
  Fetches and reads record values from the database using DTK’s value extracting functions. This sample is listed in Chapter 4, “Retrieving and Converting Data,” on page 37.

- Getting Column Information
  Uses one of DTK’s column information functions to get data types for each column returned by a Select statement. This sample is listed in Chapter 4, “Retrieving and Converting Data,” on page 37.

- Using Current Record Operations
  Changes values in the database using DTK’s current record functions. This sample is listed in Chapter 5, “Modifying Data,” on page 71.

- Using Query By Example
  Uses the QBE functions to retrieve a record with a first name value beginning with “T.” This sample is listed in Chapter 8, “QBE and Query Builder Functions,” on page 127.

- Using the Query Builder
  Calls the Query Builder, an interface that lets users point and click to create SQL statements. This sample is listed in Chapter 8, “QBE and Query Builder Functions,” on page 127.

- Using Transactions
  Uses DTK functions to group database modifications into transactions. This sample is listed in Chapter 6, “Using Transaction Functions,” on page 81.

- Getting Data Dictionary Information
  Uses one of DTK’s data dictionary functions to return all of the ODBC-defined data sources available to the application. This sample is listed in Chapter 9, “Utility Functions,” on page 143.

- Parsing SQL Statements
  Retrieves the Where clause of a SQL statement. This sample is listed in Chapter 9, “Utility Functions,” on page 143.

- Tracing DTK Calls
  Uses the functions that let you trace DTK and ODBC calls. This sample is listed in Chapter 7, “Error Handling and Debugging,” on page 117.

Running Other Sample Programs

In addition to the SAMPLE.EXE program, several other sample programs are included on the DTK disks. These sample programs were installed with DTK if you selected the option to do so when running the Setup program.

To see a list of these programs, open the README.HLP file that was installed in your DTK directory. You can double-click on this file from the File Manager to view it. The help window that appears includes short descriptions of each sample program.

If you did not install the sample programs when installing DTK, you can rerun the Setup program from the first DTK disk to install them without reinstalling DTK.

Solving Problems and Getting Technical Support

INTERSOLV gives you a variety of options for choosing the kind of technical support that fits your needs.

Product Documentation

This product provides both printed manuals and online Help files. Take time to explore these information sources; they are designed to help you learn how to use the product and also serve as reference material for daily use.

This manual describes DTK's functionality and provides reference information. The INTERSOLV database driver reference that accompanies it covers the database drivers included with the product; see this book for driver-specific information about system requirements, connection string options, and the particular implementation of SQL.

Technical Support for Registered Users

Please register your product immediately by sending in the registration card enclosed in your product box. Upon registration, you are automatically entitled to the following services:

- FaxPLUS can send you the latest marketing and technical information on INTERSOLV products, 24 hours a day, seven days a week. Call FaxPLUS from any touch-tone phone, and have your fax number ready. When calling FaxPLUS, you can learn how the system works, order individual documents, or order a catalog of documents. The FaxPLUS number is 1-800-432-3984.
- INTERSOLV's CompuServe forum offers 24-hour access to information. You can download files for review or installation, and share information with other users. To use this forum, type GO INTERSOLV. If you do not know your local access number for CompuServe, call 1-800-848-8990.

ServiceDirect is our solution for ensuring ongoing success with your INTERSOLV product. With ServiceDirect coverage, you are entitled to:

For more information about these services, call INTERSOLV's ServiceDirect Department at (800) 443-1601.

- Answerline Services. Technical experts are available through the toll-free Answerline number to share their experience in using INTERSOLV products.
- Product Maintenance Releases. Maintenance releases provide periodic enhancements to current products with more frequent updates.
- New Product Releases. You can leverage your investment by updating your current technology with new product releases, which provide enhanced functionality.
- Technical Bulletins. These bulletins provide product-specific tips, techniques, and technical routines to keep you proficient in current products.
- The INTERLINK Customer Newsletter. INTERLINK keeps you informed about current products, support and services, courses, user groups, and conferences.

Our Implementation Services Group offers a wide range of services that include customized training, installation and tuning, mentoring, ODBC-compliant application testing, and consulting. Contact the Implementation Services Group by phone at (800) 443-1601 or by fax at (301) 230-3314.

Our Educational Services Department offers a wide range of prescheduled classes. Call 1-800-443-1601 to obtain more information on these classes.

Before You Cal

Before you call for technical support, please try to learn as much as you can about the problems you are experiencing. Our Technical Support representatives can address your problems much faster if you have all the information they need when you call.

To streamline the problem-solving process, follow these steps before calling INTERSOLV Answerline:

- *Gather basic information* about your system to help us understand the environment in which you are working.
- *Identify the category of your product usage* so that you can effectively prepare for telephone support.
- *Troubleshoot* to learn more about the nature of the problem.

Calling for Technical Suppot

If you live in North America, call INTERSOLV Answerline at (800) 443-1601. Technical support representatives can take your call from 8:30 a.m. to 8 p.m. EST.

If you live in another location, call the international distributor nearest you. Be sure to read the online Help for support information and requirements that are specific to your geographic location.

Go To

2 Connecting to Databases

This chapter discusses the database connection functions and the DTK initialization and termination functions.

The following sample code shows how to initialize DTK, connect and disconnect from a database system, and terminate DTK. To load this sample in the SAMPLE.EXE program, choose **Connecting to a Database** from the Example List.

```
qeSTATUS connect () {

/* This routine connects to the dBASE driver, and then disconnects. */

      qeHANDLE       hdbc = 0;      /* Handle to database connection */
      qeHANDLE       hstmt = 0;     /* Handle to SQL statement execution */
      qeSTATUS       res_code;      /* Result code from DTK functions */

/* Call qeLibInit to initialize DTK, check for errors. */
      res_code = qeLibInit () ;
      if (res_code != qeSUCCESS) return (res_code) ;

/* Call qeConnect to connect to a data source.  Check to see */
/* if hdbc == 0, which indicates that the connection failed. */
      hdbc = qeConnect ("DSN=QEDBF") ;
      if (hdbc == 0) return (err_handler (hdbc, hstmt)) ;

/* Insert code here to execute SQL statements, fetch records, etc. */

/* Call qeDisconnect to disconnect from a data source. */
      res_code = qeDisconnect (hdbc) ;
      hdbc = 0 ;
      if (res_code != qeSUCCESS) return (err_handler (hdbc, hstmt)) ;
/* Call qeLibTerm to free memory allocated by DTK. */
      res_code = qeLibTerm () ;
      MessageBox (hWnd, "Sample succeeded.", "Connect", MB_OK) ;
      return (res_code) ;
}
qeSTATUS err_handler (qeHANDLE hdbc, qeHANDLE hstmt) {
```

```
/* This routine functions as an error handler for the demonstration  *     /
/* routines. It displays an error message in a message box, terminates  *     /
/* the active statement, closes the connection, ends the DTK session,  *     /
/* and returns the result code from the most recent DTK call.  *     /

       qeSTATUS res_code;                  /* Result code from DTK functions  *     /

       res_code = qeErr ();                /* Get the DTK error number   *     /
       if (res_code > 0)                   /* Display an error message  *     /
              MessageBox (hWnd, qeErrMsg (), "Sample Failed", MB_OK)     ;
       else {                /* Display error number for negative result codes  *     /
              char     buf[10]  ;
              MessageBox (hWnd, itoa (res_code, buf, 10), "Sample Failed", MB_OK)     ;
        }
       if (hstmt != 0) qeEndSQL (hstmt);    /* End statement if active  *     /
       if (hdbc != 0)  qeDisconnect (hdbc); /* Close connection if open *     /
       qeLibTerm ();                       /* Last call to the DTK API  *     /
       return (res_code)  ;
}
```

The qeConnect, qeDisconnect, qeLibInit, and qeLibTerm functions used in this example are described in the following sections.

Connecting to a Database

Before you can send SQL statements to a database system to be executed, you must open a connection to the database system.

Table 2-1 lists the functions DTK provides for establishing and closing a database connection:

Table 2-1. Functions that Establish or Close Database Connections

Function	Result
qeConnect	Opens a database system connection.
qeGetLoginTimeout	Returns the current login timeout value.
qeSetLoginTimeout	Sets the login timeout. The default is 15 seconds.
qeSetDB	Sets the default database for the application.

The qeConnect function connects your application to a database system. The parameters to qeConnect identify the database system. qeConnect returns a handle to a database connection, or hdbc. The hdbc identifies the connection and is a parameter to other functions.

When using a database system that lets you store tables in separate databases, you can set the default database for your application with a call to qeSetDB. Once qeSetDB sets the default database, all subsequent SQL statements will be sent to that database.

The qeDisconnect function closes a connection. The parameter to qeDisconnect is the hdbc returned by qeConnect. Once you have called qeDisconnect, you cannot perform any other functions on this connection.

You can have several connections open simultaneously. For example, to copy records from a Paradox file to an Oracle database, you would use one connection to Paradox and a second one to Oracle.

Note: You connect to a database system (such as dBASE, Paradox, Oracle, SQL Server), not to a specific file or table. The SQL statements identify the files or tables that are to be accessed.

Initializing and Terminating DTK

Two DTK functions, qeLibInit and qeLibTerm, specify the beginning and end of a DTK program. If you write a multi-threaded application, you should call these functions to initialize and terminate each thread of execution. Table 2-2 lists the functions.

Table 2-2. Functions that Initialize and Terminate DTK Program

Function	Result
qeLibInit	Initializes DTK.
qeLibTerm	Terminates DTK.

qeLibInit should be the first DTK function that your application calls. Calling qeLibInit ensures that DTK will allocate the memory resources that it needs. Calling qeLibTerm ensures that those memory resources are freed as soon as they are no longer needed.

Getting Setup and Version Information

Table 2-3 lists the functions DTK provides for retrieving setup information and version numbers:

Table 2-3. Functions that Retrieve Setup Information and Version Numbers

Function	Result
qeSetupInfo and qeSetupInfoBuf	The information entered when DTK was installed
qeVerNum and qeVerNumBuf	The number of the DTK version you are using

Go To

3 Executing SQL Statements

This chapter describes the use of DTK's SQL execution and statement parameter functions in the following sections:

- "Executing SQL Statements," next, describes DTK's SQL statement preparation and execution functions.
- "Using Statement Parameters" on page 28 describes the functions that let you use parameters within the Where clause of SQL statements.
- "Using Stored Procedures" on page 33 describes the functions that let you use input and input/output parameters in stored procedures.
- "Join Behavior in DTK" on page 35 describes how DTK behaves when working with records from joined tables.

Executing SQL Statements

The following sample code shows one way to execute a SQL statement from DTK.

To load this sample in the SAMPLE.EXE program, choose **Executing SQL Update Statements** from the Example List.

```
qeSTATUS dml ()  {

/* This routine demonstrates how to execute an SQL Update statement. *     /

       qeHANDLE        hdbc = 0;       /* Handle to database connection *     /
       qeHANDLE        hstmt = 0;      /* Handle to SQL statement execution *     /
       qeSTATUS        res_code;       /* Result code from DTK functions *     /
       long            modrecs  ;
```

```
/* Call qeLibInit to initialize DTK, check for errors. *   /
       res_code = qeLibInit ()  ;
       if (res_code != qeSUCCESS) return (res_code)   ;

/* Call qeConnect to connect to a data source.  Check to see *   /
/* if hdbc == 0, which indicates that the connection failed. *   /
       hdbc = qeConnect ("DSN=QEDBF")  ;
       if (hdbc == 0) return (err_handler (hdbc, hstmt))   ;

/* Call qeExecSQL to execute the update statement. Check if hstmt == 0, *    /
/* which indicates that the statement did not execute successfully. *    /
      hstmt = qeExecSQL (hdbc, "Update emp set first_name = 'Joe' where first_name
= 'Richard'") ;
       if (hstmt == 0) return (err_handler (hdbc, hstmt))   ;

/* Find out how many records were affected by the statement. *   /
       modrecs = qeNumModRecs (hstmt)  ;
       if (qeErr () != qeSUCCESS) return (err_handler (hdbc, hstmt))    ;

/* Close the statement. *  /
       res_code = qeEndSQL (hstmt)  ;
       hstmt = 0 ;
       if (res_code != qeSUCCESS) return (err_handler (hdbc, hstmt))    ;

/* Call qeDisconnect to disconnect from a data source. *   /
       res_code = qeDisconnect (hdbc)  ;
       hdbc = 0 ;
       if (res_code != qeSUCCESS) return (err_handler (hdbc, hstmt))    ;

/* Call qeLibTerm to free memory allocated by DTK. *   /
       res_code = qeLibTerm ()  ;
       MessageBox (hWnd, "Sample succeeded.", "SQL Update Statement", MB_OK)    ;
       return (res_code)  ;
}

/* err_handler routine goes here.  *  /
```

This sample on page 25 shows how to use the qeExecSQL, qeEndSQL and qeNumModRecs functions.

Table 3-1 shows the entire set of DTK SQL statement execution functions.

Table 3-1. Functions that Execute SQL Statements

Function	Action
qeExecSQL	Prepares and executes a SQL statement
qeSQLPrepare	Prepares a SQL statement for execution.
qeSQLExecute	Executes a statement that was prepared with qeSQLPrepare.
qeSetSQL	Places a partial statement in the SQL buffer.
qeAppendSQL	Appends text to the SQL buffer.
qeMoreResults	Begins a new result set from stored procedures or multiple SQL statements.
qeNumModRecs	Returns the number of records modified by a SQL statement.
qeEndSQL	Ends the execution of a SQL statement. It is important to call qeEndSQL to free system resources.
qeSetQueryTimeout	Sets the time to wait for a SQL statement to execute before returning to the application.
qeGetQueryTimeout	Returns the time to wait for a SQL statement to execute before returning to the application.
qeSetOneHstmtPerHdbcOptions	Provides control over DTK behavior when connected to databases that support only one statement per connection.
qeGetOneHstmtPerHdbcOptions	Returns the flag settings that determine DTK behavior when connected to databases that support only one statement per connection.

Once you have opened a connection, you can send SQL statements to the underlying database system. The qeExecSQL function prepares and executes a SQL statement. The parameters to qeExecSQL are the *hdbc* of the connection to use and the SQL statement. qeExecSQL returns a handle for the statement (*hstmt*). The *hstmt* identifies the statement and is a parameter to other functions that operate on the statement.

When the SQL statement is a Select statement, qeExecSQL does not return the resulting records. The records are read using the data fetching functions described in Chapter 5, “Modifying Data,” on page 71.

The qeSQLPrepare and qeSQLExecute functions are provided for when you want your application to process a SQL statement. For example, to issue a SQL statement containing parameters, you first call qeSQLPrepare to prepare the statement—place it in the statement buffer and return a handle to it (*hstmt*). You use qeSQLExecute to execute the statement once the parameters are bound or set by the parameter functions discussed in the next section.

The qeEndSQL function terminates a statement and frees the system resources allocated to it. The parameter you supply to qeEndSQL is the *hstmt* returned by qeExecSQL or qeSQLPrepare. Once you have called qeEndSQL, you cannot perform any other functions on this statement.

Some macro languages (like Excel) limit the length of character strings, which makes it impossible to send a complete SQL statement to qeExecSQL. For these languages, use the qeSetSQL and qeAppendSQL functions to send the SQL statement in parts.

Using Statement Parameters

DTK provides functions that allow you to use parameters in SQL statements. By using parameters instead of values in a SQL statement, you can improve the performance of your applications.

The sample on page 29 shows a SQL statement that uses parameters. To load this sample in the SAMPLE.EXE program, choose **Using Parameters in Update Statements** from the Example List.

```
qeSTATUS params ()  {

/* This routine demonstrates the use of bound parameters in SQL Update *    /
/* statements. * /

       qeHANDLE        hdbc = 0;       /* Handle to database connection *    /
       qeHANDLE        hstmt = 0;      /* Handle to SQL statement execution *    /
       qeSTATUS        res_code;       /* Result code from DTK functions *    /
       char            new_name[31], old_name[31]   ;
       long            new_name_len = 30, old_name_len = 30    ;

/* Call qeLibInit to initialize DTK, check for errors. *   /
       res_code = qeLibInit ()  ;
       if (res_code != qeSUCCESS) return (res_code)   ;

/* Call qeConnect to connect to a data source.  Check to see *    /
/* if hdbc == 0, which indicates that the connection failed. *    /
       hdbc = qeConnect ("DSN=QEDBF")  ;
       if (hdbc == 0) return (err_handler (hdbc, hstmt))   ;

/* Call qeSQLPrepare to prepare the update statement. The Update statement *    /
/* will change the first_name of an employee.  Check if hstmt == 0, *    /
/* which indicates that the function did not succeed. *   /
      hstmt = qeSQLPrepare (hdbc, "Update emp set first_name = ? where first_name
= ?");
       if (hstmt == 0) return (err_handler (hdbc, hstmt))    ;

/* Bind the parameters to local variables that contain the parameter values. *    /
       res_code = qeBindParamChar (hstmt, 1, new_name, &new_name_len)    ;
       if (res_code != qeSUCCESS) return (err_handler (hdbc, hstmt))    ;

       res_code = qeBindParamChar (hstmt, 2, old_name, &old_name_len)    ;
       if (res_code != qeSUCCESS) return (err_handler (hdbc, hstmt))    ;

/* Set the parameters. *  /
       lstrcpy (new_name, "Joe")  ;
       new_name_len = lstrlen (new_name)   ;
       lstrcpy (old_name, "Tim")  ;
       old_name_len = lstrlen (old_name)   ;

/* Execute the statement. *  /
       res_code = qeSQLExecute (hstmt)  ;
       if (res_code != qeSUCCESS) return (err_handler (hdbc, hstmt))    ;
```

```
/* Note: To make repeated updates, you simply change the new_name and *     /
/* old_name variables and call qeSQLExecute. */
/* Close the statement. *  /
       res_code = qeEndSQL (hstmt)  ;
       hstmt = 0 ;
       if (res_code != qeSUCCESS) return (err_handler (hdbc, hstmt))    ;

/* Call qeDisconnect to disconnect from a data source. *    /
       res_code = qeDisconnect (hdbc)  ;
       hdbc = 0 ;
       if (res_code != qeSUCCESS) return (err_handler (hdbc, hstmt))    ;

/* Call qeLibTerm to free memory allocated by DTK. *   /
       res_code = qeLibTerm ()  ;
       MessageBox (hWnd, "Sample succeeded.", "Binding Parameters", MB_OK)    ;
       return (res_code) ;
}

/* err_handler routine goes here.  *  /
```

This sample shows how to use the qeSQLPrepare, qeBindParamChar, and qeSQLExecute functions. Table 3-2 lists the entire set of DTK SQL statement parameter functions.

Table 3-2. Functions to Use with SQL-Statement Input Parameter

Function	Results
qeBindParamBinary	Binds a parameter to a binary variable.
qeBindParamChar	Binds a parameter to a character variable.
qeBindParamDate	Binds a parameter to a date variable.
qeBindParamDateTime	Binds a parameter to a date-time variable.
qeBindParamDecimal	Binds a parameter to a decimal variable.
qeBindParamDouble	Binds a parameter to a double-precision floating-point variable.
qeBindParamFloat	Binds a parameter to a floating-point variable.
qeBindParamInt	Binds a parameter to a 2-byte integer variable.
qeBindParamLong	Binds a parameter to a 4-byte integer variable.

Table 3-2. Functions to Use with SQL-Statement Input Parameters (cont.)

Function	Results
qeBindParamTime	Binds a parameter to a time variable.
qeClearParam	Clears the value of a parameter.
qeNumParams	Returns the number of parameters that appeared in the statement.
qeParamNum	Returns the number of the parameter corresponding to a specified name.
qeSetParamBinary	Sets the value of a binary parameter.
qeSetParamChar	Sets the value of a character parameter.
qeSetParamDate	Sets the value of a date parameter.
qeSetParamDateTime	Sets the value of a date-time parameter.
qeSetParamDecimal	Sets the value of a decimal parameter.
qeSetParamDouble	Sets the value of a double-precision floating-point parameter.
qeSetParamFloat	Sets the value of a floating-point parameter.
qeSetParamInt	Sets the value of a 2-byte integer parameter.
qeSetParamIOType	Sets a parameter's input/output (IO) type.
qeSetParamLong	Sets the value of a 4-byte integer parameter.
qeSetParamTime	Sets the value of a time parameter.
qeSetParamNull	Sets the value of a parameter.

These functions let you create *parameterized* queries—queries in which certain criteria are supplied by the user or by other processes within the program.

To use parameters in a SQL statement, first call the qeSQLPrepare function, which takes as an argument a SQL statement string that contains question marks (?) to identify the position of the parameters in the statement. qeSQLPrepare returns the handle to the statement (*hstmt*) that you use in other DTK calls. The question marks in the statement may be followed by a name for the parameter. You can refer to the parameters either by name or by their order in the SQL statement. To use named parameters, you must call qeParamNum to convert parameter names to numbers.

A parameter can be one of three types: *input*, *output*, or *input/output*. An input parameter passes a value to the SQL statement, an output parameter stores a result from an executed SQL statement, and an input/output parameter does both. Input/output and output parameters are used only with stored procedures (see “Using Stored Procedures” on page 33). DTK needs to know whether a parameter is an input, output, or input/output parameter. To set the parameter input/output (I/O) type for each parameter, use the function qeSetParamIOType, which should be called for every parameter in new code.

Note: If qeSetParamIOType is not called, DTK assumes the parameter is an input parameter. Thus, existing code that works with input parameters will continue to run without any changes.

Two sets of functions are provided for setting parameter values. The qeSetParam functions assign a value directly to a parameter. The qeBindParam functions bind the parameter to a buffer that holds the value. In both sets of functions, the second argument is a number representing the position of the parameter. The qeClearParam function removes assigned values from the parameters. The qeNumParams function returns the number of parameters in the statement.

After you have assigned values to all parameters in the statement, call qeSQLExecute to execute the statement.

Using Stored Procedures

As discussed in "Using Statement Parameters" on page 28, DTK provides functions that let you use parameters in SQL statementsthat are written in your code, and values must be provided for statement parameters before the SQL statements are processed. Because the parameters in source-code SQL statements always provide input values, they are considered input parameters.

Some database systems let you store compiled sequences of SQL statements directly in the database; these SQL statements are called *stored procedures*. As with the SQL statements in your source code, stored procedures frequently use input parameters.

Many, though not all, of the database systems that support stored procedures let you use output parameters with the procedure. An output parameter is a parameter that stores a result of the SQL statement execution. In some situations, the same parameter might provide an input value, and then be used to return a result. In that case, the parameter is considered an input/output parameter.

To reference a stored procedure in DTK, use a qeSQLPrepare function's second argument to pass a call to the stored procedure, then use the qeSQLExecute function to execute the stored procedure. For example,

```
hdbc = qeConnect ("DSN=QEDBF")   ;
hstmt = qeSQLPrepare (hdbc, "{call DeptName(?)}")    ;
. . .
res_code = qeSQLExecute(hstmt)   ;
```

calls a stored procedure named DeptName.

In a stored procedure, DTK needs to know whether a parameter is an input, output, or input/output parameter. To set the parameter input/output (I/O) type, use qeSetParamIOType, which you must call for each parameter.

The qeBindParam functions set the data type of input, output, or input/output parameters. You must call a qeBindParam function for each parameter. For input parameters, the value in the buffer is used for execution of the SQL statement or stored procedures. For output parameters, the value generated for the parameter from the stored procedure is placed into the buffer.

When an application is not binding parameters, the DTK has a set of qeGetParam functions to retrieve parameter values of output parameters after the qeSQLExecute is complete. These functions are similar to the qeVal functions.

The qeBindParam and qeSetParam functions tell DTK the data type of each parameter; however, when the qeBindParam functions are not being used, the qeSetParam functions are called to set input and input/output parameters but not output parameters. To set the data type and size of output parameters, use qeSetParamDataType.

Table 3-3 lists the set of DTK functions that support output and input/output parameters in stored procedures.

Table 3-3. Functions that Support Stored-Procedure I/O Parameters

Function	Results
qeSetParamIOType	Sets a parameter's I/O type.
qeSetParamDataType	Sets the data type of a stored procedure's output parameters.
qeGetParamBinary and qeGetParamBinaryBuf	Return an output or input/output parameter's value as a binary value.
qeGetParamBit	Returns an output or input/output parameter's value as a bit in a 2-byte integer.
qeGetParamChar and qeGetParamCharBuf	Return a character string containing the value from an output or input/output parameter.
qeGetParamDate and qeGetParamDateBuf	Return an output or input/output parameter's value as a date value.
qeGetParamDateTime and qeGetParamDateTimeBuf	Return an output or input/output parameter's value as a date-time value.

Table 3-3. Functions that Support Stored-Procedure I/O Parameters

Function	Results
qeGetParamDecimal and qeGetParamDecimalBuf	Return an output or input/output parameter's value as a decimal value.
qeGetParamDouble	Returns an output or input/output parameter's value as a double-precision floating point number.
qeGetParamFloat	Returns an output or input/output parameter's value as a single-precision floating point number.
qeGetParamInt	Returns an output or input/output parameter's value as a 2-byte integer.
qeGetParamLong	Returns an output or input/output parameter's value as a 4-byte integer.
qeGetParamTime and qeGetParamTimeBuf	Return an output or input/output parameter's value as a time value.

Join Behavior in DTK

A *join* combines two or more database tables in one SQL Select statement. The joined tables must share a common column having values that can be compared to join the records in each table. For example, the following Select statement creates a join of the tables EMP and DEPT:

```
SELECT first_name, last_name, dept, dept_name
    FROM emp,dept WHERE emp.dept = dept.dept_id
```

The EMP and DEPT tables can be joined because the DEPT and DEPT_ID columns contain department ID values that join the records for each employee with those for each department.

DTK allows you to issue SQL statements that join multiple tables in the same database system. The joins are performed by the database system, not by DTK, so you cannot join tables from different database systems. For systems that separate tables into multiple databases, DTK can join tables in separate databases if the database system supports such joins.

DTK allows you to update records returned from joined tables, but you cannot insert or delete them.

Go To

4 Retrieving and Converting Data

This chapter describes the DTK functions that retrieve record and column information, as well as the functions that convert the data types of database values. It contains the following sections:

- "Fetching Records" on page 37 describes the qeFetch functions, which retrieve records from a Select statement's result set, the qeBindCol functions, which bind column values to variable buffers, and the qeVal functions, which retrieve individual column values.
- "Getting Column Information" on page 47 describes the qeCol functions, which retrieve information describing the columns returned by a Select statement.
- "Converting Data Types" on page 50 lists the data conversion functions that DTK provides.
- "Data Types in DTK" on page 53 discusses data type usage and conventions in DTK, as well as specific considerations for handling some data types.
- "Format Strings" on page 59 lists the format strings available for formatting your data when using the data-type conversion functions.

Fetching Records

DTK lets you use two different techniques to read records from databases.

The preferred technique uses the qeBindCol functions to bind variables to each of the columns in the Select statement prior to calling a qeFetch function. Each time a record is fetched, the variables are automatically filled with the column values and their lengths.

Another technique is to use the qeVal functions to read each column value separately after calling a qeFetch function. Using this method, you must call the same set of qeVal functions after fetching each record. Using the qeVal functions is slower than using the qeBindCol functions. However, some macro and script languages do not permit you to use functions like qeBindCol that pass pointers to integer variables.

The following sample uses the column binding method. To load this sample in the SAMPLE.EXE program, choose **Reading Records Using qeBindCb** from the Example List.

```
qeSTATUS bindfetch ()  {

/* This routine demonstrates how to use the bind functions to fetch data *     /
/* from Select statements directly into program variables. *    /

       qeHANDLE        hdbc = 0;       /* Handle to database connection *    /
       qeHANDLE        hstmt = 0;      /* Handle to SQL statement execution *     /
       qeSTATUS        res_code;       /* Result code from DTK functions *     /
       char            last_name [11]  ;
       long            last_name_len = 11   ;
       float           salary  ;
       long            salary_len = sizeof(salary)   ;

/* Call qeLibInit to initialize DTK, check for errors. *    /
       res_code = qeLibInit ()  ;
       if (res_code != qeSUCCESS) return (res_code)   ;
/* Call qeConnect to connect to a data source.  Check to see *    /
/* if hdbc == 0, which indicates that the connection failed. *    /
       hdbc = qeConnect ("DSN=QEDBF")  ;
       if (hdbc == 0) return (err_handler (hdbc, hstmt))   ;

/* Call qeExecSQL to execute the select statement. Check if hstmt == 0, *     /
/* which indicates that the statement did not execute successfully. *     /
       hstmt = qeExecSQL (hdbc, "Select last_name, salary from emp")    ;
       if (hstmt == 0) return (err_handler (hdbc, hstmt))   ;

/* Bind the result columns to program variables. *   /
       res_code = qeBindColChar (hstmt, 1, last_name, &last_name_len, "")     ;
       if (res_code != qeSUCCESS) return (err_handler (hdbc, hstmt))    ;

       res_code = qeBindColFloat (hstmt, 2, &salary, &salary_len)    ;
       if (res_code != qeSUCCESS) return (err_handler (hdbc, hstmt))    ;
```

```
/* Fetch the records from the select statement. *   /
       while (qeFetchNext (hstmt) == qeSUCCESS)    {
               MessageBox (hWnd, last_name, "Bind Fetch", MB_OK)    ;
        }

/* Check for errors, EOF is ok. *  /
       res_code = qeErr ()  ;
       if ((res_code != qeSUCCESS) && (res_code != qeEOF)    )
               return (err_handler (hdbc, hstmt))   ;

/* Close the SQL statement. *  /
       res_code = qeEndSQL (hstmt)  ;
       hstmt = 0 ;
       if (res_code != qeSUCCESS) return (err_handler (hdbc, hstmt))    ;

/* Call qeDisconnect to disconnect from a data source. *    /
       res_code = qeDisconnect (hdbc)  ;
       hdbc = 0 ;
       if (res_code != qeSUCCESS) return (err_handler (hdbc, hstmt))    ;

/* Call qeLibTerm to free memory allocated by DTK. *   /
       res_code = qeLibTerm ()  ;
       MessageBox (hWnd, "Sample succeeded.", "Bind Fetch", MB_OK)    ;
       return (res_code)  ;
}
/* err_handler routine goes here.  *  /
```

This example uses the qeFetchNext function to retrieve each record from the result set, and uses the column binding function qeBindColChar to get the values from each record.

The column binding functions are described in "Binding Data to Columns" on page 41.

Table 4-1 lists the set of DTK data fetching functions.

Table 4-1. Functions that Fetch Data

Function	Result
qeFetchNext	Retrieves the next record returned by the hstmt
qeFetchPrev	Retrieves the previous record returned by the hstmt
qeFetchRandom	Retrieves a specified record returned by the hstmt
qeFetchNumRecs	Returns the number of records chosen by the Select statement.
qeSetSelectOptions	Specifies the following options: Whether your application only reads forward through the records resulting from a Select statement, or also needs to position to records that have already been read. Whether DTK will write records in the result set to log files when connected to databases for which it is not necessary to do so. The level of fetching that is possible after a transaction ends.
qeGetSelectOptions	Returns whether previous and random fetching is enabled for the current database connection, whether DTK will use log files when connected to databases for which it is not necessary to do so, and the level of fetching that is possible after a transaction ends.
qeFetchLogClose	Closes the log file used with DTK's fetching functions.
qeSetMaxRows	Sets the maximum number of rows that a statement will return.
qeGetMaxRows	Returns the maximum number of rows that a statement will return.

The qeFetchNext, qeFetchPrev, and qeFetchRandom functions retrieve a record from the database and make it the current record in DTK.

DTK lets you use two different techniques to read column values from databases.

- The first technique uses the qeBindCol functions to bind column values to variable buffers (see “Binding Data to Columns,” next).
- The second technique uses the qeVal functions to retrieve individual column values following each call to qeFetchNext (see “Using qeVal Functions” on page 43).

Because some database systems do not support the previous and random record fetching provided by qeFetchPrev and qeFetchRandom, DTK provides this capability when connected to such databases by saving the results of a Select statement in a log file. The qeSetSelectOptions function lets you set the level of fetching and log file usage that DTK provides to a specified connection. When DTK uses a log file, you can close the log file by calling qeFetchLogClose.

When the SQL statement is a Select statement or stored procedure, qeFetchNumRecs returns the number of records in the result set. You can use the qeFetchNumRecs function only when previous and random fetching is enabled. You can set a maximum number of records that a Select statement can return by calling qeSetMaxRows.

Binding Data to Columns

Use the qeBindCol functions to obtain maximum performance. Call the qeBindCol functions to bind variables in your program to each of the columns returned by the Select statement. Each subsequent call to a qeFetch function fills your variables with the column values. The maximum data size bound by qeBindCol functions is 64K.

Many macro and script languages, including Visual Basic, do not support the qeBindCol functions.

Table 4-2 lists the functions that bind data to columns.

Table 4-2. Functions that Bind Data to Columns

Function	Result
qeBindCol	Specifies value and length variables that receive a column's value and length each time a record is fetched.
qeBindColChar	Similar to qeBindCol. Data is converted to a character string, using a format string if supplied
qeBindColDecimal	Similar to qeBindCol. Data is converted to a decimal value with the specified precision and scale.
qeBindColDouble	Similar to qeBindCol. Data is converted to a double-precision floating-point value.
qeBindColFloat	Similar to qeBindCol. Data is converted to a single-precision floating-point value.
qeBindColInt	Similar to qeBindCol. Data is converted to a 2-byte integer.
qeBindColLong	Similar to qeBindCol. Data is converted to a 4-byte integer.

The qeBindCol function performs no data type conversion. Before calling qeBindCol, you can call qeColType to determine the data type of a column's values. The values put in your variables by qeFetchNext, qeFetchPrev, or qeFetchRandom will be of this type.

Use the other six qeBindCol functions as needed to convert the data type of the column.

You can also call qeColWidth before calling a qeBindCol function in order to determine the maximum size (in bytes) of a column's values. You can use this width to allocate variables large enough to hold the largest values.

When character or date-time values are retrieved by a qeFetch function, a zero terminator byte is added to the end of the values. This is the C-language convention supported by most macro languages.

For the character data types, the maximum size may be very large. The variable you bind to a column can be smaller than the maximum size. However, the length variable (pointed to by *len_ptr*) must contain the actual length of the variable you are binding. For example, you need to allocate a 21-byte variable to retrieve values from a column defined as VARCHAR (20).

Each time you call a qeFetch function, DTK compares the length of the column's value to the length of the variable you bound to the column. If the value is longer than the variable you bound, the value is truncated to the size of your variable and your length variable is set to qeTRUNCATION (-1). It is not necessary to set the length variable before calling a qeFetch function.

When a qeFetch function is called and a column's value is null, its length variable is set to qeNULL_DATA (-2).

Make all calls to qeBindCol functions before the first call to qeFetchNext, qeFetchPrev, or qeFetchRandom. Each time you call a qeFetch function, another record will be read and its values placed in the buffers specified by the calls to qeBindCol.

Important: When you use the qeBindCol functions, you must call a qeBindCol function for every column in the Select statement, in the order they occur in the statement. If you omit any columns, an error will be returned by your first call to a qeFetch function.

Using qeVal Functions

The qeVal functions retrieve column values following each call to qeFetchNext. The following example uses this method. To load this sample in the SAMPLE.EXE program, choose **Reading Records Using qeVAL** from the Example List.

The example on page 44 also shows how to call DTK functions in a loop to read all records in a database.

```
qeSTATUS valfetch ()  {

/* This routine demonstrates how to fetch data from SELECT statements using *     /
/* the qeVal functions. *  /

       qeHANDLE        hdbc = 0;       /* Handle to database connection *     /
       qeHANDLE        hstmt = 0;      /* Handle to SQL statement execution *     /
       qeSTATUS        res_code;       /* Result code from DTK functions *     /
       qeLPSTR         last_name  ;
       qeLPSTR         nameptr  ;
       float           salary  ;

/* Call qeLibInit to initialize DTK, check for errors. *    /
       res_code = qeLibInit ()  ;
       if (res_code != qeSUCCESS) return (res_code)    ;

/* Call qeConnect to connect to a data source.  Check to see *    /
/* if hdbc == 0, which indicates that the connection failed. *    /
       hdbc = qeConnect ("DSN=QEDBF")   ;
       if (hdbc == 0) return (err_handler (hdbc, hstmt))    ;

/* Call qeExecSQL to execute the select statement. If hstmt == 0, *    /
/* then the statement did not execute successfully. *    /
       hstmt = qeExecSQL (hdbc, "Select last_name, salary from emp")    ;
       if (hstmt == 0) return (err_handler (hdbc, hstmt))    ;

/* Fetch the records from the select statement, and get the individual *     /
/* column values. * /
       while (qeFetchNext (hstmt) == qeSUCCESS)     {

               nameptr = qeValChar (hstmt, 1, "", 0)    ;
               if (qeErr () != qeSUCCESS && qeErr () != qeNULL_DATA) break     ;
               MessageBox (hWnd, nameptr, "Val Fetch", MB_OK)    ;


               salary = qeValFloat (hstmt, 2)   ;
               if (qeErr () != qeSUCCESS && qeErr () != qeNULL_DATA) break     ;
        }

/* Check for errors, EOF is ok. *  /
       res_code = qeErr ()  ;
       if ((res_code != qeSUCCESS) && (res_code != qeEOF)    )
               return (err_handler (hdbc, hstmt))    ;
/* Close the SQL statement. *  /
       res_code = qeEndSQL (hstmt)   ;
```

```
        hstmt = 0 ;
        if (res_code != qeSUCCESS) return (err_handler (hdbc, hstmt))    ;

/* Call qeDisconnect to disconnect from a data source. *    /
        res_code = qeDisconnect (hdbc)  ;
        hdbc = 0 ;
        if (res_code != qeSUCCESS) return (err_handler (hdbc, hstmt))    ;

/* Call qeLibTerm to free memory allocated by DTK. *    /
        res_code = qeLibTerm ()  ;
        MessageBox (hWnd, "Sample succeeded.", "Val Fetch", MB_OK)    ;
        return (res_code) ;
}

/* err_handler routine goes here.  *   /
```

The qeVal functions that return values from the current record are listed in Table 4-3.

Table 4-3. Functions that Return Values from the Current Recdr

Function	Result
qeDataLen	Reports the length of a value retrieved by a qeVal function.
qeValChar and qeValCharBuf	Return a column's value as a character string.
qeValMultiChar and qeValMultiCharBuf	Return the values of multiple columns as a single character string.
qeValDecimal and qeValDecimalBuf	Return a column's value as a decimal number (BCD)
qeValInt	Returns a column's value as a 2-byte integer.
qeValLong	Returns a column's value as a 4-byte integer.
qeValFloat	Returns a column's value as a floating-point number.
qeValDouble	Returns a column's value as a double-precision floating-point number.

After calling a qeVal function, you can call qeDataLen to obtain the length in bytes (or characters) of the value.

Performance can be improved by retrieving more than one column at a time. You can call qeValMultiChar and qeValMultiCharBuf to simultaneously retrieve multiple column values.

The tradeoffs of using the qeBindCol functions versus the qeVal functions are discussed in "Comparing qeBindCol and qeVal Techniques" on page 46.

Comparing qeBindCol and qeVal Techniques

You cannot mix the two techniques for reading records. If you use qeBindCol functions, you cannot call any of the qeVal functions.

The advantages of using the qeBindCol functions are as follows:

- Records can be read faster.
- You need to call a qeBindCol function only one time for each column you are retrieving, as opposed to calling a qeVal function for each column every time you fetch another record. This greatly decreases the processing overhead so performance is improved.
- They allow the use of the qePutUsingBindColumns function.

The advantages of using the qeVal functions are as follows:

- They are easier to use from most macro and script languages. Some languages will not allow you to send a pointer to a 4-byte long integer variable as a parameter to a function as is required by the qeBindCol functions.
- The qeValChar and qeValCharBuf functions can return large values in pieces. If the maximum size of a column is very large, 60,000 characters for example, the qeValChar function lets you retrieve the value in smaller pieces—up to 1000 characters at a time. If you use qeBindCol functions, you must declare a variable of the maximum size you want to receive. If

you declare a variable smaller than the maximum size, you will not get the entire value.

Getting Column Information

The column definition functions allow you to get information about the columns returned by a Select statement. For example, if your Select statement is

```
SELECT * FROM em p
```

then you may not know the names, data types, or number of columns returned. The column definition functions allow you to get this information.

The following sample shows how to use the qeCol functions to get information about the columns returned by a Select statement. To load this sample in the SAMPLE.EXE program, choose **Getting Column Information** from the Example List.

```
qeSTATUS colinfo ()  {

/* This routine demonstrates how to use the qeCol functions functions to *    /
/* get information about the columns returned by a Select statement. *    /

       qeHANDLE        hdbc = 0;       /* Handle to database connection *    /
       qeHANDLE        hstmt = 0;      /* Handle to SQL statement execution *    /
       qeSTATUS        res_code;       /* Result code from DTK functions *    /
       short           col_count, col  ;
       qeLPSTR         col_name_ptr  ;

/* Call qeLibInit to initialize DTK, check for errors. *   /
       res_code = qeLibInit ()  ;
       if (res_code != qeSUCCESS) return (res_code)   ;

/* Call qeConnect to connect to a data source.  Check to see *    /
/* if hdbc == 0, which indicates that the connection failed. *    /
       hdbc = qeConnect ("DSN=QEDBF")  ;
       if (hdbc == 0) return (err_handler (hdbc, hstmt))   ;
```

```
/* Call qeExecSQL to execute the select statement. Check if hstmt == 0, *    /
/* which indicates that the statement did not execute successfully. *    /
       hstmt = qeExecSQL (hdbc, "Select * from emp")   ;
       if (hstmt == 0) return (err_handler (hdbc, hstmt))    ;
/* Get the column names returned by the Select statement *    /
       for (col_count = qeNumCols (hstmt), col = 1; col_count--; col++ )      {
               if (qeErr () != qeSUCCESS   )
                       return (err_handler (hdbc, hstmt))    ;

               col_name_ptr = qeColName (hstmt, col)   ;
               if (qeErr () != qeSUCCESS   )
                       return (err_handler (hdbc, hstmt))    ;
               MessageBox (hWnd, col_name_ptr, "Column Name", MB_OK)    ;
        }

/* Close the SQL statement. *  /
       res_code = qeEndSQL (hstmt)  ;
       hstmt = 0 ;
       if (res_code != qeSUCCESS) return (err_handler (hdbc, hstmt))    ;

/* Call qeDisconnect to disconnect from a data source. *    /
       res_code = qeDisconnect (hdbc)  ;
       hdbc = 0 ;
       if (res_code != qeSUCCESS) return (err_handler (hdbc, hstmt))    ;

/* Call qeLibTerm to free memory allocated by DTK. *    /
       res_code = qeLibTerm ()  ;
       MessageBox (hWnd, "Sample succeeded.", "Bind Fetch", MB_OK)    ;
       return (res_code)  ;
}

/* err_handler routine goes here.  *   /
```

Table 4-4 lists the qeCol functions that return information about the columns in the Select statement.

Table 4-4. Functions that Return Select-Statement Column Info

Function	Returns
qeNumCols	The number of columns in the statementThe number of columns in the statement
qeColName and qeColNameBuf	A column's name.
qeColAlias and qeColAliasBuf	A column's alias.
qeColExpr and qeColExprBuf	A column's expression.
qeColType	A column's data type.
qeColDBType	The database's native type for the requested column
qeColDBTypeName and qeColDBTypeNameBuf	The database's native type name for the requested column.
qeColWidth	A column's maximum width in bytes.
qeColPrecision	A decimal column's precision.
qeColScale	A decimal column's scale.
qeColTypeAttr	Whether a column is updatable, nullable, searchable, unsigned, autoincremented, or type Money, depending on the attribute you specify.
qeColDateStart	A date-time column's starting offset
qeColDateEnd	A date-time column's ending offset

Each function has an *hstmt* parameter to identify the SQL statement. All functions except qeNumCols also have a column number as a parameter identifying the column whose information is to be returned.

qeColName returns the column name, unless the column is an expression. If the column is an expression, qeColName returns null.

qeColAlias returns an alias if one exists, otherwise it returns null.

qeColExpr returns an expression if there is one, otherwise it will return the same value for the column as would qeColName.

Converting Data Types

Each column in a table has a *data type*. The data type determines the type of information that can be stored in the column: character strings, integer numbers, floating-point numbers, dates, etc. See "Data Types in DTK" on page 53 for more information on data types.

DTK provides a number of data type conversion functions. These functions, discussed in detail in Appendix A, "Data Conversion Functions," on page 493, allow you to convert values from any of the eight standard data types to any other data type.

In addition to converting data types, the functions listed in Table 4-5 can be used to format numbers and date-time values into character strings, and convert character string values to numbers or dates.

Table 4-5. Functions that Convert Data Types

Function	Converts
qeBinToHex qeBinToHexBuf	Binary value to hexadecimal value.
qeCharToDate qeCharToDateBuf	Character string to date.
qeCharToDecimal qeCharToDecimalBuf	Character string to decimal numbers.

Table 4-5. Functions that Convert Data Types (cont.)

Function	Converts
qeCharToDouble	Character string to double-precision floating-point number.
qeCharToFloat	Character string to floating-point number.
qeCharToInt	Character string to 2-byte integer.
qeCharToLong	Character string to 4-byte integer.
qeDateToChar qeDateToCharBuf	Date to character string.
qeDateToDouble	Date to double-precision Julian value.
qeDateToLong	Date to 4-byte integer Julian value.
qeDecimalToChar qeDecimalToCharBuf	Decimal number to character string.
qeDecimalToDouble	Decimal number to double-precision floating-point number.
qeDecimalToFloat	Decimal number to floating-point number.
qeDecimalToInt	Decimal number to 2-byte integer.
qeDecimalToLong	Decimal number to 4-byte integer.
qeDoubleToChar qeDoubleToCharBuf	Double-precision floating-point number to character string.
qeDoubleToDecimal qeDoubleToDecimalBuf	Double-precision floating-point number to decimal number.
qeDoubleToFloat	Double-precision floating-point number to floating-point number.
qeDoubleToInt	Double-precision floating-point number to 2-byte integer.
qeDoubleToLong	Double-precision floating-point number to 4-byte integer.
qeFloatToChar qeFloatToCharBuf	Floating-point number to character string.
qeFloatToDecimal qeFloatToDecimalBuf	Floating-point number to decimal number.

Table 4-5. Functions that Convert Data Types (cont.)

Function	Converts
qeFloatToDouble	Floating-point number to double-precision floating-point number.
qeFloatToInt	Floating-point number to 2-byte integer.
qeFloatToLong	Floating-point number to 4-byte integer.
qeHextoBin qeHexToBinBuf	Hexadecimal value to binary value.
qeIntToChar qeIntToCharBuf	Integer to character string.
qeIntToDecimal qeIntToDecimalBuf	Integer to decimal number.
qeIntToDouble	Integer to double-precision floating-point number.
qeIntToFloat	Integer to floating-point number.
qeIntToLong	2-byte integer to 4-byte integer.
qeLongToChar qeLongToCharBuf	4-byte integer to character string.
qeLongToDecimal qeLongToDecimalBuf	4-byte integer to decimal number.
qeLongToDouble	4-byte integer to double-precision floating-point number.
qeLongToFloat	4-byte integer to floating-point number.
qeLongToInt	4-byte integer to 2-byte integer.

Those functions that convert to or from character strings include a format string parameter to control formatting. See "Format Strings" on page 59 for more information.

The following section describes how DTK handles data types.

Data Types in DTK

Different database systems support different data types for their columns. DTK maps the various data types into one of eight standard data types:

Identifier	Data Type
1	Fixed-length character string*
2	character string*
3	Decimal number (BCD)
4	Long integer (4-byte)
5	Integer (2-byte)
6	Single-precision floating-point numbers (4-byte)
7	Double-precision floating-point numbers (8-byte)
8	Date-time (26-byte character string)

* These data types can also be used for binary data. See “Blobs and Memos” on page 57 for more information.

DTK returns all values as one of these eight data types.

If you call qeExecSQL to execute a Select statement such as

```
SELECT last_name, salary, hire_date FROM em    p
```

you can use the qeColType function to get the data type of each of the columns being returned. LAST_NAME’s data type will be 1 or 2 since it is a character string, SALARY may be any type from 3 to 7 depending on how it is stored, and HIRE_DATE will be type 8.

You sometimes need to know the exact data type used in the underlying database system. Database systems support data types that are variations of one of the eight standard data types. Some database systems include logical data types, binary strings, money, etc. In each case, DTK automatically converts the data to one of the eight data types. To determine the database's data type for a column, use qeColDBType, qeColDBTypeName, or qeColDBTypeNameBuf. These functions return the native data type for one column in a SQL Select statement. The *DataDirect ODBC Drivers Reference* lists the native data types of each system.

Fixed and Variable Character Strings

The difference between fixed (type 1) and variable (type 2) character string data types is whether trailing blanks are added to values. For example, suppose a LAST_NAME column is declared with a limit of 12 characters (bytes) and the name Smith is stored. If the column is fixed length, the value is returned as `'Smith       '` (Smith followed by 7 blanks). If the column is variable length, the value is returned as `'Smith'` with no trailing blanks. Both types of character string are terminated with a zero-terminator character (a C language convention).

Date-Time Values

Date-time values are 26-byte character strings having the following format:

```
YYYY-MM-DD HH:MM:SS.SSSSS   S
```

Hour values are expressed in terms of a 24-hour clock. See "Binary and Date-Time Constants" on page 56 for information on handling date-time values with DTK.

For date-time columns, the qeColDateStart and qeColDateEnd functions return offsets identifying the relevant part of the date-time value. For example, if the column contains date values with no time, qeColDateStart returns 0 and qeColDateEnd returns 9, indicating that only the first 10 characters in the date-time value are relevant. The only combinations of values returned by these functions are as follows.

See Appendix A, “Data Conversion Functions,” on page 493 for complete descriptions of these functions.

Value	qeColDateStart	qeColDateEnd
Date-Time	0	15, 18, 22, or 25
Date	0	9
Time	11	18, 22, or 25

Decimal Number Format

Many database systems store numbers using a proprietary decimal format. DTK retrieves these numbers and converts them, if necessary, into a standard decimal format. DTK uses a Binary-Coded Decimal (BCD) format.

The BCD format stores two digits per byte. In each byte, the first digit is in the top 4 bits of the byte, and the second digit in the lower 4 bits. The sign of the number is stored in the lower 4 bits of the last byte. The hexadecimal value 0xC is the sign for positive numbers, and 0xD is the sign for negative numbers.

Decimal numbers are defined by their *precision* and *scale*. Precision is the number of digits that can be stored in the number. You can determine the length in bytes of a decimal number by its precision. The formula is

```
bytes = (precision+2)/  2
```

If a decimal number has an even precision, the upper 4 bits of the first byte are not used, and the first digit is in the lower 4 bits of the first byte. In all cases, the last byte contains the last digit in the upper 4 bits and the sign in the lower 4 bits.

The *scale* specifies the actual position of the decimal point in the number. The scale is the number of digits to move the decimal point to the left of the sign. You can also think of scale as the number of digits right of the decimal point.

For example, if the precision=4, scale=2, value=12.34, then the bytes contain the following hexadecimal values:

```
01 23 4 C
```

The qeColPrecision and qeColScale functions return the precision and scale of decimal columns.

Binary and Date-Time Constants

Database systems vary as to how you specify date-time constants and binary constants in SQL statements. For example, to compare date values in a Where clause, dBASE uses

```
hire_date > {01/27/95  }
```

and Oracle uses

```
hire_date > to_date('01/27/95','MM/DD/YY'    )
```

If your applications need to access more than one database system, these differences can cause problems.

DTK supports database-independent syntax for date-time and binary constants so you don't have to modify your programs for the different database systems.

The database-independent syntax for the date example is

```
hire_date > [d'1995-01-27 00:00:00.000000'    ]
```

The constant is enclosed in square brackets. The letter code, which is case-sensitive, indicates the data type. The letter code is followed by a character string enclosed in single quotes. The following table lists the codes:

Code	Data Type
d	Date
t	Time
dt	Date and time
b	Binary

For date-time codes, the character string must be in the full 26-character standard date format described earlier in "Date-Time Values" on page 54. All 26 characters must be present whether the code is d, t, or dt.

For the binary code, the character string must be the binary value represented as a hexadecimal string. For example,

```
UPDATE tab SET binvar=[b'0F4A512A8C']
WHERE prikey = 11  1
```

Blobs and Memos

Many databases support a binary data type designed for storing large amounts of text or image data. These data types are frequently called memos or blobs. Although DTK does not provide functions specifically designed for retrieving and writing such data, you can do so using repeated calls to DTK functions.

There are two methods you can use to read binary data:

- *Use qeValChar or qeValCharBuf*. These functions read data in chunks of up to 64K (actually, 65280 bytes). Whenever these functions fail to read all of the available data in a column, qeDataLen returns qeTRUNCATION (–1). Another call to the qeVal function will return the next piece of data. By using a loop that checks for truncation after each call to one of these functions, you can easily read very large values.

- *Use qeBindColChar*. Because you cannot call this function more than once for a single value, you are limited to 64K as the maximum size of the value returned. However, this function enables you to fetch multiple binary values under the 64K limit without repeated calls to the qeBindCol functions.

The reason that there are no equivalent functions for binary data types is that the only difference would be the absence of the zero byte that terminates the data being read.

For writing binary data, two methods are available:

- *Use SQL parameters*. For example, if you had a long, free-form text column called INTERESTS stored as binary data, you could issue the following statement with qeExecSQL:

  ```
  UPDATE emp SET interests = ?
  WHERE emp_id = D10 1
  ```

  You could then write the value with the qeBindParamBinary or qeSetParamBinary function. You cannot write values larger than 64K.

- *Use current record functions*. When you have an open Select statement, you can change the column value using the qePutBinary and qeRecUpdate functions. Again, you cannot write values larger than 64K.

Because all of these methods use functions that handle a maximum value length of 64K (65280 bytes), that is the maximum value size you can handle except when reading values with repeated calls to qeValChar or qeValCharBuf, in which case there is no limit.

Null Values

Many database systems have the concept of a null value. A null value for a database column means that the record contains no value for this column. When you retrieve a value, you cannot determine if the value is null. For example, qeValInt always returns a valid integer value, since every possible value is valid.

To determine if a qeVal function returned a null value, you must call qeDataLen or qeWarning. qeDataLen returns the length of the value returned by the qeVal function in bytes (or characters). If qeDataLen or qeWarning return qeNULL_DATA (-2), then the value returned by the qeVal function was null.

Logical Values

Some database systems support logical (true/false) data types. DTK returns values of this type as numbers: 0 for False, and 1 for True.

Format Strings

When you use qeValChar or qeValCharBuf to get column values as character strings, or when you use the data conversion functions to convert values to character strings (like qeDoubleToChar, qeLongToChar), DTK allows you to specify a format string that is used to format the value.

Also, when you use the data conversion functions to convert a character string to a numeric value, DTK allows you to specify a format string to show how the character string is formatted.

The following table shows some examples of format strings that are described in the following sections.

Format String	Value	Formatted Value
$#,##0.00	100.5	$100.50
	0	$0.00
	2500.25	$2,500.25
	-145.10337	-$145.10
$#,##0.00;($#,##0.00)	100.50365	$100.50
	-145.10	($145.10)

Format String	Value	Formatted Value
$#,##0.00"CR";$#,##0.00"DB"	1125.9	$1,125.90CR
	-2500	$2,500.00DB
0[S/1000]	12375	12
	199	0
GN	147	147
	1.875	1.875
mm/dd/yy	Jan 15, 1996	01/15/96
mm/dd/yyyy	Jan 9, 1996	01/09/1996
m/d/yy	Jan 9, 1996	1/9/96
dd.mm.yy	Jan 9, 1996	09.01.96
Mmm d, yyyy	Jan 9, 1996	Jan 9, 1996
dd-MMM-yy	Jan 9, 1996	09-JAN-96
Mmmm d, yyyy	Jan 9, 1996	January 9, 1996
hh:mm:ss	4:53:10 PM	16:53:10
hh:mm:ss AM/PM	4:53:10 PM	04:53:10 PM
mm/dd/yy hh:mm:ss	Jan 9, 1996 9:43	01/09/96 09:43:00

Numeric Format Strings

Format strings allow you to format numeric values with dollar signs, thousand separators, scientific notation, percents, etc. You can format positive numbers and negative numbers differently.

Numeric format strings can have one or two sections, separated by a semicolon. If the format string has one section, then positive and negative values use the same format. A negative sign is automatically inserted for negative numbers. If there are two sections, the first section is for positive numbers and the second for negative numbers.

The symbols in the format strings determine the way the values are to be formatted. Some of the symbols refer to strings specified in the International section of the Control Panel. You can change these strings by running the Control Panel program provided with Windows or OS/2. In the Control Panel, click on the International icon to see and change these strings.

Format String	Value	Formatted Value
0.00	100.5 -145.1	100.50 -145.10
0.00;(0.00)	100.5 -145.1	100.50 (145.10)

The following table describes the symbols allowed in a numeric format string.

Symbol	Description
$	Output the currency string. The currency string is specified in the International section of the Control Panel.
.	Output the decimal point character. The decimal point character is specified in the International section of the Control Panel.
,	Output the thousand's separator character. The thousand's separator character is specified in the International section of the Control Panel.
#	Output a digit. If there is no digit to output in the position, output nothing. For example, if the format string is "`###.##`", 12.3 is formatted as "`12.3`", 125.22475 is formatted as "`125.22`", 0 is formatted as "`.`", and 1500 is formatted as "`1500.`". **Note:** If the value has more digits to the left of the decimal than there are symbols in the format string, the format string is automatically extended to the left. However, if the value has more digits to the right of the decimal point than appear in the format string, the value is rounded into the last digit.

Symbol	Description
0	Output a digit. If there is no digit to output in the position, output a zero. For example, if the format string is "`000.00`", 12.3 is formatted as "`012.30`", 125.22475 is formatted as "`125.22`", 0 is formatted as "`000.00`", and 1500 is formatted as "`1500.00`". **Note:** See note for “#” symbol.
?	Output a digit. If there is no digit to output in the position, output a space character. For example, if the format string is "`???.??`", 12.3 is formatted as "` 12.3 `", 125.22457 is formatted as "`125.22`", 0 is formatted as "`   .  `", and 1500 is formatted as "`1500.  `" **Note:** See note for “#” symbol.
%	Output the value as a percent. The value is multiplied by 100 and the percent character (%) is output. For example, if the format string is "`#0%`", 0.15 is formatted as "`15%`".
e+ e-	Output using scientific notation. e+ outputs the sign of the exponent only if it is negative, e- always outputs the sign of the exponent. For example, if the format string is "`0.00e+#0`", the value 12500 is formatted as "`1.25e04`", and .005 is formatted as "`5.00e-03`". If the format string is "`0.00e-#0`", the value 12500 is formatted as "`1.25e+04`", and the value .005 is formatted as "`5.0e-03`". **Note:** You can also use E+ or E- in the format string. This causes the “E” to be uppercase in the formatted value.

Symbol	Description
–+() space	Output plus or minus signs, parentheses, or blank spaces. These characters are often used to distinguish positive and negative values. For example, if the format string is "`+0.00;–0.0 0`", 12.3 is formatted as "`+12.3 0`", and –1.1 is formatted as "`–1.10`". Blank spaces are output in the position you specify. **Note:** These are the only characters that can be included in numeric format strings to be output directly. To output other characters or strings, use the "\" symbol or enclose the characters in quotation marks.
\	Output the character following the backslash. For example, if the format string is "`0.00 \t\o\n\ s`", the value 1.25 is formatted as "`1.25 ton s`".
"*string*"	Output the string. The quotation marks are not output. For example, if the format string is "`0.00 "tons"`"", the value 1.25 is formatted as "`1.25 ton s`".
'*string*'	Output the string. The quotation marks are not output. For example, if the format string is "`0.00 'tons '`", the value 1.25 is formatted as "`1.25 ton s`".
GN	General format for numbers. This is the format used if no format string is given. For example, if the format string is "`GN`", 12.3 is formatted as "`12.3`", 125.22475 is formatted as "`125.2247 5`", 0 is formatted as "`0`", and -1500 is formatted as "`-150 0`". **Note:** If you use GN, the only other symbols you can use in the format string are those enclosed in brackets; for example, [US].
GF	General fixed format for numbers. The "Number Format" in the International section of the Control Panel is used. **Note:** If you use GF, the only other symbols you can use in the format string are those enclosed in brackets; for example, [US].

Symbol	Description
GC	General currency format for numbers. The "Currency Format" in the International section of the Control Panel is used. **Note:** If you use GC, the only other symbols you can use in the format string are those enclosed in brackets; for example, [US].
[S/n] [S*n]	Scale the number before it is output. "`[S/n]`" divides the number by ‘n’ before it is formatted. "`[S*n]`" multiplies the number by ‘n’ before it is formatted. ‘n’ must be a power of 10 (10, 100, 1000, etc.). For example, if the format string is "`#0.00[S/1000]`", 12340 is formatted as "`12.34`".
[US]	The information specified in the International section of the Control Panel is ignored. Instead, the United States defaults are substituted (periods for decimal points, commas for thousand separators, and $ for the currency symbol). For example, if the format string is "`$#,##0.00[US]`", 1234.56 is formatted as "`$1,234.56`", regardless of the International settings in the Control Panel.

Date-time formats allow you to control which parts of the date or time are to be output, their order, and whether to spell out months and days.

The following table describes the symbols allowed in a date-time format string.:

Symbol	Description
m mm	Output the month's number (1-12). If the month's number is less than 10, "m" does not output the leading 0, and "mm" outputs the leading 0.
mmm	Output the month's three-letter abbreviation. Whether the M's are upper or lowercase determines whether the abbreviation is upper or lowercase: mmm jan Mmm Jan MMM JAN
mmmm	Output the month's full name. Whether the M's are upper or lowercase determines whether the name is upper or lowercase: mmmm january Mmmm January MMMM JANUARY
d dd	Output the day of the month's number (1-31). If the day's number is less than 10, "d" does not output the leading 0, and "dd" outputs the leading 0.

Symbol	Description
ddd	Output the day of the week's three-letter abbreviation. Whether the D's are upper or lowercase determines whether the abbreviation is upper or lowercase: ddd sun Ddd Sun DDD SUN
dddd	Output the day of the week's full name. Whether the D's are upper or lowercase determines whether the name is upper or lowercase: dddd sunday Dddd Sunday DDDD SUNDAY
yy yyyy	Output the year's number. For "yy," only the last two digits of the year are output. For "yyyy," the four-digit year is output.
h hh	Output the hour of the day (0-23). If the hour's number is less than 10, "h" does not output the leading 0, and "hh" outputs the leading 0. **Note:** Whether a 12-hour or 24-hour clock is used depends on whether the "AM/PM" symbol is used.

Symbol	Description
m mm i ii	Output the minute of the hour (0-59). You can use “m” or “i” for minute. If you use “m,” the previous date-time component must be an hour symbol to avoid confusion with the month symbol. If the minute’s number is less than 10, “m” or “i” do not output the leading zero, and “mm” or “ii” outputs the leading 0.
ss.ssssss	Output the second of the hour (0-59). You can use one or two “s” symbols to the left of the decimal point. If one “s” is used, a leading zero is not output for seconds less than 10. The decimal point and the “s” symbols to the right of the decimal point are optional. They are used to output fractions of seconds. You can use up to 6 “s” symbols to the right of the decimal.
am/pm a/p	Output the “am” or “pm” string. These strings are specified in the International section of the Control Panel. Whether the symbol is upper or lowercase determines whether the string is upper or lowercase: am/pm “am” or “pm” is output AM/PM “AM” or “PM” is output You can also use the symbol “a/p.” This causes the first letter of the strings to be output. If you use a/p, “a” or “p” is output. With A/P, “A” or “P” output. **Note:** If this symbol is used, a 12-hour clock is assumed. The hour symbols output hour numbers between 1 and 12.

Symbol	Description
J	Output the Julian value for the date-time. The Julian value is a numeric value giving the date as the number of days since 4712 BC, and the time as a fraction of a day.
/ - . : , space	Output the character. These characters are used to separate the parts of a date or time. **Note:** These are the only characters that can be included in date format strings to be output directly. To output other characters or strings, use the “\” symbol or enclose the characters in quotation marks
\	Output the character following the backslash. For example, if the format string is "`hh:mm:ss \G\M\ T`", the value 10:05:12 AM is formatted as "`10:05:12 GM T`".
"*string*"	Output the string. The quotation marks are not output. For example, if the format string is "`hh:mm:ss  "GMT"`"", the value 10:05:12 AM is formatted as "`10:05:12 GM T`".
'*string*'	Output the string. The quotation marks are not output. For example, if the format string is "`hh:mm:ss 'GMT '`", the value 10:05:12 AM is formatted as "`10:05:12 GM T`".
GD	General format for dates. This is the format used if no format string is given. The “Short Date Format” in the International section of the Control Panel is used. **Note:** The only other symbols you can use with GD are those enclosed in brackets; for example, [US].

Symbol	Description
GDT	General format for dates with times. The “Time Format” in the International section of the Control Panel is appended to the “Short Date Format.” **Note:** The only other symbols you can use with GDT are those enclosed in brackets; for example, [US].
GL	General long format for dates. The “Long Date Format” in the International section of the Control Panel is used. **Note:** The only other symbols you can use with GL are those enclosed in brackets; for example, [US].
GLT	General long format for dates with times. The “Time Format” in the International section of the Control Panel is appended to the “Long Date Format.” **Note:** The only other symbols you can use with GLT are those enclosed in brackets; for example, [US].
GT	General format for time. The “Time Format” in the International section of the Control Panel is used. **Note:** Do not combine any other formatting symbols with GT.
[US]	The information specified in the International section of the Control Panel is not used. Instead, the United States defaults are substituted. You should use this symbol only with GD, GDT, GL, or GLT. For example, if the format string is "GD[US]", July 15, 1995, is formatted as "07/15/95".

Go To

5 Modifying Data

Once you have executed a SQL Select statement, DTK lets you position to individual records and update or delete the current record, or insert new records. This method of modifying the current record is often more convenient than having to generate the appropriate SQL Insert, Update, or Delete statement. This chapter describes the column (qePut) and record (qeRec) functions that perform current-record operations in DTK, and discusses DTK's use of unique keys in performing these operations.

Current-Record Functions

After you execute a SQL Select statement, you can use the qeFetch functions to position to specific records. The record you are positioned on is called the current record.

Two sets of functions affect the current record. The qePut functions assign new values to the individual columns of the current record. The qeRec functions modify or get information about the current record.

The sample on page 72 shows the use of the qePut and qeRec functions, which operate on the current record of an *hstmt* resulting from executing a Select statement. The sample uses these functions to insert, update, and delete records. The base Select statement for this example is

```
SELECT first_name, last_name FROM emp
```

In the example, the first record is read and its first name value is changed to "Gerald," the second record is deleted, and a new record employee record is inserted before the first record.

To load this sample in the SAMPLE.EXE program, choose **Using Current Record Operations** from the Example List.

```
qeSTATUS recordop () {

/* This routine demonstrates the use of the qeRec functions.  These *    /
/* functions operate on the current record of an hstmt resulting from *    /
/* executing a Select statement. *  /

       qeHANDLE        hdbc = 0;       /* Handle to database connection *    /
       qeHANDLE        hstmt = 0;      /* Handle to SQL statement execution *    /
       qeSTATUS        res_code;       /* Result code from DTK functions *    /

/* Call qeLibInit to initialize DTK, check for errors *   /
       res_code = qeLibInit ()  ;
       if (res_code != qeSUCCESS) return (res_code)   ;

/* Call qeConnect to connect to a data source.  Check to see *    /
/* if hdbc == 0, which indicates that the connection failed. *    /
       hdbc = qeConnect ("DSN=QEDBF")  ;
       if (hdbc == 0) return (err_handler (hdbc, hstmt))   ;

/* Call qeExecSQL to execute the select statement. Check if hstmt == 0, *    /
/* which indicates that the statement did not execute successfully. *    /
       hstmt = qeExecSQL (hdbc, "Select first_name, last_name from emp")    ;
       if (hstmt == 0) return (err_handler (hdbc, hstmt))   ;

/* Position to the first record. *  /
       res_code = qeFetchNext (hstmt)  ;
       if (res_code != qeSUCCESS) return (err_handler (hdbc, hstmt))    ;

/* Update the employee's first name to Gerald. *   /
       res_code = qePutChar (hstmt, 1, "", "Gerald")   ;
       if (res_code != qeSUCCESS) return (err_handler (hdbc, hstmt))    ;
       res_code = qeRecUpdate (hstmt)  ;
       if (res_code != qeSUCCESS) return (err_handler (hdbc, hstmt))    ;

/* Position on second record and delete it. *   /
       res_code = qeFetchNext (hstmt)  ;
       if (res_code != qeSUCCESS) return (err_handler (hdbc, hstmt))    ;
       res_code = qeRecDelete (hstmt)  ;
       if (res_code != qeSUCCESS) return (err_handler (hdbc, hstmt))    ;

/* Create a new record, make it record number 2 in the result set. *    /
/* Set the new employee's name to Ed Allen. The call to qeRecUpdate *    /
/* will insert the record into the database table. *   /
       res_code = qeRecNew (hstmt, 2)  ;
       if (res_code != qeSUCCESS) return (err_handler (hdbc, hstmt))    ;
```

```
       res_code = qePutChar (hstmt, 1, "", "Ed")   ;
       if (res_code != qeSUCCESS) return (err_handler (hdbc, hstmt))    ;

       res_code = qePutChar (hstmt, 2, "", "Allen")   ;
       if (res_code != qeSUCCESS) return (err_handler (hdbc, hstmt))    ;

       res_code = qeRecUpdate (hstmt)  ;
       if (res_code != qeSUCCESS) return (err_handler (hdbc, hstmt))    ;

/* Close the statement. * /
       res_code = qeEndSQL (hstmt)  ;
       hstmt = 0 ;
       if (res_code != qeSUCCESS) return (err_handler (hdbc, hstmt))    ;

/* Call qeDisconnect to disconnect from a data source. *    /
       res_code = qeDisconnect (hdbc)  ;
       hdbc = 0 ;
       if (res_code != qeSUCCESS) return (err_handler (hdbc, hstmt))    ;

/* Call qeLibTerm to free memory allocated by DTK. *   /
       res_code = qeLibTerm ()  ;
       MessageBox (hWnd, "Sample succeeded.", "Current Record", MB_OK)    ;
       return (res_code)  ;
}

/* err_handler routine goes here.  *  /
```

This sample also shows how to use the qePutChar, qeRecNew, qeRecUpdate, and qeRecDelete functions to modify the database.

The following sections provide information on these and other current-record functions.

Column Functions

Once your application has positioned to a record using the qeFetch functions, you can change the values of the columns of the current record using the qePut functions. Table 5-1 lists the qePut functions.

Table 5-1. Functions that Change Column Values in the Current Record

Function	Result
qePutBinary	Updates a column with a binary value.
qePutChar	Updates a column with a character value.
qePutDecimal	Updates a column with a decimal value.
qePutDouble	Updates a column with a double-precision floating-point value.
qePutFloat	Updates a column with a floating-point value.
qePutInt	Updates a column with a 2-byte integer.
qePutLong	Updates a column with a 4-byte integer.
qePutNull	Updates a column to have the value null.
qePutUsingBindColumns	Updates columns with the values placed in bind buffers by the qeBindCol functions.

qePut functions take as arguments the *hstmt* of the active SQL Select statement, the number of the column being updated, and the new value. qePut functions change the values in the current record buffer but not the values in the database. In order to modify the database, you must first modify the values with qePut functions and then call qeRecUpdate. To insert a new record, first call qeRecNew to clear the field values for the new record, use the qePut functions to assign values to the new record's columns, then call qeRecUpdate to add the new record to the database.

Whenever you move off of the current record, the auto-updating options that are set via the qeSetAutoUpdate function affect what happens to values you have changed using the qePut functions. See the following section for more information.

Record Functions

Once your application has positioned to a record using the qeFetch functions, you can perform operations on the current record using the qeRec functions. Table 5-2 shows the set of qeRec functions.

Table 5-2. Functions that Operate on the Current Record

Function	Result
qeRecNew	Creates a new record that can be inserted by a call to qeRecUpdate.
qeRecUpdate	Updates or inserts a record with the new values set using qePut functions.
qeRecDelete	Deletes the current record.
qeRecUndo	Discards all changes to a record that have not been sent to the database.
qeRecState	Returns the state of the current record.
qeRecLock	Locks the current record during a transaction.
qeSetLockOptions	Controls the locking behavior for a statement.
qeGetLockOptions	Returns the locking behavior in effect for a statement.
qeRecNum	Returns the current record number.
qeRecSetKey	Declares whether the specified column is part of a unique key for a record.
qeRecGetKey	Reports whether a column is part of the unique key.
qeSetAutoUpdate	Determines what happens when the *hstmt* is moved to a new record before changed values have been updated or inserted.

Table 5-2. Functions that Operate on the Current Record (cont.)

Function	Result
qeGetAutoUpdate	Returns the auto update setting specified in the last call to qeSetAutoUpdate.
qeApplyAll	Updates all records that have not been explicitly updated by calls to qeRecUpdate.
qeUndoAll	Discards changes to all records that have not been explicitly updated by calls to qeRecUpdate.

The qeRec functions require an active SQL Select statement and therefore have an *hstmt* as a parameter. You can activate a Select statement by calling qeExecSQL or by calling qeSQLPrepare followed by qeSQLExecute. The qeRec functions operate on the current record. The current record is determined by the most recent call to a qeFetch function, qeRecNew, or qeRecFind.

To insert a new record, call qeRecNew to clear the field values and make the new record the current record, call the qePut functions to set the values in the new record, and then call qeRecUpdate to insert the record into the database.

To update a record, call the qePut functions to change the value of one or more columns, and then call qeRecUpdate to update the record in the database.

To delete a record, call qeRecDelete to delete the current record from the database.

When inserting or updating a record, if you call qeRecUndo before the call to qeRecUpdate, all of the changes to the column values made by the qePut functions will be undone. If qeRecNew has been called to create a new record, the new record will be discarded.

qeRecState returns the state of the current record. The state indicates whether the current record is a record read from the database or a new record, and whether one or more column values have been changed by calls to the qePut functions.

qeRecState also indicates whether DTK is currently positioned on a record or between records. qeRecState returns a state of qeSTATE_NOREC whenever the current position is between records. This state occurs following calls to qeRollback and after encountering EOF conditions. When in this state, record-oriented operations (qePut, qeRecUpdate, etc.) fail until you call a qeFetch function to reposition DTK on a valid record. qeRecNum returns a valid record number during this state, so you should always call qeRecState before calling qeRecNum to ensure proper record positioning.

qeRecLock obtains a shared lock on the current record. This function can only be used if a transaction has been started by a previous call to qeBeginTran. The shared lock is held until the transaction ends by a call to qeCommit or qeRollback.

qeRecNum returns the current record number. Each record retrieved from a Select statement is assigned a record number starting with 1. You can position to a record by calling qeFetchRandom and specifying the desired record number.

qeRecSetKey and qeRecGetKey specify which columns of the Select statement are to be used to identify the current record in the database. If you call qeRecLock, qeRecUpdate, or qeRecDelete, first call qeRecSetKey on the columns of your Select statement that together uniquely identify each record. See “Unique Keys” on page 78 for more information.

Use qeSetAutoUpdate to specify what DTK does when you move off of the current record before its values are updated in the database; that is, when the current record—which was create via qeRecNew or changed using the qePut functions—is not updated via qeRecUpdate before the user changes the current record. If qeSetAutoUpdate is set to qeAUTOUPD_UPDATE (3), then DTK automatically performs the qeRecUpdate function when the current record changes. If qeSetAutoUpdate is set to qeAUTOUPD_DEFER (2), then DTK saves the changes to the current record—not the database—before moving to another record. If qeSetAutoUpdate is set to qeAUTOUPD_DISCARD (1), DTK discards all changes made when you move off of a record that has not been updated. The qeGetAutoUpdate function returns which option is set.

If qeSetAutoUpdate is set to qeAUTOUPD_DEFER, you can use the DTK *hstmt* as a temporary record storage. For example, you can create several records by calling qeRecNew and set their column values by calling qePut functions. Or, you can modify a number of records by positioning to them using the qeFetch functions and changing them by calling qePut functions. You can position to any record by calling the qeFetch functions, and the new records and the changed records will be maintained by DTK, but the changes are not sent to the database until you call qeRecUpdate. When you position to a record, you can use the qeRecState function to determine whether it is new or has been changed. You can call qeApplyAll to apply all of the changed records to the database. You can call qeUndoAll to discard all of the changes made to all records.

Unique Keys

The qeRecSetKey and qeRecGetKey functions identify the columns of the Select statement that are used to uniquely identify the current record in the database.

For some database systems, DTK generates SQL Update and Delete statements to perform the qeRecUpdate and qeRecDelete functions. SQL statements may also be generated to perform the qeRecLock function. In these cases, DTK must generate a Where clause that uniquely identifies the record in the database corresponding to the current record.

To generate these Where clauses, DTK adds a condition for each column that you designate as a key by calling qeRecSetKey.

For example, in an employee table containing a unique employee ID field (EMP_ID), you can designate the employee ID field as the key field by calling qeRecSetKey. Then when DTK needs to generate a Where clause to identify a record, it generates a Where clause of the form “Where EMP_ID=*xxxxx*”, where *xxxxx* is the employee ID value for the current record. If there is no employee ID field in the employee table, you could specify both the LAST_NAME and FIRST_NAME columns as key columns. In this case DTK uses both values in the Where clause to identify the current record.

If you do not call qeRecSetKey and DTK needs to generate a Where clause to find the current record, it will create a default key that includes all of the columns in the Select statement that can be used. Depending on the columns in the Select statement, the generated Where clause may or may not uniquely identify the current record. For example, if the Select statement is

```
SELECT last_name FROM em  p
```

then the only field available to include in the Where clause is LAST_NAME. Since last names are typically not unique, the Where clauses will not uniquely identify records. Depending on the database system, some data types may not be allowed in Where clauses. DTK will not generate Where clauses containing columns that are not allowed in Where clauses.

DTK does not generate a default key until you call qeRecDelete, qeRecUpdate, qeRecLock, or qeUniqueWhereClause, function for the current *hstmt*. Until you call one of these functions (or qeRecSetKey), there will be no key for the *hstmt*—every column will return 0 (False) on calls to qeRecGetKey. If you call qeRecSetKey to set the key before calling one of the other “default key” functions listed above, the default key is never generated; instead, the key you specified is used.

Because DTK cannot guarantee that the Where clauses generated for qeRecUpdate or qeRecDelete uniquely identify one record, these calls may in fact affect more than one record, or no records. Your application should call qeNumModRecs to determine the number of records affected.

Go To

6 Using Transaction Functions

This chapter describes functions that let you group database operations into transactions. It also describes the functions that set the fetching, logging, and locking options that DTK provides. It contains the following sections:

- "Transaction Functions," next, describes the DTK functions used to implement transactions.
- "Transactions, Locking, and Logging" on page 84 describes the concept of transactions and many important concepts related to locking and logging within transactions. If you are not familiar with these concepts, you should read this section first.

Transaction Functions

DTK provides functions that let you group sets of database changes into *transactions*. A transaction is a set of database operations that can be committed or rolled back (undone) as a single unit.

The sample program on page 81 shows the use of transactions to roll back changes made by an SQL Update statement. To load this sample in the SAMPLE.EXE program, choose **Using Transactions** from the Example List.

```
qeSTATUS trans ()  {

/* This routine demonstrates the use of transactions to rollback changes *     /
/* made by an SQL Update statement. *   /

       qeHANDLE        hdbc = 0;       /* Handle to database connection *     /
       qeHANDLE        hstmt = 0;      /* Handle to SQL statement execution *     /
       qeSTATUS        res_code;       /* Result code from DTK functions *     /

/* Call qeLibInit to initialize DTK, check for errors. *    /
       res_code = qeLibInit ()  ;
```

```
       if (res_code != qeSUCCESS) return (res_code)   ;

/* Call qeConnect to connect to a data source.  Check to see *    /
/* if hdbc == 0, which indicates that the connection failed. *    /
       hdbc = qeConnect ("DSN=QEDBF")  ;
       if (hdbc == 0) return (err_handler (hdbc, hstmt))   ;

/* Start a transaction. * /
       res_code = qeBeginTran (hdbc)  ;
       if (res_code != qeSUCCESS) return (err_handler (hdbc, hstmt))    ;

/* Call qeExecSQL to execute the update statement. Check if hstmt == 0, *     /
/* which indicates that the statement did not execute successfully. *    /
       hstmt = qeExecSQL (hdbc, "Update emp set first_name = 'Richard' where
first_name = 'Joe'") ;
       if (hstmt == 0) return (err_handler (hdbc, hstmt))   ;

/* Rollback the transaction. *  /
       res_code = qeRollback (hdbc)  ;
       if (res_code != qeSUCCESS) return (err_handler (hdbc, hstmt))    ;

/* Close the statement. *  /
       res_code = qeEndSQL (hstmt)  ;
       hstmt = 0 ;
       if (res_code != qeSUCCESS) return (err_handler (hdbc, hstmt))    ;

/* Call qeDisconnect to disconnect from a data source. *    /
       res_code = qeDisconnect (hdbc)  ;
       hdbc = 0 ;
       if (res_code != qeSUCCESS) return (err_handler (hdbc, hstmt))    ;
/* Call qeLibTerm to free memory allocated by DTK. *    /
       res_code = qeLibTerm ()  ;
       MessageBox (hWnd, "Sample succeeded.", "Transactions", MB_OK)    ;
       return (res_code)  ;
}

/* err_handler routine goes here.  *  /
```

The functions listed in Table 6-1 let you use transactions in your applications:

Table 6-1. Functions that Support Transactions

Function	Result
qeBeginTran	Begins a SQL transaction.
qeCommit	Ends a transaction by committing all changes to the database.
qeRollback	Ends a transaction by rolling back all changes to the database.
qeGetSupportedIsolationLevels	Returns the set of isolation levels supported by the database system.
qeSetIsolationLevel	Sets the isolation level to any of the ones supported by the database system.
qeGetIsolationLevel	Returns the default isolation level provided by the database system.
qeSetSelectOptions	Specifies the following options:
	The level of fetching that is possible after a transaction ends.
	Whether your application only reads forward through the records resulting from a Select statement, or also needs to position to records that have already been read
	Whether DTK will write records in the result set to log files when connected to databases for which it is not necessary to do so.
qeGetSelectOptions	Returns whether previous and random fetching is enabled for the current database connection, whether DTK will use log files when connected to databases for which it is not necessary to do so, and the level of fetching that is possible after a transaction ends.

The qeGetSupportedIsolationLevels, qeSetIsolationLevel, and qeGetIsolationLevel functions let you control the isolation level that the database system provides to your transactions. For information on isolation levels and using them, see “Isolation Levels” on page 85.

The qeSetSelectOptions and qeGetSelectOptions functions provide control over DTK behavior relative to fetching and the use of log files during and after transactions see “Logging” on page 89 and “Controlling Statement Persistence” on page 92 for information on using these functions.

Transactions, Locking, and Logging

This section explains the concept of database transactions. It also explains the concepts of locking and logging as they apply to DTK.

Transactions

A *transaction* is a set of database operations that are grouped into a single unit. In a transaction, multiple database operations are combined so that if a problem occurs at some point during the process, the entire transaction can be canceled and the individual operations that were completed can be undone. Such cancellation marks the end of the transaction, and is called a *rollback*. The way to end a successful transaction is with a *commit*, which accepts the changes made during the transaction and makes them permanent in the database.

The qeBeginTran function starts a transaction. The qeRollback function ends the transaction and discards all database changes made during the transaction. The qeCommit function ends the transaction and makes all changes permanent in the database.

Whenever you are not within a transaction—that is, have not called qeBeginTran to begin a transaction, or have just called qeCommit or qeRollback to end one, you are in auto-commit mode. In auto-commit mode, each database operation you perform is immediately processed by the database system and the changes are immediately committed. You cannot rollback or undo the changes.

Transactions are closely linked to the concepts of *locking* and *isolation levels*, which are described in the following sections.

Locking

Locking is a vital activity in multi-user databases, where different users can try to access or modify the same records concurrently. While such concurrent database activity is desirable, it can create problems. Without locking, for example, if two users try to modify the same record at the same time, they might encounter problems ranging from retrieving bad data to deleting data that the other user needs. However, if the first user to access a record is able to lock that record—temporarily prevent other users from modifying it—such problems can be avoided. Locking provides a way to manage concurrent database access while minimizing the various problems it can cause.

Some locks are automatically acquired by the database system as it processes SQL statements. DTK users can explicitly lock records by calling qeRecLock.

Isolation Levels

An isolation level represents a particular locking strategy employed in the database system to improve data consistency. The higher the isolation level, the more complex the locking strategy behind it.

The isolation level provided by the database determines whether a transaction will encounter the following behaviors in data consistency:

Dirty reads	User 1 modifies a row. User 2 reads the same row before User 1 commits. User 1 performs a rollback. User 2 has read a row that has never really existed in the database. User 2 may base decisions on false data.
Non-repeatable reads	User 1 reads a row but does not commit. User 2 modifies or deletes the same row and then commits. User 1 rereads the row and finds it has changed (or has been deleted).
Phantom reads	User 1 uses a search condition to read a set of rows, but does not commit. User 2 inserts one or more rows that satisfy this search condition, then commits. User 1 rereads the rows using the search condition, and discovers rows that were not present before.

Isolation levels represent the database system's ability to prevent these behaviors. There are four isolation levels defined by ANSI: *read uncommitted* (0), *read committed* (1), *repeatable read* (2), and *serializable* (3). In ascending order (0–3), these isolation levels provide an increasing amount of data consistency to the transaction. At the lowest level, all three behaviors can occur. At the highest level, none of them can occur. The success of each level in preventing these behaviors is due to the locking strategies that they employ, which are as follows:

Read uncommitted (0)	Locks are obtained on modifications to the database and held until end of transaction (EOT). Reading from the database does not involve any locking.
Read committed (1)	Locks are acquired for reading and modifying the database. Locks are released after reading but locks on modified objects are held until EOT.

Repeatable read (2)	Locks are obtained for reading and modifying the database. Locks on all modified objects are held until EOT. Locks obtained for reading data are held until EOT. Locks on non-modified access structures (indexes, hashing structures, etc.) are released after reading.
Serializable (3)	All data read or modified is locked until EOT. All access structures that are modified are locked until EOT. Access structures used by the query are locked until EOT. Some databases provide an additional isolation level, Versioning (4). This isolation level is actually a different implementation of isolation level 3, serializable, but provides greater concurrency through the use of non-locking “record versioning” protocols.

The following table shows what data consistency behaviors can occur at each isolation level:

Level	**Dirty reads**	**Non-repeatable reads**	**Phantom reads**
0, Read uncommitted	Yes	Yes	Yes
1, Read committed	No	Yes	Yes
2, Repeatable read	No	No	Yes
3, Serializable	No	No	No

Support for each isolation level depends on the database system. Many databases do not support all four levels. Refer to the *DataDirect ODBC Drivers Reference* for the isolation levels supported by each database. Your applications can find out what isolation levels the current database system supports by calling qeGetSupportedIsolationLevels. DTK uses the default isolation level provided by the database unless you specifically request one with the qeSetIsolationLevel function. A call to qeGetIsolationLevel returns the current isolation level.

While higher isolation levels provide better data consistency, this consistency can be costly in terms of the *concurrency* provided to individual users. Concurrency is the ability of multiple users to access and modify data simultaneously. As isolation levels increase, so does the chance that the locking strategy used will create problems in concurrency. Put another way: the higher the isolation level, the more locking involved, and the more time users may spend waiting for data to be freed by another user. Because of this inverse relationship between isolation levels and concurrency, you must carefully consider how people use the database before choosing an isolation level. You must weigh the trade-offs between data consistency and concurrency and decide which is more important to your users.

Isolation levels are also a consideration when DTK uses a log file to enable backward and random record fetching. See "Logging and Isolation Levels" on page 90 for more information.

Locking Modes and Granularity

Different database systems employ various locking modes, but they have two basic ones in common: *shared* and *exclusive*. Shared locks can be held on a single object by multiple users. If one user has a shared lock on a record, then a second user can also get a shared lock on that same record. However, the second user cannot get an exclusive lock on that record. Exclusive locks are exclusive to the user that obtains them. If one user has an exclusive lock on a record, then a second user cannot get either type of lock on the same record.

Performance and concurrency can also be affected by the *locking granularity* used in the database system. The locking granularity determines the size of an object that is locked in a database. For example, many database systems let you lock an entire table, as well as individual records. An intermediate level of locking, *page*-level locking, is also common. A page contains one or more records and is typically the amount of data read from the disk in a single disk access. The major disadvantage of page-level locking is that if one user locks a record, a second user may not be able to lock other records because they are stored on the same page as the locked record.

Using qeRecLock

The qeRecLock function explicitly locks the current record. qeRecLock works only if called within a transaction; otherwise, it returns an error. All locks are freed by a call to qeCommit or qeRollback.

qeRecLock enables you to control the locking strategies rather than depending on the database system. For example, by calling qeRecLock after fetching a record, you lock that record until the end of the transaction. This eliminates the possibility of a non-repeatable read, which is the same as if your transaction had operated at isolation level 2 (repeatable read).

See “Logging and qeRecLock” on page 91 for information on using qeRecLock with a log file.

Logging

Most SQL database systems provide only a fetch next function; neither previous nor random fetches are permitted. The qeSetSelectOptions function lets you specify random and previous fetching, as well as forward fetching. For database systems that do not support random or previous fetching, DTK provides the capability by saving each record read in a temporary log file that is stored in your TEMP directory (specified by the “SET TEMP=” line in your DOS AUTOEXEC.BAT or OS/2 CONFIG.SYS file). DTK allows your application to randomly fetch records by reading them back from the log file.

Because many database systems don’t provide a function that returns the number of records selected, DTK provides the qeFetchNumRecs function for this purpose. To call this function you must have enabled random and previous fetching using the qeSetSelectOptions function. Since DTK may need to read and count the records in order to return this information, log files may be required to save the records. These log files are deleted when qeEndSQL is called.

Some database systems automatically terminate Select statements when a transaction ends, preventing you from reading records following a commit or rollback unless you re-execute the statement. DTK lets you avoid this limitation. The qeSetSelectOptions function lets you specify what happens to active Select statements when a transaction ends. If you enable the option to continue reading records after a transaction ends, and the underlying database system does not support it, DTK saves the records in log files.

Since an application can have only a limited number of files open at any time (20 is the DOS/Windows default), you may exceed the limit if your application has other files open or if you have several Select statements active at the same time. You can call qeFetchLogClose to close the temporary log file used by a statement. DTK automatically reopens the file when you call a qeFetch function.

DTK creates and maintains log files containing saved records whenever you use qeSetSelectOptions to enable capabilities that aren’t provided directly by the underlying database system. When log files are used to save records retrieved by a Select statement, DTK reads records from the log file as much as possible instead of re-reading them from the database system. This use of log files creates important considerations regarding locking and isolation levels. The following sections describe these considerations.

Logging and Isolation Levels

Because DTK reads record values from the log file whenever possible, the isolation level provided by the database system affects the accuracy of the data in the log file. Some isolation levels allow records that are saved in the log file to be changed in the database by another user, causing the values in the log file to be different from those in the database. The lower the isolation level, the greater the possibility of this kind of behavior. The level of consistency provided by each isolation level during a transaction is as follows:

0, 1	Records in the log file may not match records in the database.
2	Records in the log file will match those in the database, but there may be new records that have been inserted in the database that aren't present in the log file.
3	The log file will always match the database.

Isolation level 3 provides the best degree of consistency when log files are used. If it is not possible or desirable to use isolation level 3 (or level 4, Versioning), you can call qeRecLock after fetching each record to ensure consistency between the log file and the database. The next section describes how this method prevents consistency problems.

Logging and qeRecLock

Because DTK reads record values from the log file whenever possible, you may want call qeRecLock on each fetch to ensure consistency between the log file and the database, especially if the isolation level is 0 or 1.

qeRecLock always acquires a lock on the current record. Optionally, this function will either warn you when the locked record has changed or automatically refresh the copy in the log file with the corresponding values from the database so that the values you see are always current. The qeSetLockOptions function lets you choose one of the following options:

Constant	Value	Description
qeLOCK_NO_OPTIONS	0	Default; DTK neither compares nor refreshes the record in the log file.

Constant	Value	Description
qeLOCK_COMPARE	1	When locking, DTK compares the record in the log file to the corresponding record in the database, and raises a warning if they are different.
qeLOCK_REFRESH	2	When locking, DTK automatically refreshes the record in the log file with new column values.

Emulated Transactions

Some database systems do not support transactions. When using the *DataDirect ODBC Drivers Reference* drivers for these database systems, DTK transparently emulates transactions so that your application can call qeBeginTran, qeCommit, and qeRollback for these database systems. This emulation is not supported when using third-party database drivers.

Controlling Statement Persistence

Sometimes database systems do not maintain the Select statement's result set beyond the end of a transaction. In such databases, after you issue a commit or rollback you can no longer fetch records using the current *hstmt* because the database can no longer provide a point of reference for the fetch. However, if you are using DTK's logging to enable random and previous record fetching, you don't experience this problem. The log file tracks the current record, so DTK always knows where it is in the database. Because of this ability, you may want to force the use of log files, even if the database you are using doesn't require their use for random and previous record fetching. You can do this by calling qeSetSelectOptions with the qeLOG_ALWAYS option (0x0010).

The qeSetSelectOptions function provides additional control over logging and statement persistence by letting you specify the level of statement persistence that DTK provides at the end of transactions. By default (qeSELECT_PERSIST, 0x0060), DTK will read all of the records that you have not yet fetched into the log file, allowing you to continue updating the entire result set. If you do not need DTK to read all the records but want to continue working with the records you've already fetched, set the qeSELECT_TRUNCATE flag (0x0040). If you don't want DTK to save any records in the log file when a transaction ends, set the qeSELECT_INVALIDATE flag (0x0020). Because these settings affect what happens when changes are committed, they also affect statement persistence when in auto-commit mode. Because auto-commit mode represents an implicit commit of each change you make, whatever you choose to have happen at the end of a transaction will also happen whenever a change is made in auto-commit mode. Therefore, if you intend to use auto-commit mode and change multiple records returned by a statement, you should use the default setting of qeSELECT_PERSIST.

7 Error Handling and Debugging

This chapter describes DTK’s error-handling and debugging functions. The last two sections describe problems that do not return errors.

The following example shows tracing enabled in a DTK program, as well as the error handling routine used for all samples in SAMPLE.EXE. The trace files it creates are listed in the section “Debugging Your Applications.” To load this sample in the SAMPLE.EXE program, choose **Tracing DTK Calls** from the Example List.

```
qeSTATUS trace ()  {

/* This routine demonstrates the use of the tracing facilities. *    /

       qeHANDLE        hdbc = 0;       /* Handle to database connection *    /
       qeHANDLE        hstmt = 0;      /* Handle to SQL statement execution *    /
       qeSTATUS        res_code;       /* Result code from DTK functions *    /
       long            modrecs  ;

/* Call qeLibInit to initialize DTK, check for errors. *    /
       res_code = qeLibInit ()  ;
       if (res_code != qeSUCCESS) return (res_code)   ;

/* Turn tracing on, set options to trace everything. *    /
       res_code = qeTraceOn ("c:\qelib\trace.txt")   ;
       if (res_code != qeSUCCESS) return (err_handler (hdbc, hstmt))    ;

       res_code = qeSetTraceOptions (qeTRACE_NON_VAL_CALLS + qeTRACE_USER     +
                                  qeTRACE_VAL_CALLS + qeTRACE_ODBC)    ;
       if (res_code != qeSUCCESS) return (err_handler (hdbc, hstmt))    ;
/* Call qeConnect to connect to a data source.  Check if hdbc == 0, which *    /
/* indicates that the connection failed. *   /
       hdbc = qeConnect ("DSN=QEDBF")  ;
       if (hdbc == 0) return (err_handler (hdbc, hstmt))   ;

/* Set the ODBC tracefile. *  /
       res_code = qeSetDriverTracefile (hdbc, "c:\qelib\odbc.txt")    ;
       if (res_code != qeSUCCESS) return (err_handler (hdbc, hstmt))    ;
```

```
/* Call qeExecSQL to execute the update statement. Check if hstmt == 0, *     /
/* which indicates that the statement did not execute successfully. *     /
      hstmt = qeExecSQL (hdbc, "Update emp set first_name = 'Joe' where first_name
= 'Richard'") ;
       if (hstmt == 0) return (err_handler (hdbc, hstmt))    ;

/* Find out how many records were affected by the statement. *    /
       modrecs = qeNumModRecs (hstmt)  ;
       if (qeErr () != qeSUCCESS) return (err_handler (hdbc, hstmt))    ;

/* Close the statement. * /
       res_code = qeEndSQL (hstmt)  ;
       hstmt = 0 ;
       if (res_code != qeSUCCESS) return (err_handler (hdbc, hstmt))    ;

/* Call qeDisconnect to disconnect from a data source. *    /
       res_code = qeDisconnect (hdbc)  ;
       hdbc = 0 ;
       if (res_code != qeSUCCESS) return (err_handler (hdbc, hstmt))    ;

/* Close the tracefiles. * /
       res_code = qeTraceOff ()  ;
       if (res_code != qeSUCCESS) return (err_handler (hdbc, hstmt))    ;
/* Call qeLibTerm to free memory allocated by DTK. *    /
       res_code = qeLibTerm ()  ;
       MessageBox (hWnd, "Sample succeeded.", "Trace Functions", MB_OK)    ;
       return (res_code)  ;
}

/* err_handler routine goes here.  *   /
```

The following sections describe the use of DTK's error-handling and debugging functions.

Handling Errors and Warnings

DTK provides the error handling functions listed in Table 7-1.

Table 7-1. Error Handling Functions

Function	Returns
qeErr	The result code of the last DTK function you called
qeDBErr	The database error resulting from the last DTK function
qeErrMsg and qeErrMsgBuf	The error message generated by the last DTK function you called.
qeWarning	The DTK or database warning generated by the last DTK function you called.

DTK allows various methods of error checking, because every DTK function can detect errors in its execution. Many DTK functions return an error status *result code* as the value returned by their execution. Similarly, functions that return a handle to the database connection (*hdbc*) or SQL statement (*hstmt*) return a value of zero if they do not execute successfully. Also, the qeErr function is available to report the result code of the last function that executed. qeErr reports the status of all DTK functions.

All DTK functions that return result codes, including qeErr, report the same set of status constants. A result code of zero from these functions indicates that they succeeded (qeSUCCESS). A non-zero result indicates that an error or warning occurred. When checking the result code value, you can use either the constant name (such as qeSUCCESS) or explicit value (like 0).

A result code of qeSUCCESS_WITH_INFO (1) means that the function was successful, but returned warning information. When this occurs, call qeWarning to get the warning information. You should also call qeWarning whenever qeErr returns qeNO_DATA_WITH_INFO (2).

qeWarning returns warnings, including the qeTRUNCATION and qeNULL_DATA warnings. If a function results in both an error and a warning, qeErr will report only the error, so you should call qeWarning in your error-handling routines to see if any warnings were issued.

You can call qeErrMsg or qeErrMsgBuf to get the error message associated with the result code. DTK error messages contain up to 512 characters. When you call either of these functions, your programs must be able to handle these messages. When you call qeErrMsgBuf, the variable you pass as the parameter must be large enough to hold 512 characters.

If the error is detected by the underlying database system, the database system's error code can be retrieved with qeDBErr. For example, if you are using Oracle and Oracle detects an error, qeErr returns qeDBSYS_ERROR (4), qeDBErr returns the Oracle error code, and qeErrMsg returns the text of the message. The Oracle error codes are described in the Oracle documentation set.

Note: It is very important that you check for errors following every call to a DTK function. Ignoring errors in your programs may result in your program or a DTK function causing a General Protection Fault (GPF).

Debugging Your Applications

DTK provides tracing functions that let you log calls to the functions for database connection and SQL execution.

When tracing is on, all parameters sent to DTK functions, as well as all values they return, are written to an ASCII file. You can look at this file to see where errors in your program exist.

This sample application at the beginning of this chapter returns the trace file shown on page 121, named TRACE.TXT.

```
qeTraceOn (c:\qelib\trace.txt   )
qeTraceOn returns (0  )
qeSetTraceOptions (23  )
qeSetTraceOptions returns (0   )
qeConnect (DSN=QEDBF  )
qeConnect returns (1  )
qeSetDriverTracefile (1, c:\qelib\odbc.txt    )
qeSetDriverTracefile returns (0   )
qeExecSQL (1, Update emp set first_name = 'Joe' where
first_name = 'Richard'  )
qeExecSQL returns (2  )
qeNumModRecs (2 )
qeNumModRecs returns (0  )
qeEndSQL (2 )
qeEndSQL returns (0  )
qeDisconnect (1 )
qeDisconnect returns (0  )
qeTraceOff ( )
```

The sample also returns an ODBC trace file named ODBC.TXT.

```
SQLAllocStmt(hdbc116F0000, phstmt08DF0000)    ;
SQLPrepare(hstmt08DF0000, "Update emp set
first_name = 'Joe' where first_name = 'Richard'", 62)     ;
SQLExecute(hstmt08DF0000)  ;
SQLNumResultCols(hstmt08DF0000, pccol)    ;
SQLRowCount(hstmt08DF0000, pcrow)   ;
SQLMoreResults(hstmt08DF0000)   ;
SQLFreeStmt(hstmt08DF0000, 1)   ;
SQLDisconnect(hdbc116F0000)   ;
```

DTK provides the trace functions listed in Table 7-2; these functions let you log calls to the database connection functions and SQL execution functions:

Table 7-2. Functions that Log Calls to Database-Connection and SQL-Execution Functions

Function	Result
qeTraceOn	Starts tracing calls to the DTK API by writing debugging information to a trace file.
qeTraceOff	Closes the trace file opened by qeTraceOn and discontinues the tracing of calls to the DTK API.
qeSetDriverTracefile	Specifies a file as the driver trace file.
qeSetTraceOptions	Sets the type of information that is sent to the trace file.
qeGetTraceOptions	Returns the type of information that is sent to the trace file.
qeTraceUser	Sends a string to the trace file.

When tracing is on, all parameters sent to DTK functions, as well as all values they return, are written to an ASCII file. You can look at this file to see where errors in your program exist. DTK continues to write to the trace file until you call qeTraceOff.

Tracing Statement and Connection Errors

The trace file created by the qeTrace functions provides the best method for discovering errors in the following:

- Connection strings passed to the database via qeConnect
- SQL statements passed via qeExecSQL and qeSQLExecute

The two sections that follow show how such errors affect the contents of the DTK trace file.

Trace files resulting from each type of error are compared to the following trace file text, which was created by the DTK tracing example at the beginning of the chapter:

```
qeTraceOn (c:\qelib\trace.txt   )
qeTraceOn returns (0  )
qeSetTraceOptions (23  )
qeSetTraceOptions returns (0   )
qeConnect (DSN=QEDBF  )
qeConnect returns (1  )
qeSetDriverTracefile (1, c:\qelib\odbc.txt    )
qeSetDriverTracefile returns (0   )
qeExecSQL (1, Update emp set first_name = 'Joe' where
first_name = 'Richard'  )
qeExecSQL returns (2  )
qeNumModRecs (2 )
qeNumModRecs returns (0  )
qeEndSQL (2 )
qeEndSQL returns (0  )
qeDisconnect (1 )
qeDisconnect returns (0  )
qeTraceOff ( )
```

Calling qeExecSQL with an Invalid SQL Statement

Suppose that the program that created the preceding trace file contained a qeExecSQL call like this:

```
hstmt = qeExecSQL (hdbc, "Updte emp set first_name =
'Joe' where first_name = 'Richard'")   ;
```

Note that the Update keyword in the statement parameter is misspelled. This error results in the trace file shown on page 124.

```
qeTraceOn ("c:\qelib\trace.txt"   )
qeTraceOn returns (0  )
qeSetTraceOptions (23  )
qeSetTraceOptions returns (0   )
qeConnect ("DSN=QEDBF"  )
qeConnect returns (1  )
qeSetDriverTracefile (1, "c:\qelib\odbc.txt"    )
qeSetDriverTracefile returns (0   )
qeExecSQL (1, "Updte emp set first_name = 'Joe' where
first_name = 'Richard'"  )
qeExecSQL returns (0  )
qeExecSQL DBErr is (3800   )
qeErr returns (4  )
qeErr DBErr is (3800  )
qeErrMsg returns ("[INTERSOLV][ODBC dBase
driver][dBase]Only SELECT, INSERT, UPDATE, DELETE,
CREATE, and DROP statements are supported."    )
qeDisconnect (1  )
qeDisconnect returns (0  )
```

This trace file shows that qeExecSQL returned a zero (0), which indicates an error. qeErr returns 4 (qeDBSYS_ERR), which indicates that the error was reported by the database. The 3800 code returned by qeDBErr was reported by the ODBC dBASE driver DLL. qeErrMsg reports the corresponding text of this message.

Note: Since the qeExecSQL statement results in an error, none of the subsequent DTK function calls appear in the trace file.

Calling qeConnect with an Invalid Connection String

For this example, suppose that the qeConnect call contains an invalid connection string (a typographical error is made when entering the data source name attribute; it should be DSN rather than DSM).

```
hdbc = qeConnect ("DSM=QEDBF")   ;
```

The trace file reads as follows:

```
qeTraceOn ("c:\qelib\trace.txt"   )
qeTraceOn returns (0  )
qeSetTraceOptions (23  )
qeSetTraceOptions returns (0   )
qeConnect ("DSM=QEDBF"  )
qeConnect returns (0  )
qeErr returns (2106  )
qeErrMsg returns ("Connection string must contain a
DSN=<driver_name>: DSM=QEDBF"   )
```

The invalid call to qeConnect returned a zero (0), which indicates a connection could not be made.

Note: Since the qeConnect statement results in an error, none of the subsequent DTK function calls appear in the trace file.

Go To

8 QBE and Query Builder Functions

This chapter describes the DTK functions that allow you to add flexible querying features to your applications. With these functions you can design applications that let your users dynamically control which records will be retrieved, find records based on values in their fields, or even specify complete SQL Select statements to determine the data to be retrieved. DTK has two sets of functions that allow you to add a Query By Example (QBE) interface or a Query Builder interface to your application.

This chapter contains the following sections:

- "Using Query By Example and Finding Records" on page 127 describes the concepts and techniques related to the QBE interface.
- "Using QBE Functions" on page 130 describes the QBE functions and their usage.
- "Using Query Builder Functions" on page 132 describes the functions that use the Query Builder interface and the query file (QEF) format.
- "The Query Builder Interface" on page 136 explains the query builder interface and some of the features it provides.

Using Query By Example and Finding Records

Query By Example (QBE) is a way to let users of your application dynamically change the Where clause of SQL Select statements.

For example, assume your base Select statement is

```
SELECT last_name, first_name, salary, hire_date
FROM emp ORDER BY last_name
```

and you want to enable users of your application to limit the employee records that will be returned. To implement a QBE interface, you could display a window containing an edit box for each of the four fields, and let the user enter values in each edit box. When the user clicks the **OK** button, you could use the values in each edit box to generate the Where clause in the Select statement.

For example, if the user entered “S” in the LAST_NAME edit box, you could add

```
WHERE last_name LIKE 'S%'
```

to the base Select statement. Similarly, you could add additional conditions to the Where clause as the user enters values in the other edit boxes.

The QBE functions serve as tools that make it easier for your program to modify the Where clause of Select statements.

DTK also enables your program to specify conditions on the column values. For example, assume your program's base Select statement is

```
SELECT last_name, first_name FROM emp
ORDER BY last_name
```

This may return a large number of records. You may want to let users position to the first record having a last name that starts with an “S” without changing the Select statement. DTK provides functions that allow you to position to records based on field values.

The following code gives you a framework for using the QBE functions in your application. The base Select statement in this example is

```
SELECT first_name, last_name FROM emp
```

The example uses DTK's QBE functions to add a condition that returns only those employees whose first name begins with a T, then reads the records and displays the first name values in message boxes. To load this sample in the SAMPLE.EXE program, choose **Using Query By Example** from the Example List.

```
qeSTATUS qbe () {

/* This routine demonstrates the use of Query By Example (QBE). *    /

       qeHANDLE        hdbc = 0;       /* Handle to database connection *     /
       qeHANDLE        hstmt = 0;      /* Handle to SQL statement execution *     /
       qeSTATUS        res_code;       /* Result code from DTK functions *     /
       qeLPSTR         first_name  ;
/* Call qeLibInit to initialize DTK, check for errors. *    /
       res_code = qeLibInit ()  ;
       if (res_code != qeSUCCESS) return (res_code)   ;

/* Call qeConnect to connect to a data source.  Check to see *    /
/* if hdbc == 0, which indicates that the connection failed. *    /
       hdbc = qeConnect ("DSN=QEDBF")  ;
       if (hdbc == 0) return (err_handler (hdbc, hstmt))   ;

/* Select first & last names from emp. */
       hstmt = qeExecSQL (hdbc, "select first_name, last_name from emp")     ;
       if (hstmt == 0) return (err_handler (hdbc, hstmt))   ;

/* Set a condition to search for all first names starting with 'T'. *     /
       res_code = qeRecSetConditionChar (hstmt, 1, qeFIND_LIKE, "T%", "" FALSE)     ;
       if (res_code != qeSUCCESS) return (err_handler (hdbc, hstmt))    ;

/* Re-Execute the Select statement incorporating the QBE conditions. *     /
       res_code = qeSQLxecute (hstmt)  ;
       if (res_code != qeSUCCESS) return (err_handler (hdbc, hstmt))    ;
/* Fetch and display the first names of the records found. */
       while (qeFetchNext (hstmt) == qeSUCCESS)    {
               first_name = qeValChar (hstmt, 1, "", 0)   ;
               if (qeErr () != qeSUCCESS && qeErr () != qeNULL_DATA) break     ;
               MessageBox (hWnd, first_name, "Query By Example", MB_OK)    ;
       }
       if ((qeErr () !=qeSUCCESS) && (qeErr () != qeEOF)    )
               return (err_handler (hdbc, hstmt))   ;
/* Close the statement. *  /
       res_code = qeEndSQL (hstmt)  ;
```

```
        hstmt = 0 ;
        if (res_code != qeSUCCESS) return (err_handler (hdbc, hstmt))    ;

/* Call qeDisconnect to disconnect from a data source. *    /
        res_code = qeDisconnect (hdbc)  ;
        hdbc = 0 ;
        if (res_code != qeSUCCESS) return (err_handler (hdbc, hstmt))    ;

/* Call qeLibTerm to free memory allocated by DTK. *    /
        res_code = qeLibTerm ()  ;
        MessageBox (hWnd, "Sample succeeded.", "QBE Conditions", MB_OK)    ;
        return (res_code) ;
}

/* err_handler routine goes here.  *   /
```

Using QBE Functions

The Query By Example (QBE) and Find functions make it easier for you to write a program that enables users to change query conditions at runtime and position to records using field values.

Table 8-1 lists the DTK functions that provide these capabilities.

Table 8-1. Functions that Change Query Conditions at Runtime

Function	Results
qeQBEPrepare	Prepares a statement containing QBE search conditions.
qeRecClearConditions	Clears a statement's search conditions.
qeRecSetConditionBinary	Adds a search condition to the statement having a binary value to compare.
qeRecSetConditionChar	Adds a search condition to the statement having a character value to compare.
qeRecSetConditionDecimal	Adds a search condition to the statement having a decimal value to compare.

Table 8-1. Functions that Change Query Conditions at Runtime (cont.)

Function	Results
qeRecSetConditionDouble	Adds a search condition to the statement having a double-precision floating-point value to compare.
qeRecSetConditionFloat	Adds a search condition to the statement having a floating-point value to compare.
qeRecSetConditionInt	Adds a search condition to the statement having a 2-byte integer value to compare.
qeRecSetConditionLong	Adds a search condition to the statement having a 4-byte integer value to compare.
qeRecSetConditionNull	Adds a search condition to the statement having a value to compare of null.
qeRecFind	Locates the row matching the qeRecSetCondition search criteria.

All QBE functions require an active SQL Select statement and therefore require an *hstmt* as a parameter. You can activate a Select statement by calling either qeExecSQL, qeSQLPrepare, or qeQBEPrepare.

The qeRecSetCondition functions specify the conditions to be added to the Where clause. These functions have a parameter that identifies the column of the Select statement that receives the condition, an operator parameter that specifies the SQL relational operator to be used, and the value that is to be compared against.

qeRecClearConditions removes all conditions that have been specified.

After the conditions have been set, a call to qeQBEPrepare adds to the Select statement's Where clause and prepares the resulting statement. You must then call qeSQLExecute to execute this statement. Subsequent calls to the qeFetch functions retrieve the records that result from the modified Select statement.

To find records using their field values, you execute Select statements and set conditions just as you do for QBE. However, instead of calling qeQBEPrepare, you call qeRecFind. qeRecFind does not change the Where clause or re-execute the Select statement. Instead, it locates a record in the result set that matches the specified conditions and makes it the current record. When using qeRecFind, you can specify whether you want to position to the first or last record that matches the conditions, or whether you want to search for the next or previous record that matches the conditions.

Using Query Builder Functions

DTK's Query Builder functions provide a simple way for users to create SQL Select statements. Calling qeQryBuilder in your application (available only in Windows, Windows 95, and Windows NT) displays a window that allows your users to create or modify Select statements by pointing and clicking. Your users can manipulate Select statements even if they have no knowledge of SQL.

The following sample code allows the user to enter a Select statement with the Query Builder, executes the resulting statement, and then reads and displays the values in the first column returned by the statement. To load this sample in the SAMPLE.EXE program, choose **Using the Query Builder** from the Example List.

```
qeSTATUS querybuilder ()   {

/* This routine demonstrates the execution of the Query Builder from within *     /
/* a DTK program. * /

       qeHANDLE        hdbc = 0;       /* Handle to database connection *    /
       qeHANDLE        hstmt = 0;      /* Handle to SQL statement execution *     /
       qeSTATUS        res_code;       /* Result code from DTK functions *    /
       qeHANDLE        hqry = 0;       /* Handle to query object *    /

/* Call qeLibInit to initialize DTK, check for errors. *    /
       res_code = qeLibInit ()  ;
       if (res_code != qeSUCCESS) return (res_code)   ;
```

```
/* Call qeConnect to connect to a data source.  Check to see *    /
/* if hdbc == 0, which indicates that the connection failed. *    /
       hdbc = qeConnect ("DSN=QEDBF")  ;
       if (hdbc == 0) return (err_handler (hdbc, hstmt))   ;

/* Allocate a query structure to be used for Query Builder calls. *    /
       hqry = qeQryAllocate (hdbc, "")  ;
       if (hqry == 0) return (err_handler (hdbc, hstmt))   ;

/* Run the query builder.  The resulting statement will be stored in hqry. *     /
       res_code = qeQryBuilder (hqry, hWnd   ,
               qeQRY_BIG_ICONS + qeQRY_TABLES + qeQRY_VIEWS, qeQRY_DEFAULT)     ;
       if (res_code != qeSUCCESS) return (err_handler (hdbc, hstmt))    ;
/* Prepare & execute the statement created in the query builder. *    /
       hstmt = qeQryPrepare (hqry)  ;
       if (hstmt == 0) return (err_handler (hdbc, hstmt))   ;

       res_code = qeSQLExecute (hstmt)  ;
       if (res_code != qeSUCCESS) return (err_handler (hdbc, hstmt))    ;

/* Free query structure. *  /
       res_code = qeQryFree (hqry)  ;
       if (res_code != qeSUCCESS) return (err_handler (hdbc, hstmt))    ;

/* Call qeDisconnect to disconnect from a data source. *    /
       res_code = qeDisconnect (hdbc)  ;
       hdbc = 0 ;
       if (res_code != qeSUCCESS) return (err_handler (hdbc, hstmt))    ;

/* Call qeLibTerm to free memory allocated by DTK. *   /
       res_code = qeLibTerm ()  ;
       MessageBox (hWnd, "Sample succeeded.", "Query Builder", MB_OK)    ;
       return (res_code)  ;
}

/* err_handler routine goes here.  *  /
```

Table 8-2 lists the functions DTK provides for using the Query Builder tool.

Table 8-2. Functions that Support the Query Builder Tool

Function	Result
qeQryAllocate	Builds a query based on a string containing a SQL statement.
qeQryFree	Frees the memory associated with an *hqry*.
qeQryGetFileName and qeQryGetFileNameBuf	Returns the file name associated with the query represented in *hqry*.
qeQryGetFileOffset	Returns the offset of the extra information within the query file that is associated with the query.
qeQryGetHdbc	Returns the *hdbc* associated with the query represented by *hqry*.
qeQryGetNumParams	Returns the number of parameters associated with the query represented by *hqry*.
qeQryGetParamDefault and qeQryGetParamDefaultBuf	Returns the default value of a parameter associated with the specified query.
qeQryGetParamFormat and qeQryGetParamFormatBuf	Returns the format string to be applied to the value of a parameter associated with the specified query.
qeQryGetParamName and qeQryGetParamNameBuf	Returns the name of a parameter associated with the specified query.
qeQryGetParamPrompt and qeQryGetParamPromptBuf	Returns the prompt for a parameter associated with the specified query.
qeQryGetParamType	Returns the type of a parameter associated with the specified query.
qeQryGetStmt and qeQryGetStmtBuf	Returns the statement associated with the query represented in *hqry*.
qeQryOpenQueryFile	Builds a handle to a query based on the contents of the query file.

Table 8-2. Functions that Support the Query Builder Tool (cont.)

Function	Result
qeQryGetSource and qeQryGetSourceBuf	Returns the data source name used in a query (.QEF) file.
qeQrySetSource	Sets the data source name used in a query file.
qeQrySetHdbc	Resets the *hdbc* from a query file with the current *hdbc*.
qeQrySaveQueryFile	Writes a query to a query file.
qeQrySetFileName	Sets the file name for a query file.
qeQrySetNumParams	Sets the number of parameters associated with the query represented by *hqry*.
qeQrySetParamDefault	Sets the default value of a parameter associated with the specified query.
qeQrySetParamFormat	Sets the format string for a parameter associated with the specified query.
qeQrySetParamName	Sets the name of a parameter associated with the specified query.
qeQrySetParamPrompt	Sets the prompt for a parameter associated with the specified query.
qeQrySetParamType	Sets the type of a parameter associated with the specified query.
qeQrySetStmt	Sets the statement associated with the specified query.
qeQryBuilder	Runs the Query Builder.
qeQryPrepare	Prepares a SQL statement for execution.

The Query Builder functions operate on query objects. A query object is created by a call to either qeQryAllocate or qeQryOpenQueryFile and is freed by a call to qeQryFree. qeQryAllocate and qeQryOpenQueryFile return a handle to the query object (*hqry*) that identifies the query object in other

Query Builder functions. qeQryAllocate allows you to specify an optional Select statement for the new query object. qeQryOpenQueryFile reads a Select statement from a query file (.QEF extension) that has been previously created by DTK, INTERSOLV DataDirect Explorer, or another INTERSOLV product.

Once you have a query object, calling qeQryBuilder creates a window that displays the query object's current Select statement, if any. After the user changes the Select statement, clicking **OK** closes the window and updates the query object with the modified Select statement.

The attributes of the query object can be read or changed by calling the qeQryGet and qeQrySet functions. A query file can be generated from the query object by calling qeQrySaveQueryFile.

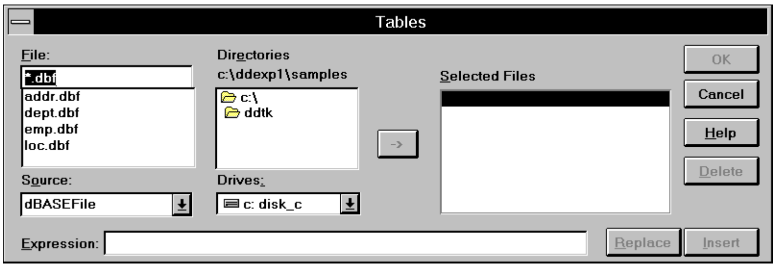
You can execute the Select statement contained in a query object by calling qeQryPrepare followed by qeSQLExecute.

The Query Builder window also allows users to define parameters for the Select statement. If parameters have been defined, calling qeSQLExecute causes DTK to display a dialog box requesting the values to be substituted for the parameters. You can determine the number of parameters that have been defined by calling qeQryGetNumParams. You can read or change parameter definitions by calling the qeQryGetParam and qeQrySetParam functions.

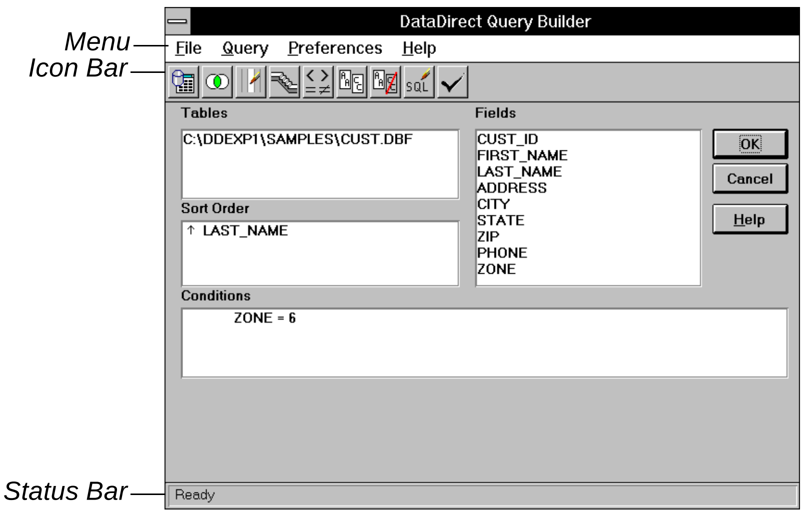
The Query Builder Interface

When your program calls qeQryBuilder, DTK creates windows that let your users create or modify Select statements. These windows don't display the actual text of the Select statement; instead, they split the Select statement into various parts and display the parts in separate list boxes. Presenting the Select statement this way enables users to modify Select statements using the Query Builder's point and click interface—without having to learn the SQL language.

When qeQryBuilder is called, the first window displayed depends on whether a Select statement is already defined for the query object. If there is no Select statement, the first window displayed allows the user to choose one or more tables that are to be included in the Select statement.



Once the user chooses a table and clicks the **OK** button, the main Query Builder window appears. This is the first window displayed if the query object contains a Select statement when qeQryBuilder is called. It has menu and icon bars across the top, and a status bar across the bottom.



In this window, the separate parts of the Select statement appear in the following list boxes:

Tables	Lists the database tables from which records will be retrieved (From clause).
Fields	Lists the fields of the database table to be displayed (the column expressions).
Sort order	Lists sort orders for the records (Order By clause).
Conditions	Lists conditions used to specify which records are to be displayed (Where clause).

To modify the information in one of these boxes, the user can either click on the box, use the corresponding command in the Query menu, or click the corresponding icon on the icon bar. The Query Builder then displays a dialog box which lets the user change the information in the list box. A Help button in each of these dialog boxes displays detailed information on how to use them.

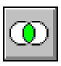
Once all changes have been made, clicking the OK button changes the Select statement in the query object to reflect the changes. Clicking Cancel discards all changes leaving the query object unchanged.

Query Builder Icons

The following icons are available when you are using the Query Builder:

Three additional boxes may be displayed—a Table Joins box, a Group By box, and a Having box. The Table Joins box appears when the user defines a join among database tables. The other two boxes appear when a Group By clause is defined.

The *Table* icon allows you to define the database tables from which fields will be selected.

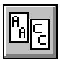


The *Joins* icon allows you to specify how to relate tables. This is valid only if you have specified more than one database table for the query.

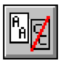
The *Field* icon allows you to specify which fields of the table you want retrieved.

The *Sort* icon allows you to specify the fields by which you want the records sorted.

The *Conditions* icon allows you to specify conditions (for example, display all employees who have an annual salary greater than $30,000).



The *Groupings* icon allows you to group sets of records and to define aggregate functions to compute (for example, average the salaries in each department).



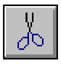
The *Having* icon allows you to specify additional conditions for groups of records (for example, retrieve only the departments that have an average salary of more than $20,000). You can have a Having clause only if you have already defined a Group By clause.

The *Edit Query Text* icon displays the SQL Select statement that corresponds to the current query definition. You can edit the statement from this screen.

The *Validity Check* icon checks the syntax of a SQL Select statement that you have modified and reports any errors.

Edit Query Text Icons

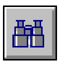
The next six icons are available only when you are in the Edit Query Text screen.



The *Cut* icon removes a highlighted section of text from the screen and places it onto the clipboard.

The *Copy* icon copies a highlighted section of text from the screen to the clipboard.

The *Paste* icon pastes clipboard contents onto the screen in front of the cursor or replaces the highlighted section with the contents of the clipboard.



The *Find* icon searches and moves the cursor to the text that you specify.

The *Find Next* icon finds the next occurrence of the specified text.

The *Replace* icon searches for the specified text and replaces it with different text that you have specified.

The Preferences menu contains options you can set. The three options are:

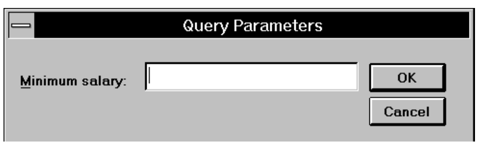
Use Database to Validate	If set, the Query Builder uses the database system to validate the query conditions as you build them. If not set, the Query Builder does not use the database system to check for errors, so you may construct conditions that have errors when you execute the query. The default is to validate.
Large/Small Icons	This option determines whether large or small icons are displayed on the icon bar. Large icons are the default.
Sample Values from Database	This option determines whether database values are displayed when you are defining field conditions. The default is to display values.

Query Builder Parameters

The Query Builder supports parameters in Select statements. For example, you can use the Query Builder to generate the following Select statement:

```
Select * from emp where salary > ?sal
```

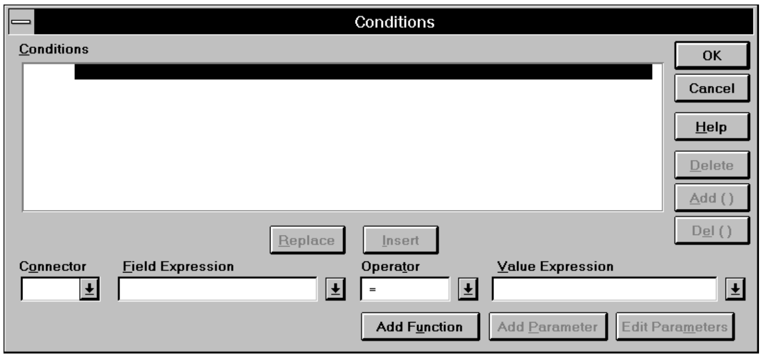
A subsequent call to qeSQLExecute will display the following dialog box:



The user is prompted for a salary value to substitute for the ?sal parameter. This value is used when qeSQLExecute is called to execute the statement.

To build a Select statement with a parameter, you modify the Where clause by clicking the Conditions list box in the Query Builder's main window.

This causes the Conditions dialog box to appear:



The easiest way to specify a condition is to choose a field from the drop-down Field Expression list, choose an operator from the drop-down Operator list, and choose or type a value for the Value Expression box. Once the condition is complete, click the Insert button to insert it in the list box labeled Conditions.

For example, to create the condition “salary > ?sal,” you could do the following:

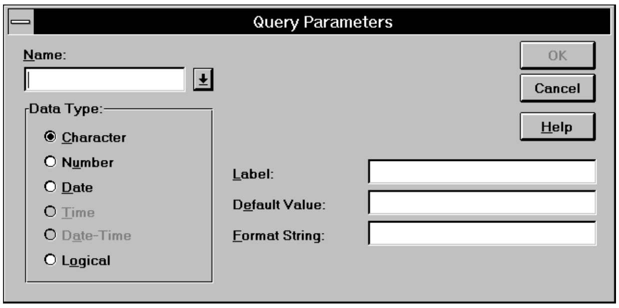
1 Choose SALARY from the drop-down Field Expression list.

2 Choose > from the drop-down Operator list.

3 Click in the Value Expression box.

4 Click the Add Parameter button.

The following dialog box appears, allowing you to define the parameter:



5 Enter the name of the parameter, sal in our example, in the Name box. Optionally, supply the following information:

- A Label for the parameter value
- A Default Value for the parameter
- A Format String used to describe how date, time, or numeric values will be entered by the user.

The qeQryPrepare function uses this information when displaying the dialog box in which the user enters the parameter value.

These steps can be repeated to add additional parameters to the condition.

If a query object's Select statement contains parameters, the attributes of the parameters can be read or modified using the qeQryGetParam and qeQrySetParam functions. The number of parameter can be read or modified using the qeQryGetNumParams and qeQrySetNumParams functions.

Go To

9 Utility Functions

- This chapter describes the following DTK functions:
- "Using Data Dictionary Functions," next, describes the functions that query the system to determine what data sources, databases, table, and stored procedures are available.
- "Parsing SQL Statements" on page 147 describes the functions that parse the Where, Having, Group By, Order By, and Compute By clause, or other database-specific condition clauses from a SQL Select statement.
- "ODBC Handle Conversion" on page 149 describes the functions that convert DTK handles to ODBC handles for direct addressing of the ODBC API.

Using Data Dictionary Functions

Many database systems have information available about the data that is stored in them. This data can include information about the databases, tables, columns, indexes, keys, and privileges associated with the data. DTK returns this information as if it were a result set from a query, returning records that have a fixed format for each type of information requested.

The sample program on page 143 shows how to call the data dictionary functions. To load this sample in the SAMPLE.EXE program, choose **Getting Data Dictionary Information** from the Example List.

```
qeSTATUS datadict ()  {

/* This routine demonstrates calls to qeSources, a data dictionary routine *     /
/* that returns an hstmt whose result set contains a list of the available *     /
/* database system sources. *  /
```

```
      qeHANDLE       hdbc = 0;      /* Handle to database connection *    /
      qeHANDLE       hstmt = 0;     /* Handle to SQL statement execution *    /
      qeSTATUS       res_code;      /* Result code from DTK functions *    /
      char           source [qeSRC_MAX_LEN+1]   ;
      long           source_len = qeSRC_MAX_LEN+1   ;
      char           extension [qeSRC_MAX_LEN+1]   ;
      long           extension_len = qeSRC_MAX_LEN+1   ;
      short          source_hdbc  ;
      long           source_hdbc_len = sizeof (source_hdbc)    ;
      char           remark [qeSRC_REMARK_MAX_LEN+1]   ;
      long           remark_len = qeSRC_REMARK_MAX_LEN+1    ;

/* Call qeLibInit to initialize DTK, check for errors *   /
      res_code = qeLibInit ()  ;
      if (res_code != qeSUCCESS) return (res_code)   ;

/* Note: you do not have to be connected to get the list of Sources *    /

/* Get an hstmt whose result set contains records that describe each Source. *    /
      hstmt = qeSources (1)  ;
      if (hstmt == 0) return (err_handler (hdbc, hstmt))    ;

/* Bind local variables to the columns returned for each record *    /
      res_code = qeBindColChar (hstmt, 1, source, &source_len, "")    ;
      if (res_code != qeSUCCESS) return (err_handler (hdbc, hstmt))    ;

      res_code = qeBindColChar (hstmt, 2, extension, &extension_len, "")     ;
      if (res_code != qeSUCCESS) return (err_handler (hdbc, hstmt))    ;

      res_code = qeBindColInt (hstmt, 3, &source_hdbc, &source_hdbc_len)     ;
      if (res_code != qeSUCCESS) return (err_handler (hdbc, hstmt))    ;

      res_code = qeBindColChar (hstmt, 4, remark, &remark_len, "")    ;
      if (res_code != qeSUCCESS) return (err_handler (hdbc, hstmt))    ;
/* Fetch rows and display each source name in a message box. *    /
      while (qeFetchNext (hstmt) == qeSUCCESS)    {
              MessageBox (hWnd, source, "Data Dictionary: qeSources", MB_OK)     ;
       }
      if ((qeErr () != qeSUCCESS) && (qeErr () != qeEOF)    )
              return (err_handler (hdbc, hstmt))   ;

/* Close the data dictionary statement. *   /
      res_code = qeEndSQL (hstmt)  ;
      hstmt = 0 ;
      if (res_code != qeSUCCESS) return (err_handler (hdbc, hstmt))    ;
```

```
/* Call qeLibTerm to free memory allocated by DTK. *   /
       res_code = qeLibTerm ()  ;
       MessageBox (hWnd, "Sample succeeded.", "Data Dictionary", MB_OK)    ;
       return (res_code)  ;
}

/* err_handler routine goes here.  *  /
```

This example shows how to get ODBC data source information using the qeSources function.

For database systems that support indexing, the qeIndexes function returns information on the set of indexes for a table. An index is a storage structure that provides quick access to a table's rows based on the values of one or more columns in the row. It is analogous to the index in a book: it stores data values in ascending or descending order, and each index value contains a pointer to the value's location within the table. Thus, if the database system needs to search for a value on a column that has been indexed, the database system does not search the table itself, whose rows are in random order; rather, it quickly searches the ordered index, locates the value, and then follows the index pointers to locate the row or rows that contain a value that meets the search criteria. (If the column has not been indexed, the database system must sequentially scan each row in the table and evaluate the column's value; to ensure it finds all values that meet the search criteria, it needs to scan the entire table, which could be time-consuming.)

For database systems that support primary keys, the qePrimaryKeys function returns information on the set of columns that compose a table's primary keys. A primary key is a column or combination of columns whose values uniquely identify each row in the table. For example, an EMP_ID column might uniquely identify each row in an EMP table and could be defined as the table's primary key. If a single column cannot uniquely identify each row, a combination of columns can be defined as the primary key. For example, a PARTS table might contain PART_NO and MFR columns. In this case, the part number might not uniquely identify rows since two manufacturers might use the same part number, but the combination of PART_NO and MFR might be better for the table's primary key.

For database systems that support foreign keys, the qeForeignKeys function returns information on the set of columns that compose a table's foreign keys. A foreign key is a column in one table whose values are derived from the primary key in another table. For example, a SALESREP table might include a column for SALES_TERR, which contains values identifying sales territories. These territory values might match the values in a TERRITORY field, which has been defined as the primary key in a TERRITORIES table.

Table 9-1 lists the entire set of data dictionary functions.

Table 9-1. Data Dictionary Functions

Function	Returns
qeColumns	Information on the set of column definitions for a table.
qeDatabases	Information on the set of databases that can be accessed.
qeForeignKeys	Information on the set of columns that compose a table's foreign keys.
qeIndexes	Information on the set of indexes for a table.
qePrimaryKeys	Information on the set of columns that compose a table's primary keys.
qeProcedureColumns	Information that describes the parameters to a stored procedure and the result columns for that procedure. The rows may be retrieved subject to the same restrictions as qeTables (and other DTK procedures which return result sets).
qeSources	Information on the database Sources (systems) that can be accessed.
qeTables	Information on the available database tables.
qeTypeInfo	Information about the types supported on a particular database.
qeGetTableCaching	Returns the caching setting specified in the last call to qeSetTableCaching.

Table 9-1. Data Dictionary Functions (cont.)

Function	**Returns**
qeSetTableCaching	Controls whether table information is cached after calls to qeTables.
qeSetCacheFileName	Sets the file name to be used when caching table names.

Except for qeGetTableCaching, qeSetTableCaching, and qeSetCacheFileName, these functions return an *hstmt*. The records for the *hstmt* can be read using qeFetchNext, and column values can be retrieved using the qeVal or qeBindCol functions. After all processing is completed on the returned *hstmt*, qeEndSQL must be called to terminate the *hstmt*.

Parsing SQL Statements

DTK's parsing functions allow you to return useful information from the active SQL statement.

The following sample shows how to use the parsing functions. To load this sample in the SAMPLE.EXE program, choose **Parsing SQL Statements** from the Example List.

```
qeSTATUS parse ()  {

/* This routine demonstrates the functions which return individual clauses *     /
/* from a SELECT statement. *  /

       qeHANDLE        hdbc = 0;       /* Handle to database connection *     /
       qeHANDLE        hstmt = 0;      /* Handle to SQL statement execution *     /
       qeSTATUS        res_code;       /* Result code from DTK functions *     /
       qeLPSTR         clause  ;
```

```
/* Call qeLibInit to initialize DTK, check for errors. *    /
      res_code = qeLibInit ()  ;
      if (res_code != qeSUCCESS) return (res_code)   ;

/* Call qeConnect to connect to a data source.  Check to see *    /
/* if hdbc == 0, which indicates that the connection failed. *    /
      hdbc = qeConnect ("DSN=QEDBF")  ;
      if (hdbc == 0) return (err_handler (hdbc, hstmt))   ;

/* Call qeExecSQL to execute the update statement. Check if hstmt == 0, *     /
/* which indicates that the statement did not execute successfully. *     /
      hstmt = qeSQLPrepare (hdbc, "select * from emp where first_name = ?")     ;
      if (hstmt == 0) return (err_handler (hdbc, hstmt))   ;

/* Set the statement parameters. *  /
      res_code = qeSetParamChar (hstmt, 1, "Joe", 20)   ;
      if (res_code != qeSUCCESS) return (err_handler (hdbc, hstmt))    ;

/* Execute the statement. *  /
      res_code = qeSQLExecute (hstmt)  ;
      if (res_code != qeSUCCESS) return (err_handler (hdbc, hstmt))    ;

/* Get the Where clause. *  /
      clause = qeClauseGet (hstmt, qeCLAUSE_WHERE)   ;
      if (qeErr () != qeSUCCESS) return (err_handler (hdbc, hstmt))    ;
      MessageBox (hWnd, clause, "Parsing", MB_OK)   ;

/* Close the statement. *  /
      res_code = qeEndSQL (hstmt)  ;
      hstmt = 0 ;
      if (res_code != qeSUCCESS) return (err_handler (hdbc, hstmt))    ;

/* Call qeDisconnect to disconnect from a data source. *    /
      res_code = qeDisconnect (hdbc)  ;
      hdbc = 0 ;
      if (res_code != qeSUCCESS) return (err_handler (hdbc, hstmt))    ;

/* Call qeLibTerm to free memory allocated by DTK. *   /
      res_code = qeLibTerm ()  ;
      MessageBox (hWnd, "Sample succeeded.", "Parse Functions", MB_OK)    ;
      return (res_code)  ;
}

/* err_handler routine goes here.  *  /
```

DTK's parsing functions allow you to return useful information from the active SQL statement. These functions take the *hstmt* as a parameter and return the information listed in Table 9-2.

Table 9-2. Functions that Parse the Active SQL Statemetn

Function	Returns
qeClauseGet and qeClauseGetBuf	A clause from a Select statement.
qeNativeSQL and qeNativeSQLBuf	The SQL string as translated by the driver.
qeUniqueWhereClause and qeUniqueWhereClauseBuf	A Where clause that uniquely identifies the current record in an active Select statement.

qeUniqueWhereClause and qeUniqueWhereClauseBuf use the columns specified by qeRecSetKey if that function is called, otherwise they generate the list of columns on their own.

ODBC Handle Conversion

These functions convert between DTK handles and ODBC handles, allowing you to call the ODBC driver directly.

ODBC makes available a routine called SQLGetInfo. The DTK functions qeGetODBCInfoChar, qeGetODBCInfoCharBuf, and qeGetODBCInfoLong make it easier to access SQLGetInfo. See the sections on these functions in

Part II for lists of the SQLGetInfo constants they support. There is no guarantee that every database driver will support all of the SQLGetInfo options available.

Table 9-3. Functions that Access SQLGetInfo

Function	Result
qeGetODBCInfoChar and qeGetODBCInfoCharBuf	Returns information about an ODBC connection
qeGetODBCInfoLong	Returns information about an ODBC connection
qeGetODBCHenv	Returns the ODBC environment handle associated with the instance of DTK.
qeGetODBCHstmt	Returns the ODBC *hstmt* that corresponds to the DTK *hstmt*.
qeGetODBCHdbc	Returns the ODBC *hdbc* that corresponds to the DTK *hdbc*.
qeSetODBCHdbc	Sets the ODBC *hdbc* that corresponds to the DTK *hdbc*.

Important: The ODBC handle conversion routines are potentially dangerous. Using the ODBC *hdbc* to change the state of the ODBC connection may create situations that trap. In particular, there is no guarantee of proper behavior when the qeSetODBCHdbc function is called, because DTK cannot know any information about the *hstmt* or *hdbc* involved. Use at your own risk.

Go To

Part 2: Function Reference

10 DTK Functions

This chapter provides a complete, alphabetical reference to the DTK functions. It begins by describing parameter conventions employed in the functions.

Parameter Conventions

Each DTK function has parameters that must be included when you call the functions. The values you send as parameters determine the function's behavior.

Parameter Data Types

DTK's parameter conventions have been designed to work with every Windows and OS/2 product that has a macro or script language with the ability to call functions in DLLs. Only a limited number of types are used for parameters. Also, the DTK functions do not change the values of the parameters. Each function has one result, its return value.

The types used as parameters and return types are as follows:

Type	Description	C data type
INT16	2-byte integer	short
INT32	4-byte integer	long
FLOAT32	floating-point number	float

Type	Description	C data type
FLOAT64	double-precision floating-point number	double
PTRSTR	pointer to a string variable	char far *
PTRINT16	pointer to a 2-byte integer variable	short far *
PTRINT32	pointer to a 4-byte integer variable	long far *
PTRFLT32	pointer to a floating-point variable	float far *
PTRFLT64	pointer to a double-precision floating-point variable	double far *

The *pointer* data types are used in cases where you must pass a pointer to the value. In general, this is handled automatically by the macro or script language.

Some DTK functions return a pointer to a value (such as qeValChar). Some macro and script languages do not allow functions to return pointers. These functions, and considerations for using them, are described in the following section.

Functions That Return Pointers

For DTK functions that return a character string or a decimal number, two forms of the functions are provided (like qeValChar and qeValCharBuf). The first form returns a pointer to the resulting value. The second form (the function name ending in Buf) has an additional parameter which is a pointer to a buffer in which DTK is to put the value.

When a DTK function returns a pointer (as does qeValChar), the pointer refers to a buffer allocated by DTK. DTK allocates global memory and locks it to obtain the pointer value, and returns that pointer. Your program should then copy the value to its own variables.

DTK maintains one buffer per process. Each time a DTK function is called, the contents of the buffer may change, or the buffer memory may be freed. Therefore, be sure to copy character string or decimal values before you call another DTK function from the same process.

If you are running two different programs, such as ToolBook and Excel, and both programs are calling DTK functions, they are separate processes and do not share the same buffers.

If you use the second form of the functions (like qeValCharBuf), then your program must allocate a buffer and pass a pointer to the buffer as a parameter to the DTK function. In this case make sure that the size of the buffer you allocate is large enough to hold the value returned by the function.

If you get an error on a call to a “Buf” function, the information written to the buffer by the call may not to be trusted. You may want to include a routine in your error-handling procedure to flush the buffer of such data.

Functions that Vary by Data Type or Column Type

This manual sometimes collectively refers to a set of functions whose names vary by data type or by column type, but it does not specifically identify the name of functions in the set. For example, it might refer to the *qeVal* functions; however, there is no function named qeVal, although there is a qeValChar, a qeValInt, a qeValLong, and so on.

To give you a better idea of the specific functions that might be referenced this way, the following sections list some but not all of the functions that are sometimes referenced by a collective term.

qeBindCol functions

A set of functions referred to as the *qeBindCol functions* specify value and length variables that receive a column's value and length each time a record is fetched. The qeBindCol function performs no data type conversion on the value being bound; other functions in the set convert the data to the data type suggested by the function name.

The qeBindCol functions are:

- qeBindCol
- qeBindColChar
- qeBindColDecimal
- qeBindColDouble
- qeBindColFloat
- qeBindColInt
- qeBindColLong

qeCol functions

A set of functions referred to as the *qeCol functions* return information about a specified column.

The qeCol functions are:

- qeColAlias and qeColAliasBuf
- qeColDateEnd
- qeColDateStart
- qeColDBType
- qeColDBTypeName and qeColDBTypeNameBuf
- qeColExpr and qeColExprBuf
- qeColName and qeColNameBuf
- qeColPrecision
- qeColScale
- qeColType
- qeColTypeAttr
- qeColumns
- qeColWidth

qePut functions

A set of functions referred to as the *qePut functions* update columns with values that match the column's data type.

The qePut functions are:

- qePutBinary
- qePutChar
- qePutDecimal
- qePutDouble
- qePutFloat
- qePutInt
- qePutLong
- qePutNull
- qePutUsingBindColumns

qeRecSetCondition functions

A set of functions referred to as the *qeRecSetCondition functions* add a search condition to a statement; the comparison value's data type matches the data type of the column being compared against the condition.

The qeRecSetCondition functions are:

- qeRecSetConditionBinary
- qeRecSetConditionChar
- qeRecSetConditionDecimal
- qeRecSetConditionDouble
- qeRecSetConditionFloat
- qeRecSetConditionInt
- qeRecSetConditionLong
- qeRecSetConditionNull

qeVal functions

A set of functions referred to as the *qeVal functions* return column values whose data types match the data type suggested by the function names.

The qeVal functions are:

- qeValChar and qeValCharBuf
- qeValDecimal and qeValDecimalBuf
- qeValDouble
- qeValFloat
- qeValInt
- qeValLong
- qeValMultiChar and qeValMultiCharBuf

Functions

The following sections describe the syntax, parameters, and usage of each DTK database function.

DTK version 2.*x* includes functions that will be obsolete in future versions. These functions are listed in Appendix E, "Compatibility Issues," on page 553.

DTK also provides a set of data conversion functions that are described separately in Appendix A, "Data Conversion Functions," on page 493.

qeAppendSQL

qeAppendSQL appends text to the SQL buffer.

Syntax

```
int16 res_code qeAppendSQ L (int16 hdbc, ptrstr
partial_stmt)
```

Description

Some macro languages cannot send an entire SQL statement to qeExecSQL due to limits in the lengths of strings they support. For example, Excel strings are limited to 255 characters. Since many Select statements are longer than 255 characters, Excel cannot send long Select statements to qeExecSQL.

Internally, DTK maintains one SQL buffer per *hdbc*.

SQL replaces the contents of the SQL buffer with the partial statement sent as a parameter. Each subsequent call to qeAppendSQL appends text to the SQL buffer. Once the complete SQL statement has been sent to the DTK API, you can call qeSQLPrepare (with "" as the *sql_stmt* value) or qeExecSQL to use the SQL statement saved in the SQL buffer.

Parameters

hdbc is the handle to the database connection returned by qeConnect.

partial_stmt is the character string to append to the contents of the SQL buffer. It must contain part of a SQL statement.

res_code is the result code returned by qeAppendSQL, which returns the same set of result codes as qeErr. See Appendix D, "Result and Error Message Codes," on page 537 for a list of these result codes.

Example

To send a SQL Server database a Select statement in sections and execute it:

```
hdbc = qeConnect ("DSN=QESS;UID=sa;SRVR=PION1")     ;
...
res_code = qeSetSQL (hdbc, "SELECT *")    ;
res_code = qeAppendSQL (hdbc, " FROM emp")    ;
res_code = qeAppendSQL (hdbc, " ORDER BY last_name"      )
hstmt = qeExecSQL (hdbc, "")   ;
...
res_code = qeEndSQL (hstmt)   ;
res_code = qeDisconnect (hdbc)   ;
```

See Also

qeExecSQL, qeSetSQL.

qeApplyAll

qeApplyAll updates the database with all deferred record changes.

Syntax

```
int16 res_code qeApplyAl l (int16 hstmt)
```

Description

When qeSetAutoUpdate is set to qeAUTOUPD_DEFER (2) to enable record changes to be deferred—saved but not updated in the database, qeApplyAll updates the database with all changes that have been performed on the statement.

You can call qeNumModRecs to determine the number of records affected.

Parameters

hstmt is the handle to the statement returned by qeExecSQL or qeSQLPrepare.

res_code is the result code returned by qeApplyAll, which returns the same set of result codes as qeErr. See Appendix D, “Result and Error Message Codes,” on page 537 for a list of these result codes.

Example

```
hdbc = qeConnect ("DSN=QEDBF")   ;
res_code = qeSetAutoUpdate (hdbc, qeAUTOUPD_DEFER)     ;
hstmt = qeExecSQL (hdbc, "SELECT * FROM emp")    ;

res_code = qeFetchNext (hstmt)   ;
res_code = qePutChar (hstmt, 1, "", "Rachel")    ;
res_code = qeFetchNext (hstmt)   ;
res_code = qePutChar (hstmt, 1, "", "Eddie")    ;
res_code = qeFetchNext (hstmt)   ;

res_code = qeApplyAll (hstmt)   ;
res_code = qeEndSQL (hstmt)   ;
res_code = qeDisconnect (hdbc)   ;
```

See Also

qeSetAutoUpdate, qeUndoAll.

qeBeginTran

qeBeginTran begins a database transaction.

Syntax

`int16 res_code qeBeginTra n (int16 hdbc)`

Description

qeBeginTran starts a transaction on a database connection. Once a transaction begins, the SQL Insert, Update, and Delete statements that are executed using qeExecSQL are not committed to the database until qeCommit is called.

qeCommit saves the changes that have been made since qeBeginTran was called and frees all database locks.

Alternatively, qeRollback discards the changes that have been made since qeBeginTran was called and frees all database locks.

Parameters

hdbc is the handle to the database connection returned by qeConnect.

res_code is the result code returned by qeBeginTran, which returns the same set of result codes as qeErr. See Appendix D, "Result and Error Message Codes," on page 537 for a list of these result codes.

Example

To commit changes made to a SQL Server database:

```
hdbc = qeConnect ("DSN=QESS;UID=sa;SRVR=PION1")    ;
...
res_code = qeBeginTran (hdbc)   ;
hstmt = qeExecSQL (hdbc   ,
    "UPDATE emp SET salary = salary * 1.1")    ;
res_code = qeEndSQL (hstmt)   ;
res_code = qeCommit (hdbc)   ;
res_code = qeDisconnect (hdbc)   ;
```

Notes

If you execute an Insert, Update, or Delete statement without first calling qeBeginTran, the database changes are automatically committed and no database locks are held.

You cannot have more than one simultaneous transaction active on a database connection. Once you call qeBeginTran, you must call either qeCommit or qeRollback before you call qeBeginTran again on the same database connection.

Once you call qeBeginTran, you must call either qeCommit or qeRollback before you call qeDisconnect. Calling qeDisconnect with an active transaction results in an error.

See Also

qeCommit, qeRollback.

qeBindCol

qeBindCol specifies value and length variables that receive a column's value and length each time a record is fetched.

Syntax

```
int16 res_code qeBindCo l (
    int16    hstmt,
    int16    col_num,
    ptrstr   value_ptr,
    ptrint32 len_ptr)
```

Description

qeBindCol specifies the value and length variables in your program that are to receive a column's value and length each time a record is fetched.

qeBindCol performs no data type conversion on the value being bound, so it is most useful for binding where no conversion is necessary.

You must bind all columns in the statement in the order they occur.

Parameters

hstmt is the handle to the statement returned by qeExecSQL, qeSQLPrepare, or data dictionary function calls.

col_num is the column number whose variables are specified. The first column number is 1.

value_ptr points to the variable that is to receive the column's value when a record is fetched.

len_ptr points to the variable that is to receive the column value's length when a record is fetched. You can use this variable to determine whether a fetch retrieves truncated or null data (qeNULL_DATA (-2) and qeTRUNCATION (-1) may be returned). Also, when qeBindCol is called, this variable must contain the size of the *value_ptr* variable in bytes.

res_code is the result code returned by qeBindCol, which returns the same set of result codes as qeErr. See Appendix D, "Result and Error Message Codes," on page 537 for a list of these result codes.

Example

To get the first and last names of each employee in the dBASE employee file:

```
char     last_name[11]  ;
long     ln_length  ;
char     first_name[9]  ;
long     fn_length  ;
hdbc = qeConnect ("DSN=QEDBF")   ;
...
hstmt = qeExecSQL (hdbc, "SELECT first_name, last_name
    FROM emp") ;
fn_length = 9 ;
qeBindCol (hstmt, 1, first_name, &fn_length)    ;
ln_length = 11 ;
qeBindCol (hstmt, 2, last_name, &ln_length)    ;
while (qeFetchNext (hstmt) == 0)    {
    /* qeFetchNext has automatically filled *    /
    /* first_name and last_name with the *    /
    /* values from the record, and fn_length *    /
    /* and ln_length with the lengths of the *    /
    /* two values. * /
...
}
res_code = qeEndSQL (hstmt)   ;
res_code = qeDisconnect (hdbc)   ;
```

See Also

qeFetchNext, qeFetchPrev, qeFetchRandom, qeVal functions.

qeBindColChar

qeBindColChar specifies value and length variables that receive a column's value and length each time a record is fetched.

Syntax

```
int16  res_code qeBindColCha r (
    int16    hstmt,
    int16    col_num,
    ptrstr   value_ptr,
    ptrint32 len_ptr,
    ptrstr   fmt_string)
```

Description

qeBindColChar specifies the value and length variables in your program that are to receive a column's value and length each time a record is fetched. Data is converted to a character string, using a format string if supplied.

You must bind all columns in the statement in the order they occur.

Parameters

hstmt is the handle to the statement returned by qeExecSQL, qeSQLPrepare, or data dictionary function calls.

col_num is the column number whose variables are specified. The first column number is 1.

value_ptr points to the variable that is to receive the null-terminated character string value for the column when a record is fetched.

len_ptr points to the variable that is to receive the column value's length when a record is fetched. You can use this variable to determine whether a fetch retrieves truncated or null data (qeNULL_DATA (-2) and qeTRUNCATION (-1) may be returned). Also, when qeBindColChar is called, this variable must contain the size of the *value_ptr* variable in bytes.

fmt_string is a string used to control formatting of dates and numbers into a character string.

res_code is the result code returned by qeBindColChar, which returns the same set of result codes as qeErr. See Appendix D, “Result and Error Message Codes,” on page 537 for a list of these result codes.

Example

```
char fname[31], lname[31]  ;

hdbc = qeConnect ("DSN=QEDBF")   ;
hstmt = qeExecSQL (hdbc, "SELECT first_name    ,
    last_name FROM emp")  ;
fnamelen = 30 ;
lnamelen = 30 ;
res_code = qeBindColChar (hstmt, 1, fname, &fnamelen,
"");
res_code = qeBindColChar (hstmt, 2, lname, &lnamelen,
"");
while (qeFetchNext (hstmt) == qeSUCCESS)     {
    strcpy (name, fname)  ;
    strcat (name, lname)  ;
}
res_code = qeEndSQL (hstmt)   ;
res_code = qeDisconnect (hdbc)   ;
```

qeBindColDecimal

qeBindColDecimal specifies value and length variables that receive a column's value and length each time a record is fetched.

Syntax

```
int16 res_code qeBindColDecima l (
    int16    hstmt,
    int16    col_num,
    ptrstr   value_ptr,
    ptrint32 len_ptr,
    int16    precision,
    int1 6   scale)
```

Description

qeBindColDecimal specifies value and length variables that receive a column's value and length each time a record is fetched. Data is converted to a decimal value with the specified precision and scale.

You must bind all columns in the statement in the order they occur.

Parameters

hstmt is the handle to the statement returned by qeExecSQL, qeSQLPrepare, or data dictionary function calls.

col_num is the column number whose variables are specified. The first column number is 1.

value_ptr points to the variable that is to receive the column's value when a record is fetched.

len_ptr points to the variable that is to receive the column value's length when a record is fetched. You can use this variable to determine whether a fetch retrieves truncated or null data (qeNULL_DATA (-2) and qeTRUNCATION (-1) may be returned).

precision is the number of significant digits in the result.

scale specifies the location of the decimal point in the result.

res_code is the result code returned by qeBindColDecimal, which returns the same set of result codes as qeErr. See Appendix D, “Result and Error Message Codes,” on page 537 for a list of these result codes.

Example

```
char salary[10] ;

hdbc = qeConnect ("DSN=QEDBF")   ;
hstmt = qeExecSQL (hdbc, "SELECT salary FROM emp")    ;
salarylen = 10 ;
res_code = qeBindColDecimal (hstmt, 1, salary,
    &salarylen, 9, 2) ;
while (qeFetchNext (hstmt) == qeSUCCESS)    {
/* salary now holds the value of the SALARY *    /
/* field of the current record. *   /
...
}
res_code = qeEndSQL (hstmt)   ;
res_code = qeDisconnect (hdbc)   ;
```

qeBindColDouble

qeBindColDouble specifies value and length variables that receive a column's value and length each time a record is fetched.

Syntax

```
int16 res_code qeBindColDoubl e (
    int16    hstmt,
    int16    col_num,
    ptrflt64 value_ptr,
    ptrint32 len_ptr)
```

Description

qeBindColDouble specifies value and length variables that receive a column's value and length each time a record is fetched. Data is converted to a double-precision floating-point value.

You must bind all columns in the statement in the order they occur.

Parameters

hstmt is the handle to the statement returned by qeExecSQL, qeSQLPrepare, or data dictionary function calls.

col_num is the column number whose variables are specified. The first column number is 1.

value_ptr points to the variable that is to receive the column's value when a record is fetched.

len_ptr points to the variable that is to receive the column value's length when a record is fetched. You can use this variable to determine whether a fetch retrieves truncated or null data (qeNULL_DATA (-2) and qeTRUNCATION (-1) may be returned).

res_code is the result code returned by qeBindColDouble, which returns the same set of result codes as qeErr. See Appendix D, "Result and Error Message Codes," on page 537 for a list of these result codes.

Example

```
double  salary ;

hdbc = qeConnect ("DSN=QEDBF")   ;
hstmt = qeExecSQL (hdbc, "SELECT salary FROM emp")     ;
salarylen = 8 ;
res_code = qeBindColDouble (hstmt, 1, &salary    ,
    &salarylen) ;
while (qeFetchNext (hstmt) == 0)    {
/* salary now holds the value of the SALARY *    /
/* field of the current record. *   /
...
}
res_code = qeEndSQL (hstmt)   ;
res_code = qeDisconnect (hdbc)   ;
```

qeBindColFloat

qeBindColFloat specifies value and length variables that receive a column's value and length each time a record is fetched.

Syntax

```
int16  res_code qeBindColFloa t (
    int16    hstmt,
    int16    col_num,
    ptrflt32 value_ptr,
    ptrint32 len_ptr)
```

Description

qeBindColFloat specifies value and length variables that receive a column's value and length each time a record is fetched. Data is converted to a single-precision floating-point value.

You must bind all columns in the statement in the order they occur.

Parameters

hstmt is the handle to the statement returned by qeExecSQL, qeSQLPrepare, or data dictionary function calls.

col_num is the column number whose variables are specified. The first column number is 1.

value_ptr points to the variable that is to receive the column's value when a record is fetched.

len_ptr points to the variable that is to receive the column value's length when a record is fetched. You can use this variable to determine whether a fetch retrieves truncated or null data (qeNULL_DATA (-2) and qeTRUNCATION (-1) may be returned).

res_code is the result code returned by qeBindColFloat, which returns the same set of result codes as qeErr. See Appendix D, "Result and Error Message Codes," on page 537 for a list of these result codes.

Example

```
float  salary ;

hdbc = qeConnect ("DSN=QEDBF")   ;
hstmt = qeExecSQL (hdbc, "SELECT salary FROM emp")     ;
salarylen = 8 ;
res_code = qeBindColFloat (hstmt, 1, &salary,
&salarylen) ;
while (qeFetchNext (hstmt) == 0)    {
/* salary now holds the value of the SALARY *    /
/* field of the current record. *   /
...
}
res_code = qeEndSQL (hstmt)   ;
res_code = qeDisconnect (hdbc)   ;
```

qeBindColInt

qeBindColInt specifies value and length variables that receive a column's value and length each time a record is fetched.

Syntax

```
int16  res_code qeBindColIn t (
    int16    hstmt,
    int16    col_num,
    ptrint16 value_ptr,
    ptrint32 len_ptr)
```

Description

qeBindColInt specifies value and length variables that receive a column's value and length each time a record is fetched. Data is converted to a 2-byte integer.

You must bind all columns in the statement in the order they occur.

Parameters

hstmt is the handle to the statement returned by qeExecSQL, qeSQLPrepare, or data dictionary function calls.

col_num is the column number whose variables are specified. The first column number is 1.

value_ptr points to the variable that is to receive the column's value when a record is fetched.

len_ptr points to the variable that is to receive the column value's length when a record is fetched. You can use this variable to determine whether a fetch retrieves truncated or null data (qeNULL_DATA (-2) and qeTRUNCATION (-1) may be returned).

res_code is the result code returned by qeBindColInt, which returns the same set of result codes as qeErr. See Appendix D, “Result and Error Message Codes,” on page 537 for a list of these result codes.

Example

```
int  salary ;

hdbc = qeConnect ("DSN=QEDBF")   ;
hstmt = qeExecSQL (hdbc, "SELECT salary FROM emp")    ;
salarylen = 2 ;
res_code = qeBindColInt (hstmt, 1, &salary, &salarylen)     ;
while (qeFetchNext (hstmt) == 0)    {
/* salary now holds the value of the SALARY *    /
/* field of the current record. *   /
...
}
res_code = qeEndSQL (hstmt)   ;
res_code = qeDisconnect (hdbc)   ;
```

qeBindColLong

qeBindColLong specifies value and length variables that receive a column's value and length each time a record is fetched.

Syntax

```
int16 res_code qeBindColLon g (
    int16    hstmt,
    int16    col_num,
    ptrint32 value_ptr,
    ptrint32 len_ptr)
```

Description

qeBindColLong specifies value and length variables that receive a column's value and length each time a record is fetched. Data is converted to a 4-byte integer.

You must bind all columns in the statement in the order they occur.

Parameters

hstmt is the handle to the statement returned by qeExecSQL, qeSQLPrepare, or data dictionary function calls.

col_num is the column number whose variables are specified. The first column number is 1.

value_ptr points to the variable that is to receive the column's value when a record is fetched.

len_ptr points to the variable that is to receive the column value's length when a record is fetched. You can use this variable to determine whether a fetch retrieves truncated or null data (qeNULL_DATA (-2) and qeTRUNCATION (-1) may be returned).

res_code is the result code returned by qeBindColLong, which returns the same set of result codes as qeErr. See Appendix D, "Result and Error Message Codes," on page 537 for a list of these result codes.

Example

```
long  salary ;

hdbc = qeConnect ("DSN=QEDBF")   ;
hstmt = qeExecSQL (hdbc, "SELECT salary FROM emp")    ;
salarylen = 4 ;
res_code = qeBindColLong (hstmt, 1, &salary, &salarylen)     ;


while (qeFetchNext (hstmt) == 0)    {
/* salary now holds the value of the SALARY *    /
/* field of the current record. *   /
...
}
res_code = qeEndSQL (hstmt)   ;
res_code = qeDisconnect (hdbc)   ;
```

qeBindParamBinary

qeBindParamBinary binds a parameter to a binary buffer.

Syntax

```
int16  res_code qeBindParamBinar y (
    int16    hstmt,
    int16    param_num,
    ptrstr   param_val,
    ptrint32 param_len)
```

Description

qeBindParamBinary binds the value of a parameter in a SQL statement to a buffer that holds a binary value. It also binds a variable that holds the length of the *param_val* buffer at qeSQLExecute time.

For input and input/output parameters, you must place the value of the parameter into the buffer before executing the statement. The DTK uses this parameter value in place of the parameter itself in execution of the SQL statement or stored procedure. For stored procedure output and input/output parameters, the value assigned to the parameter during the execution of the stored procedure is placed in this buffer by the DTK.

Before calling qeBindParamBinary, you must call qeSQLPrepare to prepare the SQL statement for which you are supplying parameters. You must give values to all input and input/output parameters before calling qeSQLExecute.

DTK saves the value and length pointer; they must be valid when you call qeSQLExecute. This parameter continues to point to this value until qeSetParamNull or a qeSetParam or qeBindParam function is called again for this parameter. All parameters with the same name as the one identified by *param_num* are affected.

Set *param_len* to the maximum size of the binary value before calling qeBindParamBinary.

Parameters

hstmt is the handle to the statement returned by qeSQLPrepare.

param_num is the position of the parameter to be set.

param_val points to the value of the parameter. For input parameters, *param_val* points to the buffer that holds the value to be assigned to the parameter. For output parameters, *param_val* points to the buffer that holds the value assigned to the parameter by the stored procedure after qeSQLExecute is called. For an input/output parameter, *param_val* plays both roles.

param_len points to a LONG variable that holds the length of *param_val* when qeSQLExecute is called. For input parameters, if you set *param_len* to qeNULL_DATA, the parameter is set to null when you call qeSQLExecute. For output parameters, *param_len* holds the length of the parameter value after qeSQLExecute is called. Also for output parameters, *param_len* can be used to determine whether the data is NULL or truncated (qeNULL_DATA (-2) and qeTRUNCATION (-1) may be returned).

res_code is the result code returned by the function, which returns the same set of result codes as qeErr. See Appendix D, “Result and Error Message Codes,” on page 537 for a list of these result codes.

Example

```
hdbc = qeConnect ("DSN=QEDBF")   ;
hstmt = qeSQLPrepare (hdbc, "INSERT INTO em    p
    MEMO) VALUES (?)")  ;
bin_length = 10000; /* Max length of bindata *     /
res_code = qeBindParamBinary (hstmt, 1, bindata     ,
    &bin_length) ;
/* Set bindata to your binary data. *    /
...
bin_length = 4323; /* # of bytes of binary data passed *     /
res_code = qeSQLExecute (hstmt)    ;
res_code = qeEndSQL (hstmt)   ;
res_code = qeDisconnect (hdbc)   ;
```

qeBindParamChar

qeBindParamChar binds a parameter to a character buffer.

Syntax

```
int16  res_code qeBindParamCha r (
    int16    hstmt,
    int16    param_num,
    ptrstr   param_val,
    ptrint32 param_len)
```

Description

qeBindParamChar binds the value of a parameter in a SQL statement to a buffer that holds a character value.

For input and input/output parameters, you must place the value of the parameter into the buffer before executing the statement. The DTK uses this parameter value in place of the parameter itself in execution of the SQL statement or stored procedure. For stored procedure output and input/output parameters, the value assigned to the parameter during the execution of the stored procedure is placed in this buffer by the DTK.

Before calling qeBindParamChar, you must call qeSQLPrepare to prepare the SQL statement for which you are supplying parameters. You must give values to all input and input/output parameters before calling qeSQLExecute.

DTK saves the value and length pointer; they must be valid when you call qeSQLExecute. This parameter continues to point to this value until qeSetParamNull or a qeSetParam or qeBindParam function is called again for this parameter. All parameters with the same name as the one identified by *param_num* are affected.

Set *param_len* to the maximum size of the character value (the length of the associated column) before calling qeBindParamChar. This setting determines whether the buffer that holds the parameter is of varying character or long varying character type. If *param_len* is less than or equal to the largest character string allowed by the database, then the parameter is varying character type. If greater, it is long varying character type.

Important: A mismatch between the parameter type and the database column type (varying character versus long varying character) may cause unusual problems for some database drivers, for which no errors are returned.

Before calling qeSQLExecute, you must reassign the value of *param_len* to that of the length of *param_val*.

Parameters

hstmt is the handle to the statement returned by qeSQLPrepare.

param_num is the position of the parameter to be set.

param_val points to the value of the parameter. For input parameters, *param_val* points to a buffer that holds the character value to be assigned to the parameter when qeSQLExecute is called. For output parameters, *param_val* points to the buffer that holds the character value assigned to the parameter by the stored procedure after qeSQLExecute is called. For an input/output parameter, *param_val* plays both roles.

param_len points to a 4-byte long integer variable. When qeBindParamChar is called, *param_len* must hold the length of the column associated with the parameter. However, before calling qeSQLExecute, you must reassign the value of *param_len* to that of the length of *param_val*. For input parameters, if you set *param_len* to qeNULL_DATA, the parameter is set to null when you call qeSQLExecute. For output parameters, *param_len* holds the length of the parameter value after qeSQLExecute is called. Also for output parameters, *param_len* can be used to determine whether the data is NULL or truncated (qeNULL_DATA (-2) and qeTRUNCATION (-1) may be returned).

res_code is the result code returned by the function, which returns the same set of result codes as qeErr. See Appendix D, "Result and Error Message Codes," on page 537 for a list of these result codes.

Example

```
hdbc = qeConnect ("DSN=QEORA;DLG=2")   ;

/* ?IdToName inputs the employee's id and output'     s
   the name of the employee's dept. *    /
hstmt = qeSQLPrepare (hdbc, "{CALL GetEmployeeDep     t
        (?IdToName)}")  ;
char_len = 10 ;
res_code = qeBindParamChar (hstmt, 1, dept, &char_len)     ;
res_code = qeSetParamIOType (hstmt, 1, qePARAM_INOUT)     ;
strcpy (dept, "E10297")  ;
res_code = qeSQLExecute (hstmt)   ;

/* The name of the employee's department (?IdToName) i     s
   in the dept buffer*  /
res_code = qeEndSQL(hstmt)   ;
res_code = qeDisconnect (hstmt)   ;
res_code = qeEndSQL (hstmt)   ;
res_code = qeDisconnect (hdbc)   ;
```

qeBindParamDate

qeBindParamDate binds a parameter to a date buffer.

Syntax

```
int16  res_code qeBindParamDat e (
    int16    hstmt,
    int16    param_num,
    ptrstr   param_val,
    ptrint32 param_len)
```

Description

qeBindParamDate binds the value of a parameter in a SQL statement to a buffer that holds a date value.

For input and input/output parameters, you must place the value of the parameter into the buffer before executing the statement. The DTK uses this parameter value in place of the parameter itself in execution of the SQL statement or stored procedure. For stored procedure output and input/output parameters, the value assigned to the parameter during the execution of the stored procedure is placed in this buffer by the DTK.

Before calling qeBindParamDate, you must call qeSQLPrepare to prepare the SQL statement for which you are supplying parameters. You must give values to all input and input/output parameters before calling qeSQLExecute.

DTK saves the value and length pointer; they must be valid when you call qeSQLExecute. This parameter continues to point to this value until qeSetParamNull or a qeSetParam or qeBindParam function is called again for this parameter. All parameters with the same name as the one identified by *param_num* are affected.

Parameters

hstmt is the handle to the statement returned by qeSQLPrepare.

param_num is the position of the parameter to be set.

param_val points to the value of the parameter. For input parameters, *param_val* points to a buffer that holds the 26-byte date value to be assigned to the parameter when qeSQLExecute is called. For output parameters, *param_val* points to the buffer that holds the date value assigned to the parameter by the stored procedure after qeSQLExecute is called. For an input/output parameter, *param_val* plays both roles.

param_len is the date precision of the value assigned to this parameter. Set it to 10 before calling qeBindParamDate. For input parameters, if you set *param_len* to qeNULL_DATA, the parameter is set to null when you call qeSQLExecute. For output parameters, *param_len* holds the length of the parameter value after qeSQLExecute is called. Also for output parameters, *param_len* can be used to determine whether the data is NULL or truncated (qeNULL_DATA (-2) and qeTRUNCATION (-1) may be returned).

res_code is the result code returned by the function, which returns the same set of result codes as qeErr. See Appendix D, "Result and Error Message Codes," on page 537 for a list of these result codes.

Example

```
hdbc = qeConnect ("DSN=QEDBF")   ;
hstmt = qeSQLPrepare (hdbc, "SELECT * FROM em    p
    WHERE hire_date = ?")  ;
date_len = 10 ;
res_code = qeBindParamDate (hstmt, 1, hire_date,
&date_len) ;
strcpy (hire_date, "1983-06-01 00:00:00:000000")    ;
res_code = qeSQLExecute (hstmt)   ;
res_code = qeEndSQL (hstmt)   ;
res_code = qeDisconnect (hdbc)   ;
```

qeBindParamDateTime

qeBindParamDateTime binds a parameter to a date-time buffer.

Syntax

```
int16 res_code qeBindParamDateTime (
    int16    hstmt,
    int16    param_num,
    ptrstr   param_val,
    ptrint32 param_len)
```

Description

qeBindParamDateTime binds the value of a parameter in a SQL statement to a buffer that holds a date-time value.

For input and input/output parameters, you must place the value of the parameter into the buffer before executing the statement. The DTK uses this parameter value in place of the parameter itself in execution of the SQL statement or stored procedure. For stored procedure output and input/output parameters, the value assigned to the parameter during the execution of the stored procedure is placed in this buffer by the DTK.

Before calling qeBindParamDateTime, you must call qeSQLPrepare to prepare the SQL statement for which you are supplying parameters. You must give values to all input and input/output parameters before calling qeSQLExecute.

DTK saves the value and length pointer; they must be valid when you call qeSQLExecute. This parameter continues to point to this value until qeSetParamNull or a qeSetParam or qeBindParam function is called again for this parameter. All parameters with the same name as the one identified by *param_num* are affected.

Parameters

hstmt is the handle to the statement returned by qeExecSQL.

param_num is the position of the parameter to be set.

param_val points to the value of the parameter. For input parameters, *param_val* points to a buffer that will hold the 26-byte date-time value assigned to the parameter when qeSQLExecute is called. For output parameters, *param_val* points to the buffer that holds the date-time value assigned to the parameter by the stored procedure after qeSQLExecute is called. For an input/output parameter, *param_val* plays both roles.

param_len is the date-time precision of the value assigned to this parameter. Set it to 16, 19, 23, or 26 before calling qeBindParamDateTime. For input parameters, if you set *param_len* to qeNULL_DATA, the parameter is set to null when you call qeSQLExecute. For output parameters, *param_len* holds the length of the parameter value after qeSQLExecute is called. Also for output parameters, *param_len* can be used to determine whether the data is NULL or truncated (qeNULL_DATA (-2) and qeTRUNCATION (-1) may be returned).

res_code is the result code returned by the function, which returns the same set of result codes as qeErr. See Appendix D, "Result and Error Message Codes," on page 537 for a list of these result codes.

Example

```
hdbc = qeConnect ("DSN=QEDBF")   ;
hstmt = qeSQLPrepare (hdbc, "SELECT * FROM em    p
    WHERE hire_date = ?")  ;
dt_len = 26 ;
res_code = qeBindParamDateTime (hstmt, 1    ,
    hire_date, &dt_len)  ;
strcpy (hire_date, "1983-06-01 12:00:00:000000")    ;
res_code = qeSQLExecute (hstmt)   ;
res_code = qeEndSQL (hstmt)   ;
res_code = qeDisconnect (hdbc)   ;
```

qeBindParamDecimal

qeBindParamDecimal binds a parameter to a decimal buffer.

Syntax

```
int16 res_code qeBindParamDecima l (
    int16    hstmt,
    int16    param_num,
    ptrstr   param_val,
    ptrint32 param_len,
    int16    scale)
```

Description

qeBindParamDecimal binds the value of a parameter in a SQL statement to a buffer that holds a decimal value. The value is formatted based on the values of *precision* and *scale*.

For input and input/output parameters, you must place the value of the parameter into the buffer before executing the statement. The DTK uses this parameter value in place of the parameter itself in execution of the SQL statement or stored procedure. For stored procedure output and input/output parameters, the value assigned to the parameter during the execution of the stored procedure is placed in this buffer by the DTK.

Before calling qeBindParamDecimal, you must call qeSQLPrepare to prepare the SQL statement for which you are supplying parameters. You must give values to all input and input/output parameters before calling qeSQLExecute.

DTK saves the value and length pointer; they must be valid when you call qeSQLExecute. This parameter continues to point to this value until qeSetParamNull or a qeSetParam or qeBindParam function is called again for this parameter. All parameters with the same name as the one identified by *param_num* are affected.

Parameters

hstmt is the handle to the statement returned by qeSQLPrepare.

param_num is the position of the parameter to be set.

param_val points to the value of the parameter. For input parameters, *param_val* points to a buffer that holds the value to be assigned to the parameter. For output parameters, *param_val* points to the buffer that holds the value assigned to the parameter by the stored procedure after qeSQLExecute is called. For an input/output parameter, *param_val* plays both roles.

param_len is the number of bytes in the decimal value. Set it before calling qeBindParamDecimal. For input parameters, if you set *param_len* to qeNULL_DATA, the parameter will be set to null when you call qeSQLExecute. For output parameters, *param_len* holds the length of the parameter value after qeSQLExecute is called. Also for output parameters, *param_len* can be used to determine whether the data is NULL or truncated (qeNULL_DATA (-2) and qeTRUNCATION (-1) may be returned).

scale specifies the location of the decimal point in the decimal value.

res_code is the result code returned by the function, which returns the same set of result codes as qeErr. See Appendix D, "Result and Error Message Codes," on page 537 for a list of these result codes.

Example

```
hdbc = qeConnect ("DSN=QEDBF")   ;
hstmt = qeSQLPrepare (hdbc, "SELECT * FROM em     p
    WHERE salary = ?")  ;
num_length = 7 ;
res_code = qeBindParamDecimal (hstmt, 1    ,
    num_data, &num_length, 2)   ;
qeCharToDecimalBuf (num_data, 7, 2, "320000", "")     ;
res_code = qeSQLExecute (hstmt)   ;
res_code = qeEndSQL (hstmt)   ;
res_code = qeDisconnect (hdbc)   ;
```

qeBindParamDouble

qeBindParamDouble binds a parameter to a double-precision floating-point buffer.

Syntax

```
int16 res_code qeBindParamDoubl e (
    int16    hstmt,
    int16    param_num,
    ptrflt64 param_val,
    ptrint32 param_len)
```

Description

qeBindParamDouble binds the value of a parameter in a SQL statement to a buffer that holds a double-precision floating-point value.

For input and input/output parameters, you must place the value of the parameter into the buffer before executing the statement. The DTK uses this parameter value in place of the parameter itself in execution of the SQL statement or stored procedure. For stored procedure output and input/output parameters, the value assigned to the parameter during the execution of the stored procedure is placed in this buffer by the DTK.

Before you call qeBindParamDouble, you must call qeSQLPrepare to prepare the SQL statement for which you are supplying parameters. You must give values to all input and input/output parameters before calling qeSQLExecute.

DTK saves the value and length pointer; they must be valid when you call qeSQLExecute. This parameter continues to point to this value until qeSetParamNull or a qeSetParam or qeBindParam function is called again for this parameter. All parameters with the same name as the one identified by *param_num* are affected.

Parameters

hstmt is the handle to the statement returned by qeSQLPrepare.

param_num is the position of the parameter to be set.

param_val points to the value of the parameter. For input parameters, *param_val* points to a buffer to hold the double-precision floating-point value to be assigned to the parameter. For output parameters, *param_val* points to the buffer that holds the double-precision floating-point value assigned to the parameter by the stored procedure after qeSQLExecute is called. For an input/output parameter, *param_val* plays both roles.

param_len lets you set the double-precision floating-point parameter to null. For assigning a double-precision floating-point parameter value, set *param_len* to 0 before calling qeBindParamDouble. For input parameters, if you set *param_len* to qeNULL_DATA, the parameter is set to null when you call qeSQLExecute. For output parameters, *param_len* holds the length of the parameter value after qeSQLExecute is called. Also for output parameters, *param_len* can be used to determine whether the data is NULL or truncated (qeNULL_DATA (-2) and qeTRUNCATION (-1) may be returned).

res_code is the result code returned by the function, which returns the same set of result codes as qeErr. See Appendix D, “Result and Error Message Codes,” on page 537 for a list of these result codes.

Example

```
hdbc = qeConnect ("DSN=QEDBF")   ;
hstmt = qeSQLPrepare (hdbc, "SELECT * FROM em    p
    WHERE salary = ?")  ;
num_length = 0 ;
res_code = qeBindParamDouble (hstmt, 1, num_data,
    &num_length) ;
num_data = 32000.00  ;
res_code = qeSQLExecute (hstmt)   ;
res_code = qeEndSQL (hstmt)   ;
res_code = qeDisconnect (hdbc)   ;
```

qeBindParamFloat

qeBindParamFloat binds a parameter to a single-precision floating-point buffer.

Syntax

```
int16  res_code qeBindParamFloa t (
    int16    hstmt,
    int16    param_num,
    ptrflt32 param_val,
    ptrint32 param_len)
```

Description

qeBindParamFloat binds the value of a parameter in a SQL statement to a buffer that will hold a single-precision floating-point value.

For input and input/output parameters, you must place the value of the parameter into the buffer before executing the statement. The DTK uses this parameter value in place of the parameter itself in execution of the SQL statement or stored procedure. For stored procedure output and input/output parameters, the value assigned to the parameter during the execution of the stored procedure is placed in this buffer by the DTK.

Before calling qeBindParamFloat, you must call qeSQLPrepare to prepare the SQL statement for which you are supplying parameters. You must give values to all input and input/output parameters before calling qeSQLExecute.

DTK saves the value and length pointer; they must be valid when you call qeSQLExecute. This parameter continues to point to this value until qeSetParamNull or a qeSetParam or qeBindParam function is called again for this parameter. All parameters with the same name as the one identified by *param_num* are affected.

Parameters

hstmt is the handle to the statement returned by qeSQLPrepare.

param_num is the position of the parameter to be set.

param_val points to the value of the parameter. For input parameters, *param_val* points to a buffer to hold the single-precision floating-point value to be assigned to the parameter. For output parameters, *param_val* points to the buffer that holds the single-precision floating-point value assigned to the parameter by the stored procedure after qeSQLExecute is called. For an input/output parameter, *param_val* plays both roles.

param_len lets you set the floating-point parameter to null. For assigning a floating-point parameter value, set *param_len* to 0 before calling qeBindParamFloat. For input parameters, if you set *param_len* to qeNULL_DATA, the parameter is set to null when you call qeSQLExecute. For output parameters, *param_len* holds the length of the parameter value after qeSQLExecute is called. Also for output parameters, *param_len* can be used to determine whether the data is NULL or truncated (qeNULL_DATA (-2) and qeTRUNCATION (-1) may be returned).

res_code is the result code returned by the function, which returns the same set of result codes as qeErr. See Appendix D, “Result and Error Message Codes,” on page 537 for a list of these result codes.

Example

```
hdbc = qeConnect ("DSN=QEDBF")   ;
hstmt = qeSQLPrepare (hdbc, "SELECT * FROM em    p
    WHERE salary = ?")  ;
num_length = 0 ;
res_code = qeBindParamFloat (hstmt, 1, num_data     ,
    &num_length) ;
num_data = 32000.00  ;
res_code = qeSQLExecute (hstmt)   ;
res_code = qeEndSQL (hstmt)   ;
res_code = qeDisconnect (hdbc)   ;
```

qeBindParamInt

qeBindParamInt binds a parameter to a 2-byte integer buffer.

Syntax

```
int16  res_code qeBindParamIn t (
    int16    hstmt,
    int16    param_num,
    ptrint16 param_val,
    ptrint32 param_len)
```

Description

qeBindParamInt binds the value of a parameter in a SQL statement to a buffer that will hold a 2-byte integer value.

For input and input/output parameters, you must place the value of the parameter into the buffer before executing the statement. The DTK uses this parameter value in place of the parameter itself in execution of the SQL statement or stored procedure. For stored procedure output and input/output parameters, the value assigned to the parameter during the execution of the stored procedure is placed in this buffer by the DTK.

Before calling qeBindParamInt, you must call qeSQLPrepare to prepare the SQL statement for which you are supplying parameters. You must give values to all input and input/output parameters before calling qeSQLExecute.

DTK saves the value and length pointer; they must be valid when you call qeSQLExecute. This parameter continues to point to this value until qeSetParamNull or a qeSetParam or qeBindParam function is called again for this parameter. All parameters with the same name as the one identified by *param_num* are affected.

Parameters

hstmt is the handle to the statement returned by qeSQLPrepare.

param_num is the position of the parameter to be set.

param_val points to the value of the parameter. For input parameters, *param_val* points to a buffer to hold the 2-byte integer value to be assigned to the parameter. For output parameters, *param_val* points to the buffer that holds the 2-byte integer value assigned to the parameter by the stored procedure after qeSQLExecute is called. For an input/output parameter, *param_val* plays both roles.

param_len lets you set the 2-byte integer parameter to null. For assigning a integer parameter value, set *param_len* to 0 before calling qeBindParamInt. For input parameters, if you set *param_len* to qeNULL_DATA, the parameter is set to null when you call qeSQLExecute. For output parameters, *param_len* holds the length of the parameter value after qeSQLExecute is called. Also for output parameters, *param_len* can be used to determine whether the data is NULL or truncated (qeNULL_DATA (-2) and qeTRUNCATION (-1) may be returned).

res_code is the result code returned by the function, which returns the same set of result codes as qeErr. See Appendix D, “Result and Error Message Codes,” on page 537 for a list of these result codes.

Example

```
hdbc = qeConnect ("DSN=QEDBF")   ;
hstmt = qeSQLPrepare (hdbc, "SELECT * FROM em    p
    WHERE salary = ?")  ;
num_length = 0 ;
res_code = qeBindParamInt (hstmt, 1, num_data,
    &num_length) ;
num_data = 32000  ;
res_code = qeSQLExecute (hstmt)   ;
res_code = qeEndSQL (hstmt)   ;
res_code = qeDisconnect (hdbc)   ;
```

qeBindParamLong

qeBindParamLong binds a parameter to a 4-byte integer buffer.

Syntax

```
int16  res_code qeBindParamLon g (
    int16    hstmt,
    int16    param_num,
    ptrint32 param_val,
    ptrint32 param_len)
```

Description

qeBindParamLong binds the value of a parameter in a SQL statement to a buffer that will hold the 4-byte integer value when the statement is executed.

For input and input/output parameters, you must place the value of the parameter into the buffer before executing the statement. The DTK uses this parameter value in place of the parameter itself in execution of the SQL statement or stored procedure. For stored procedure output and input/output parameters, the value assigned to the parameter during the execution of the stored procedure is placed in this buffer by the DTK.

Before calling qeBindParamLong, you must call qeSQLPrepare to prepare the SQL statement for which you are supplying parameters. You must give values to all input and input/output parameters before calling qeSQLExecute.

DTK saves the value and length pointer; they must be valid when you call qeSQLExecute. This parameter continues to point to this value until qeSetParamNull or a qeSetParam or qeBindParam function is called again for this parameter. All parameters with the same name as the one identified by *param_num* are affected.

Parameters

hstmt is the handle to the statement returned by qeSQLPrepare.

param_num is the position of the parameter to be set.

param_val points to the value of the parameter. For input parameters, *param_val* points to a buffer to hold the 4-byte integer value to be assigned to the parameter. For output parameters, *param_val* points to the buffer that holds the 4-byte integer value assigned to the parameter by the stored procedure after qeSQLExecute is called. For an input/output parameter, *param_val* plays both roles.

param_len lets you set the 4-byte integer parameter to null. For assigning a 4-byte integer parameter value, set *param_len* to 0 before calling qeBindParamLong. For input parameters, if you set *param_len* to qeNULL_DATA, the parameter is set to null when you call qeSQLExecute. For output parameters, *param_len* holds the length of the parameter value after qeSQLExecute is called. Also for output parameters, *param_len* can be used to determine whether the data is NULL or truncated (qeNULL_DATA (-2) and qeTRUNCATION (-1) may be returned).

res_code is the result code returned by the function, which returns the same set of result codes as qeErr. See Appendix D, “Result and Error Message Codes,” on page 537 for a list of these result codes.

Example

```
hdbc = qeConnect ("DSN=QEDBF")   ;
hstmt = qeSQLPrepare (hdbc, "SELECT * FROM em    p
    WHERE salary = ?")  ;
num_length = 0 ;
res_code = qeBindParamLong (hstmt, 1, num_data,
    &num_length) ;
num_data = 32000  ;
res_code = qeSQLExecute (hstmt)   ;
res_code = qeEndSQL (hstmt)   ;
res_code = qeDisconnect (hdbc)   ;
```

qeBindParamTime

qeBindParamTime binds a parameter to a time buffer.

Syntax

```
int16  res_code qeBindParamTim e (
    int16    hstmt,
    int16    param_num,
    ptrstr   param_val,
    ptrint32 param_len)
```

Description

qeBindParamTime binds the value of a parameter in a SQL statement to a buffer that will hold the 26-byte time value when the statement is executed.

For input and input/output parameters, you must place the value of the parameter into the buffer before executing the statement. The DTK uses this parameter value in place of the parameter itself in execution of the SQL statement or stored procedure. For stored procedure output and input/output parameters, the value assigned to the parameter during the execution of the stored procedure is placed in this buffer by the DTK.

Before calling qeBindParamTime you must call qeSQLPrepare to prepare the SQL statement for which you are supplying parameters. You must give values to all input and input/output parameters before calling qeSQLExecute.

DTK saves the value and length pointer; they must be valid when you call qeSQLExecute. This parameter continues to point to this value until qeSetParamNull or a qeSetParam or qeBindParam function is called again for this parameter. All parameters with the same name as the one identified by *param_num* are affected.

Parameters

hstmt is the handle to the statement returned by qeSQLPrepare.

param_num is the position of the parameter to be set.

param_val points to the value of the parameter. For input parameters, *param_val* points to a buffer to hold the 26-byte time value to be assigned to the parameter. For output parameters, *param_val* points to the buffer that holds the time value assigned to the parameter by the stored procedure after qeSQLExecute is called. For an input/output parameter, *param_val* plays both roles.

param_len is the date-time precision of the value assigned to this parameter. Set it to 19 before calling qeBindParamTime. For input parameters, if you set *param_len* to qeNULL_DATA, the parameter is set to null when you call qeSQLExecute. For output parameters, *param_len* holds the length of the parameter value after qeSQLExecute is called. Also for output parameters, *param_len* can be used to determine whether the data is NULL or truncated (qeNULL_DATA (-2) and qeTRUNCATION (-1) may be returned).

res_code is the result code returned by the function, which returns the same set of result codes as qeErr. See Appendix D, "Result and Error Message Codes," on page 537 for a list of these result codes.

Example

```
hdbc = qeConnect ("DSN=QEDBF")   ;
hstmt = qeSQLPrepare (hdbc, "SELECT * FROM em    p
    WHERE hire_date = ?")  ;
time_len = 19 ;
res_code = qeBindParamTime (hstmt, 1, hire_date,
&time_len) ;
strcpy (hire_date, "0000-00-00 03:14:12:000000")    ;
res_code = qeSQLExecute (hstmt)   ;
res_code = qeEndSQL (hstmt)   ;
res_code = qeDisconnect (hdbc)   ;
```

qeClauseGet and qeClauseGetBuf

These functions return a clause from a Select statement.

Syntax

```
ptrstr  xxx_clause qeClauseGe t (
    int16    hstmt,
    int16    which_clause)

int16 res_code qeClauseGetBu f (
    int16    hstmt,
    int16    which_clause,
    ptrstr   clause_buf)
```

Description

qeClauseGet returns a pointer to the clause string. This string is stored in a buffer maintained by DTK. You must copy the string out of this buffer before you call another DTK function, because the next function may use the same buffer.

With qeClauseGetBuf, you pass in a pointer to a buffer you have allocated. The clause string is put in the buffer. You must make sure that the buffer is large enough to hold the returned string.

Parameters

hstmt is the handle to the statement returned by qeExecSQL or qeSQLPrepare.

which_clause specifies which clause is to be returned, and is one of the following:

Constant	Value	Description
qeCLAUSE_WHERE	1	Return Where clause.
qeCLAUSE_HAVING	2	Return Having clause.
qeCLAUSE_GROUPBY	3	Return Group By clause.
qeCLAUSE_ORDERBY	4	Return Order By clause.
qeCLAUSE_COMPUTEBY	5	Return Compute By clause.

Constant	Value	Description
qeCLAUSE_FROM	6	Return From clause.
qeCLAUSE_OTHER	7	Return other, database-specific clause.

xxx_clause is the clause returned by qeClauseGet.

clause_buf is a pointer to a user-allocated buffer for the clause returned by qeClauseGetBuf.

res_code is the result code returned by qeClauseGetBuf, which returns the same set of result codes as qeErr. See Appendix D, “Result and Error Message Codes,” on page 537 for a list of these result codes.

Example

```
hdbc = qeConnect ("DSN=QEDBF")   ;
hstmt = qeSQLPrepare (hdbc, "SELECT * FROM em    p
    WHERE last_name = 'Woltman'")   ;
where_clause = qeClauseGet (hstmt, qeCLAUSE_WHERE)    ;
...
res_code = qeEndSQL (hstmt)   ;
res_code = qeDisconnect (hdbc)   ;
```

qeClearParam

qeClearParam clears the value of a parameter in a SQL statement.

Syntax

int16 ***res_code*** qeClearPara m (int16 ***hstmt***, int16 ***param_num***)

Description

qeClearParam clears the value of a parameter that was set by a qeSetParam function, or unbinds a parameter that was bound by a qeBindParam function.

Before calling qeClearParam, you must call qeSQLPrepare to prepare the SQL statement for which you are supplying parameters. You must reassign values to all cleared parameters before calling qeSQLExecute or DTK returns an error.

Parameters

hstmt is the handle to the statement returned by qeSQLPrepare.

param_num is the position of the parameter to be cleared.

res_code is the result code returned by the function, which returns the same set of result codes as qeErr. See Appendix D, “Result and Error Message Codes,” on page 537 for a list of these result codes.

Example

```
hdbc = qeConnect ("DSN=QEDBF")   ;
hstmt = qeSQLPrepare (hdbc, "SELECT * FROM em    p
    WHERE last_name = ?")  ;
char_len = 10 ;
res_code = qeBindParamChar (hstmt, 1, lname, &char_len)     ;
strcpy (lname, "Bennett")  ;
res_code = qeClearParam (hstmt, 1)    ;
/* Must set param again before executing *    /
char_len = 10 ;
res_code = qeBindParamChar (hstmt, 1, lname, &char_len)     ;
res_code = qeSQLExecute (hstmt)   ;
res_code = qeEndSQL (hstmt)   ;
res_code = qeDisconnect (hdbc)   ;
```

qeColAlias and qeColAliasBuf

These functions return the alias for the requested column.

Syntax

```
ptrstr col_alias qeColAlia s (
    int16    hstmt,
    int16    col_num)

int16 res_code qeColAliasBu f (
    int16    hstmt,
    ptrstr   col_alias,
    int16    col_num)
```

Description

qeColAlias returns a pointer to the column alias string. This string is stored in a buffer maintained by DTK. You must copy the string out of this buffer before you call another DTK function, because the next function may use the same buffer.

With qeColAliasBuf, you pass in a pointer to a buffer you have allocated. The string is put in the buffer. You must make sure that the buffer is large enough to hold the returned string.

Parameters

hstmt is the handle to the statement returned by qeExecSQL or qeSQLPrepare.

col_alias points to a buffer to hold the resulting column alias.

col_num is the column number for which an alias will be returned. The first column number is 1.

res_code is the result code returned by qeColAliasBuf, which returns the same set of result codes as qeErr. See Appendix D, "Result and Error Message Codes," on page 537 for a list of these result codes.

Example

```
hdbc = qeConnect ("DSN=QEDBF")   ;
hstmt = qeSQLPrepare (hdbc, "SELECT * FROM emp")    ;
col_alias = qeColAlias (hstmt, 2)   ;
...
res_code = qeEndSQL (hstmt)   ;
res_code = qeDisconnect (hdbc)   ;
```

qeColDateEnd

qeColDateEnd returns the offset of the end of a date-time value.

Syntax

```
int16 end_offset qeColDateEn d (int16 hstmt, int16
col_num)
```

Description

qeColDateEnd returns the offset to the last significant character of the value in a date-time column. Date-time values are 26-byte character strings formatted as

```
YYYY-MM-DD HH:MM:SS.SSSSS  S
```

This format is used for date, time, or date-time values. The end offset is the (0-origin) offset to the last significant character in the value. For example, if the column contains date values without the time, the end offset is 9, the offset to the second D (see "Date-Time Values" on page 54).

Parameters

hstmt is the handle to the statement returned by qeExecSQL or qeSQLPrepare.

col_num is the column number whose offset is to be returned. The first column number is 1.

end_offset is the returned offset.

Example

To get the ending offset of the HIRE_DATE column in the dBASE employee file:

```
hdbc = qeConnect ("DSN=QEDBF")   ;
...
hstmt = qeExecSQL (hdbc, "SELECT hire_date FROM emp")     ;
end_offset = qeColDateEnd (hstmt, 1)    ;
...
res_code = qeEndSQL (hstmt)   ;
res_code = qeDisconnect (hdbc)   ;
```

See Also

qeColDateStart.

qeColDateStart

qeColDateStart returns the offset of the start of a date-time value.

Syntax

```
int16 start_offset qeColDateStar t (int16 hstmt, int16
col_num)
```

Description

qeColDateStart returns the offset to the first significant character of the value in a date-time column. Date-time values are 26-byte character strings formatted as

```
YYYY-MM-DD HH:MM:SS.SSSSS  S
```

This format is used for date, time, or date-time values. The starting offset is the (0-origin) offset to the first significant character in the value. For example, if the column contains date values without the time, the start offset is 0, the offset to the first Y (see “Date-Time Values” on page 54).

Parameters

hstmt is the handle to the statement returned by qeExecSQL or qeSQLPrepare.

col_num is the column number whose offset is to be returned. The first column number is 1.

start_offset is the returned offset (0-origin).

Example

To get the starting offset of the HIRE_DATE column in the dBASE employee file:

```
hdbc = qeConnect ("DSN=QEDBF")   ;
...
hstmt = qeExecSQL (hdbc, "SELECT hire_date FROM emp")     ;
start_offset = qeColDateStart (hstmt, 1)    ;
...
res_code = qeEndSQL (hstmt)   ;
res_code = qeDisconnect (hdbc)   ;
```

See Also

qeColDateEnd.

qeColDBType

qeColDBType returns the database system's data type.

Syntax

```
int16 col_type qeColDBTyp e (int16 hstmt, int16 col_num)
```

Description

qeColDBType returns the underlying database system's data type for a column in a SQL Select statement.

DTK returns column values in one of eight standard data types. The column's DTK data type is returned by qeColType.

Each database system supported by DTK uses different data types. DTK maps the database system data types to one of the eight data types. In some cases you may want the underlying database system's data type in addition to the DTK data type. qeColDBType returns the database system's data type. These are listed in the database driver reference.

See Appendix E, "Compatibility Issues," on page 553 for more information.

Parameters

hstmt is the handle to the statement returned by qeExecSQL or qeSQLPrepare.

col_num is the column number whose name is to be returned. The first column number is 1.

col_type is the returned data type.

Example

To get the database system's data type of the first column in the dBASE employee file:

```
hdbc = qeConnect ("DSN=QEDBF")   ;
...
hstmt = qeExecSQL (hdbc, "SELECT * FROM emp")    ;
col_type = qeColDBType (hstmt, 1)   ;
...
res_code = qeEndSQL (hstmt)   ;
res_code = qeDisconnect (hdbc)   ;
```

qeColDBTypeName and qeColDBTypeNameBuf

These functions fill in the buffer with the database's native data type name for the requested column.

Syntax

```
ptrstr  type_name qeColDBTypeNam e (
    int16     hstmt,
    int16     col_num)

int16  res_code qeColDBTypeNameBu  f (
    int16     hstmt,
    ptrstr    type_name,
    int16     col_num)
```

Description

qeColDBTypeName returns a pointer to the data type string. This string is stored in a buffer maintained by DTK. You must copy the string out of this buffer before you call another DTK function, because the next function may use the same buffer.

With qeColDBTypeNameBuf, you pass in a pointer to a buffer you have allocated. The string is put in the buffer. You must make sure that the buffer is large enough to hold the returned string.

Parameters

hstmt is the handle to the statement returned by qeExecSQL or qeSQLPrepare.

type_name points to a buffer to hold the resulting type name.

col_num is the column number whose information is to be replaced. The first column number is 1.

res_code is the result code returned by qeColDBTypeNameBuf, which returns the same set of result codes as qeErr. See Appendix D, "Result and Error Message Codes," on page 537 for a list of these result codes.

Example

```
hdbc = qeConnect ("DSN=QEDBF")   ;
hstmt = qeSQLPrepare (hdbc, "SELECT * FROM emp")    ;
type_name = qeColDBTypeName (hstmt, 2)   ;
...
res_code = qeEndSQL (hstmt)   ;
res_code = qeDisconnect (hdbc)   ;
```

qeColExpr and qeColExprBuf

These functions return the expression for the requested column.

Syntax

```
ptrstr col_expr qeColExp r (
    int16    hstmt,
    int16    col_num)

int16 res_code qeColExprBu f (
    int16    hstmt,
    ptrstr   col_expr,
    int16    col_num)
```

Description

qeColExpr returns a pointer to the expression string. This string is stored in a buffer maintained by DTK. You must copy the string out of this buffer before you call another DTK function, because the next function may use the same buffer.

With qeColExprBuf, you pass in a pointer to a buffer you have allocated. The string is put in the buffer. You must make sure that the buffer is large enough to hold the returned string.

Parameters

hstmt is the handle to the statement returned by qeExecSQL or qeSQLPrepare.

col_expr points to a buffer to hold the resulting column expression.

col_num is the column number for which an expression will be returned. The first column number is 1.

res_code is the result code returned by qeColExprBuf, which returns the same set of result codes as qeErr. See Appendix D, "Result and Error Message Codes," on page 537 for a list of these result codes.

Example

```
hdbc = qeConnect ("DSN=QEDBF")   ;
hstmt = qeSQLPrepare (hdbc, "SELECT * FROM emp")    ;
col_expr = qeColExpr (hstmt, 2)   ;
...
res_code = qeEndSQL (hstmt)   ;
res_code = qeDisconnect (hdbc)   ;
```

qeColName and qeColNameBuf

qeColName and qeColNameBuf return the name of a column.

Syntax

```
ptrstr col_name qeColNam e (int16 hstmt, int16 col_num)
```

```
int16 res_code qeColNameBu f (
    int16    hstmt,
    ptrstr   col_name,
    int16    col_num)
```

Description

qeColName and qeColNameBuf return the name of one column in a SQL Select statement.

qeColName returns a pointer to the column name string. This string is stored in a buffer maintained by DTK. You must copy the string out of this buffer before you call another DTK function, because the next function may use the same buffer.

With qeColNameBuf, you pass in a pointer to a buffer you have allocated. The string is put in the buffer. You must make sure that the buffer is large enough to hold the returned string.

Parameters

hstmt is the handle to the statement returned by qeExecSQL or qeSQLPrepare.

col_num is the column number whose name is to be returned. The first column number is 1.

col_name is the returned column name. Column name is “” for expressions in the SQL Select statement. For example, the column name of the column in the following Select statement is “”.

```
SELECT last_name + first_name FROM em    p
```

res_code is the result code returned by qeColNameBuf, which returns the same set of result codes as qeErr. See Appendix D, “Result and Error Message Codes,” on page 537 for a list of these result codes.

Example

To get the column name of the first column in the dBASE employee file:

```
hdbc = qeConnect ("DSN=QEDBF")   ;
...
hstmt = qeExecSQL (hdbc, "SELECT * FROM emp")    ;
col_name = qeColName (hstmt, 1)   ;
...
res_code = qeEndSQL (hstmt)   ;
res_code = qeDisconnect (hdbc)   ;
```

qeColPrecision

qeColPrecision returns the number of digits in a decimal column.

Syntax

```
int16 precision qeColPrecisio n (int16 hstmt, int16
col_num)
```

Description

qeColPrecision returns the number of digits in a decimal column.

Decimal columns (type 3) are defined by the total number of digits in their values (precision), and the number of digits right of the decimal point (scale).

For example, precision=8, scale=2 means that the values have 8 digits total, 2 to the right of the decimal point and 6 to the left of the decimal point.

Parameters

hstmt is the handle to the statement returned by qeExecSQL or qeSQLPrepare.

col_num is the number of the column for which you want a precision value returned. The first column number is 1. If this column is not a decimal column, the function returns an error.

precision is the returned number of digits for the column.

Example

To get the precision of the SALARY column in the dBASE employee file:

```
hdbc =  qeConnec t ("DSN=QEDBF") ;
...
hstmt = qeExecSQL (hdbc, "SELECT salary FROM emp")     ;
precision = qeColPrecision (hstmt, 1)    ;
...
res_code = qeEndSQL (hstmt)   ;
res_code = qeDisconnect (hdbc)   ;
```

See Also

qeColScale.

qeColScale

qeColScale returns the number of digits to the right of the decimal point in a decimal column.

Syntax

```
int16 scale qeColScal e (int16 hstmt, int16 col_num)
```

Description

qeColScale returns the number of digits to the right of the decimal point in a decimal column.

Decimal columns (type 3) are defined by the total number of digits in their values (precision), and the number of digits right of the decimal point (scale).

For example, precision=8, scale=2 means that the values have 8 digits total, 2 to the right of the decimal point and 6 to the left of the decimal point.

Parameters

hstmt is the handle to the statement returned by qeExecSQL or qeSQLPrepare.

col_num is the number of the column for which you want a scale value returned. The first column number is 1. If this column is not a decimal column, the function returns an error.

scale is the returned number of digits right of the decimal point for the column.

Example

To get the scale of the SALARY column in the dBASE employee file:

```
hdbc = qeConnect ("DSN=QEDBF")   ;
...
hstmt = qeExecSQL (hdbc, "SELECT salary FROM emp")     ;
scale = qeColScale (hstmt, 1)   ;
...
res_code = qeEndSQL (hstmt)   ;
res_code = qeDisconnect (hdbc)   ;
```

See Also

qeColPrecision.

qeColType

qeColType returns the data type for a column in a SQL Select statement.

Syntax

`int16 col_type qeColType (int16 hstmt, int16 col_num)`

Parameters

hstmt is the handle to the statement returned by qeExecSQL or qeSQLPrepare.

col_num is the number of the column for which a data type is to be returned. The first column number is 1.

col_type is the returned data type. This is the data type used in DTK for the database values, as follows:

Constant	Value	Description
qeCHAR	1	Fixed length character string
qeVARCHAR	2	Variable length character string
qeDECIMAL	3	Decimal number (BCD)
qeINTEGER	4	Long integer (4-byte)
qeSMALLINT	5	Integer (2-byte)
qeFLOAT	6	Floating-point number (4-byte)
qeDOUBLEPRECISION	7	Double-precision floating-point number (8-byte)
qeDATETIME	8	Date-time (26-byte character string)

Example

To get the column type of the first column in the dBASE employee file:

```
hdbc = qeConnect ("DSN=QEDBF")   ;
...
hstmt = qeExecSQL (hdbc, "SELECT * FROM emp")    ;
col_type = qeColType (hstmt, 1)   ;
...
res_code = qeEndSQL (hstmt)   ;
res_code = qeDisconnect (hdbc)   ;
```

Notes

When you retrieve column values using the qeVal functions, you do not have to use the qeVal function that matches the data type returned by qeColType. The qeVal functions automatically convert the value to the desired data type. For example, if qeColType returns 3, meaning a decimal number, you can retrieve the values using qeValDouble to get the value as a double-precision floating-point number, or qeValChar to get the value as a character string.

See “Data Types in DTK” on page 53 for more information.

See Also

qeVal functions.

qeColTypeAttr

qeColTypeAttr returns whether a column has a specified attribute.

Syntax

```
int16 result qeColTypeAtt r (
    int16    hstmt,
    int16    col_num,
    int16    attribute)
```

Parameters

hstmt is the handle to the statement returned by qeExecSQL or qeSQLPrepare.

col_num is the number of the column for which a data type is to be returned. The first column number is 1.

attribute is the specific attribute for which you are checking. You must specify one of the following attributes:

Attribute	Value	Description	
qeATTRIBUTE_ UPDATABLE	1	Reports whether a column is updatable. Possible *result* values are	
		qeCOL_READ_ONLY The column cannot be updated.	0
		qeCOL_WRITEABLE The column can be updated.	1
		qeCOL_UNKNOWN The function cannot report whether the column is searchable.	100
qeATTRIBUTE_ UNSIGNED	4	Returns whether the column is signed or unsigned. Possible *result* values are	
		qeCOL_SIGNED The column is signed.	0
		qeCOL_UNSIGNED The column is unsigned.	1

Attribute	Value	Description
qeATTRIBUTE_ MONEY	5	Returns whether the column is of type Money. Possible *result* values are qeCOL_NOT_MONEY 0 The column is not of type Money. qeCOL_MONEY 1 The column is of type Money.
qeATTRIBUTE_ AUTO_INCRE	6	Returns whether the column is automatically incremented on update or insert. qeCOL_NOT_AUTO_INCRE 0 The column is not automatically incremented. qeCOL_AUTO_INCRE 1 The column is automatically incremented.
qeATTRIBUTE_N ULLABLE	2	Returns whether the column is nullable. Possible *result* values are qeCOL_NOT_NULLABLE 0 The column cannot be null. qeCOL_NULLABLE 1 The column can be null. qeCOL_UNKNOWN 100 The function cannot report whether the column is nullable.

Attribute	Value	Description
qeATTRIBUTE_ SEARCHABLE	3	Reports whether a column can be used in a SQL Where clause to search for specific records. Possible *result* values are qeCOL_UNSEARCHABLE 0 The column cannot appear in the Where clause. qeCOL_LIKE_ONLY 1 The column can appear in the Where clause only when used with the LIKE operator. qeCOL_ALL_EXCEPT_LIKE 2 The column can appear in the Where clause except with the LIKE operator. qeCOL_SEARCHABLE 3 The column can appear anywhere within the Where clause. qeCOL_UNKNOWN 100 The function cannot report whether the column is searchable.

result contains a constant returned by the function that reports the status of the specified attribute in column *col_num*. See the description of *attribute* for possible values.

Example

```
hdbc = qeConnect ("DSN=QEDBF")   ;
hstmt = qeExecSQL (hdbc, "SELECT * FROM emp")    ;
nullable = qeColTypeAttr (hstmt, 1,
qeATTRIBUTE_NULLABLE)  ;
res_code = qeEndSQL (hstmt)   ;
res_code = qeDisconnect (hdbc)   ;
```

qeColumns

qeColumns returns information on the set of column definitions for a table.

Syntax

```
int16 hstmt qeColumn s (int16 hdbc, ptrstr table_name)
```

Description

qeColumns creates a statement execution (*hstmt*) that returns information on the set of column definitions for a table. qeColumns returns one record per column. Each record contains the following columns:

Column	Type	Description
Table Qualifier	Char(128)	Table qualifier.
Table User	Char(128)	Table user.
Table Name	Char(128)	Table name.
Column	Char(128)	Column name.
Type	Int16	Data type (DTK types).
Width	Int32	Width in bytes.
DB Type	Int16	Database data type.
DB Type Name	Char(128)	Data source-dependent data type name.
Attr1	Int16	Precision for decimal types, date start position for dates, null otherwise.
Attr2	Int16	Scale for decimal types, date end position for dates, null otherwise.
Nullable	Int16	Whether column can be null. Values: qeCOL_NULLABLE, qeCOL_NOT_NULLABLE, qeCOL_UNKNOWN.
Remarks	Char(256)	Comments (if available).

You retrieve this information like you would other database values—using the qeVal, qeBindCol, and qeFetch functions.

Parameters

hdbc is a handle to a database connection obtained from qeConnect.

table_name is the table whose columns are to be returned.

hstmt is the handle to the statement returned by qeColumns.

Example

```
hdbc = qeConnect ("DSN=QEDBF")   ;
hstmt = qeColumns (hdbc, "emp.dbf")   ;
while (qeFetchNext (hstmt) == qeSUCCESS)     {
    ...
/* Get info about columns. *   /
    ...
}
res_code = qeEndSQL (hstmt)   ;
res_code = qeDisconnect (hdbc)   ;
```

qeColWidth

qeColWidth returns the width of a column.

Syntax

```
int32 col_width qeColWidt h (int16 hstmt, int16 col_num)
```

Description

qeColWidth returns the column width of one column in a SQL Select statement. The column width is the size, in bytes, of the longest value that may be stored in this column.

Parameters

hstmt is the handle to the statement returned by qeExecSQL or qeSQLPrepare.

col_num is the column number whose width is to be returned. The first column number is 1.

col_width is the returned column width.

Example

To get the column width of the first column in the dBASE employee file:

```
hdbc = qeConnect ("DSN=QEDBF")   ;
...
hstmt = qeExecSQL (hdbc, "SELECT * FROM emp")    ;
col_width = qeColWidth (hstmt, 1)   ;
...
res_code = qeEndSQL (hstmt)   ;
res_code = qeDisconnect (hdbc)   ;
```

Notes

For character and date-time types, qeColWidth returns the maximum number of characters for a column including the zero terminator byte. Therefore the returned width is 1 greater than the maximum value length. If you are defining a variable or allocating a buffer to hold these values, you must take into account the zero terminator byte that is added by the qeValChar, qeValCharBuf, or qeBindCol functions.

When you use qeValChar to retrieve values whose type is Integer, Long Integer, Float, Double Float, Decimal, or Date-Time, you must consider that qeColWidth returns the width of the stored values, not the number of characters returned by qeValChar. The number of characters returned by qeValChar is determined by the format string you use.

See “Data Types in DTK” on page 53 and “Format Strings” on page 59, as well as Appendix E, “Compatibility Issues,” on page 553 for more information.

See Also

qeColType.

qeCommit

qeCommit ends a database transaction and commits all changes to the database made during the transaction.

Syntax

```
int16 res_code qeCommi t (int16 hdbc)
```

Description

qeCommit commits all changes that have been made using Insert, Update, or Delete statements on the connection since qeBeginTran was called. You must call qeBeginTran to start a transaction before you can call qeCommit to save all changes.

qeCommit also frees all locks that have been held in the database system.

Parameters

hdbc is the handle to the database connection returned by qeConnect.

res_code is the result code returned by qeCommit, which returns the same set of result codes as qeErr. See Appendix D, "Result and Error Message Codes," on page 537 for a list of these result codes.

Example

To commit changes made to a SQL Server database:

```
hdbc = qeConnect ("DSN=QESS;UID=sa;SRVR=PION1")    ;
...
res_code = qeBeginTran (hdbc)   ;
hstmt = qeExecSQL (hdbc   ,
    "UPDATE emp SET salary=salary*1.1")   ;
res_code = qeEndSQL (hstmt)   ;
res_code = qeCommit (hdbc)   ;
res_code = qeDisconnect (hdbc)   ;
```

See Also

qeBeginTran, qeRollback.

qeConnect

qeConnect opens a connection to a database system to allow SQL statements to be executed.

Syntax

`int16` ***`hdbc`*** `qeConnec t (ptrstr` ***`con_string`***`)`

Description

qeConnect opens a connection to a database system to allow SQL statements to be executed.

You can have several connections open simultaneously to different database systems, or simultaneous connections to the same database system, if supported by the database system you are using. Refer to the INTERSOLV database driver reference for more information on specific database systems.

Parameters

con_string is a connection string identifying the database system and any additional logon information. The connection string has the form:

"DSN=*data source name*[;*attribute*=*value*[;*attribute*=*value*]...]"

The attributes required by each database system vary. See the INTERSOLV database driver reference for the attributes supported by specific databases. DTK recognizes the following attributes for all database systems:

Attribute	Description
DSN	The name of the data source defined in the ODBC.INI file.
DLG	When DLG=1, displays a logon dialog box that allows user input of connection string information. When DLG=2, displays a logon dialog box only when the connection string supplied via qeConnect is insufficient to log on to the data source.

Attribute	Description
DRV	For compatibility with QELIB 1.0, this value is used if a data source name (DSN) is not present in the connection string. DTK changes it to the data source name.
UID	The user ID or name.
PWD	The password.
MODIFYSQL	Used by DTK to ensure compatibility between the SQL used in the application and the SQL used in the database system. When set to 1 (the default), the database driver expects ODBC-compliant syntax, which it will modify as necessary for the underlying database system. When set to 0, the database driver expects and supports the native syntax of the underlying database system. This enables you to continue using applications developed with the SQL supported by the QELIB 1.0 database drivers.
ALLOWLOCKS	If enabled (set to 1), ensures that the isolation level chosen via qeSetIsolationLevel will support locking. May reduce performance. SQLBase is the only database system currently affected by this option.
REREADAFTERUPDATE	If enabled (set to 1), DTK rereads a record from the database after updating it. This is useful for getting the correct value of auto-updated columns such as timestamps.
REREADAFTERINSERT	If enabled (set to 1), DTK rereads a record from the database after inserting it. This is useful for getting the correct value of auto-updated columns such as timestamps.

hdbc is the returned handle to the database connection. This identifies the connection and is a parameter to other functions. If the *hdbc* is 0, the connection could not be opened.

Example

To connect to dBASE files:

```
hdbc = qeConnect ("DSN=QEDBF")   ;
...
res_code = qeDisconnect (hdbc)   ;
```

To connect to SQL Server:

```
hdbc = qeConnect ("DSN=QESS;SRVR=PION1;UID=sa;PWD=magic"     )
...
res_code = qeDisconnect (hdbc)   ;
```

See Also

qeDisconnect.

qeDatabases

qeDatabases returns information on the set of databases that can be accessed from a connection.

Syntax

```
int16 hstmt qeDatabase s (int16 hdbc)
```

Description

qeDatabases creates a statement execution handle (*hstmt*) that returns information on the set of databases that can be accessed by a specific database connection.

qeDatabases returns one record per database. Each record contains the following columns:

Column	Type	Description
Database	Char(128)	A database name.
Remarks	Char(256)	Comments (if available).

You retrieve this information like you would other database values—using the qeVal, qeBindCol, and qeFetch functions.

Note: If you call this function when connected to a flat-file database such as Btrieve, dBASE, Paradox, Excel, or text files, it does not return a result.

Parameters

hdbc is a handle to a database connection obtained from qeConnect.

hstmt is the handle to the statement returned by qeDatabases.

Example

```
hdbc = qeConnect ("DSN=QEORA;DLG=1")   ;
hstmt = qeDatabases (hdbc)  ;
while (qeFetchNext (hstmt) == qeSUCCESS)     {
    ...get info about databases..   .
}
res_code = qeDisconnect (hdbc)   ;
```

qeDataLen

qeDataLen returns the length of a value retrieved by a qeVal function.

Syntax

```
int32 len qeDataLe n (int16 hstmt)
```

Description

qeDataLen returns the length from the previous call to a qeVal function.

Parameters

hstmt is the handle to the statement returned by qeExecSQL or qeSQLPrepare.

len is the returned column value length in bytes. If the column value was null, qeNULL_DATA (-2) is returned. If the column value was truncated, qeTRUNCATION (-1) is returned.

Example

To get the first column's value and its length for each record in the dBASE employee file:

```
hdbc = qeConnect ("DSN=QEDBF")   ;
...
hstmt = qeExecSQL (hdbc, "SELECT * FROM emp")    ;
while (qeFetchNext (hstmt) == 0)    {
    value = qeValChar (hstmt,1,"",0)   ;
    val_len = qeDataLen (hstmt)   ;
...
}
res_code = qeEndSQL (hstmt)   ;
res_code = qeDisconnect (hdbc)   ;
```

Notes

If your database system can store null values, you should follow every call to a qeVal function with a call to qeDataLen to determine if the value is null. See "Null Values" on page 58 for more information. If the call to qeDataLen follows a call to qeValChar or qeValCharBuf, a return value of qeTRUNCATION (-1) means that the entire column value was not returned. This occurs if a non-zero *max_len* was specified on the qeVal function and the length of the column value is greater than *max_len*, or if a zero *max_len* was specified and the length of the column value is greater than 1000 characters. See qeValChar and qeValCharBuf for more information.

See Also qeVal functions.

qeDBErr

qeDBErr returns the database error resulting from the last DTK function.

Syntax

```
int32 db_code qeDBErr ()
```

Description

qeDBErr returns the underlying database system's error code resulting from the last DTK function you called.

The purpose of this function is to allow you to get the error numbers generated by database systems such as Oracle or SQL Server.

If a database system detects an error, qeErr returns a number indicating that an error occurred. If qeErr returns qeDBSYS_ERROR (4), meaning that the error was reported by the database system, then you can call qeDBErr to get a database system error number. Use the database system error number when you consult the database system's documentation. You can also call qeErrMsg to get the underlying database system error message text.

qeDBErr is not a substitute for qeErr. First call qeErr to determine if the function succeeded. If qeErr returns qeDBSYS_ERROR (4), then you can call qeDBErr to determine if a database error number is associated with the error and call qeErrMsg to get the error message text.

Parameters

db_code is the returned error number from the underlying database system. If 0, no database system error was reported.

When qeErr returns qeDBSYS_ERROR (4), qeDBErr may return 0. This result means that the underlying database system does not have a separate error code.

Example

To execute a Select statement on a dBASE file, checking for errors after each function call:

```
hdbc = qeConnect ("DSN=QEDBF")   ;
if (qeErr () == 0)   {
    hstmt = qeExecSQL (hdbc, "SELECT * FROM emp")    ;
    if (qeErr () == 0)   {
             ...
             res_code = qeEndSQL (hstmt)   ;
    }
    else if (qeErr () == qeDBSYS_ERROR    )
             db_err = qeDBErr ()  ;
    res_code = qeDisconnect (hdbc)   ;
}
else
    db_err = qeDBErr (  )
```

See Also

qeErr, qeErrMsg and qeErrMsgBuf, qeDBErr.

qeDisconnect

qeDisconnect closes a database connection.

Syntax

```
int16 res_code qeDisconnec t (int16 hdbc)
```

Description

qeDisconnect closes a connection to a database system. You should close all connections before your program terminates to free system resources.

Parameters

hdbc is the handle to the database connection returned by qeConnect.

res_code is the result code returned by qeDisconnect, which returns the same set of result codes as qeErr. See Appendix D, “Result and Error Message Codes,” on page 537 for a list of these result codes.

Example

```
hdbc = qeConnect ("DSN=QEDBF")   ;
...
res_code = qeDisconnect (hdbc)   ;
```

See Also

qeConnect.

qeEndSQL

qeEndSQL ends the execution of a SQL statement.

Syntax

```
int16 res_code qeEndSQL (int16 hstmt)
```

Description

qeEndSQL ends the execution of a SQL statement. It is important to call qeEndSQL to free system resources.

Note that qeDisconnect closes all statements for the connection.

Parameters

hstmt is the handle to the statement returned by qeExecSQL or qeSQLPrepare.

res_code is the result code returned by qeEndSQL, which returns the same set of result codes as qeErr. See Appendix D, “Result and Error Message Codes,” on page 537 for a list of these result codes.

Example

To execute a select statement on a dBASE file:

```
hdbc = qeConnect ("DSN=QEDBF")   ;
...
hstmt = qeExecSQL (hdbc, "SELECT * FROM emp")    ;
...
res_code = qeEndSQL (hstmt)   ;
res_code = qeDisconnect (hdbc)   ;
```

See Also

qeExecSQL.

qeErr

qeErr returns the result code of the last DTK function.

Syntax

```
int16 res_code qeErr ()
```

Description

qeErr returns the result code of the last DTK function you called.

You should call qeErr immediately after calling any other DTK function that does not return a result code (for example, a qeVal function). You should determine whether any errors have occurred before using the results of a function or before calling other DTK functions.

Parameters

res_code is the returned result or error code. If res_code is qeSUCCESS (0), the last DTK function called completed without error. If res_code contains a 4- or 5-digit error code, you can call qeErrMsg to get the DTK error message text. If the result code is qeDBSYS_ERROR (4), then a call to qeErrMsg returns the underlying database system error message text. When the result code is qeDBSYS_ERROR (4), you can also call qeDBErr to get the underlying database system error code.

The following table lists the result codes returned by qeErr.

Constant	Value	Description
qeLOCK_NO_REC	-6	A lock was attempted, but either no record was selected by the primary key, the record has been deleted by another user, or another user has changed the value of a key field.
qeEOF	-5	EOF. Returned by qeFetchNext, qeFetchPrev, or qeFetchRandom when there is no record to return.
qeUSER_CANCELED	-4	User canceled out of the logon dialog box
qeOUT_OF_MEMORY	-3	Windows or OS/2 is out of memory. This is usually fatal.

Constant	Value	Description
qeSUCCESS	0	Success.
qeSUCCESS_WITH_INFO	1	Success with information (warning).
qeNO_DATA_WITH_INFO	2	EOF with additional information (usually ESC during a fetch).
qeDBSYS_ERROR	4	Database system error. Call qeDBErr to retrieve the database system's error number.
qeLIBSYS_ERROR	5	Returned when the system cannot locate the DTK Dynamic Link Library.

See Appendix D, "Result and Error Message Codes," on page 537 for a list of the 4- or 5-digit error codes returned by qeErr and their corresponding messages.

Example

To execute a select statement on a dBASE file, checking for errors after each function call:

```
hdbc = qeConnect ("DSN=QEDBF")   ;
if (qeErr () == qeSUCCESS)    {
    hstmt = qeExecSQL (hdbc, "SELECT * FROM emp")    ;
    if (qeErr () == qeSUCCESS)    {
             ...
             res_code = qeEndSQL (hstmt)   ;
    }
    res_code = qeDisconnect (hdbc)   ;
}
```

See Also

qeErrMsg and qeErrMsgBuf, qeDBErr.

qeErrMsg and qeErrMsgBuf

These functions return the text associated with the error or warning generated by the last DTK function you called.

Syntax

```
ptrstr  err_msg qeErrMsg ( )

int16  res_code qeErrMsgBuf (ptrstr   err_msg)
```

Description

qeErrMsg and qeErrMsgBuf return the text associated with the error or warning generated by the last DTK function you called. These functions are usually called after you have called **qeErr** to determine if there is an error message.

Because this function returns a pointer, it has two forms (see “Parameter Conventions” on page 151).

When you use qeErrMsg, the function returns a pointer to the string. The string is stored in a buffer maintained by DTK. Copy the string out of this buffer before you call another DTK function, because the next function may use the same buffer.

When you use qeErrMsgBuf, you pass in a pointer to a buffer you have allocated. The string is put in the buffer. Make sure the buffer is large enough to hold the returned string.

Parameters

err_msg is the returned error or warning message text. Error messages may contain up to 512 characters. It is important that the variable you pass as the parameter is declared large enough to hold 512 characters. *err_msg* can also contain multiple errors or warnings of under 512 bytes.

res_code is the result code returned by qeErrMsgBuf, which returns the same set of result codes as qeErr. See Appendix D, “Result and Error Message Codes,” on page 537 for a list of these result codes.

Example

To execute a Select statement on a dBASE file, checking for errors after each function call, and getting the message text if an error occurs:

```
hdbc = qeConnect ("DSN=QEDBF")   ;
if (qeErr () == 0)   {
    hstmt = qeExecSQL (hdbc, "SELECT * FROM emp")    ;
    if (qeErr () == 0)   {
              ...
              res_code = qeEndSQL (hstmt)   ;
    }
    else
              msg = qeErrMsg ()  ;
              res_code = qeDisconnect (hdbc)   ;
}
else
    msg = qeErrMsg ()  ;
```

See Also

qeErr, qeDBErr.

qeExecSQL

qeExecSQL executes a SQL statement.

Syntax

```
int16 hstmt qeExecSQL (int16  hdbc, ptrstr  sql_stmt)
```

Description

qeExecSQL executes a SQL statement. The SQL statement may be a Select, Insert, Update, or Delete statement, or any other valid statement for the database system.

Parameters

hdbc is the handle to the database connection returned by qeConnect.

sql_stmt is the SQL statement to be executed. If *sql_stmt* is a zero-length string (the empty string, “”), DTK executes the SQL statement sent using the qeSetSQL and qeAppendSQL functions.

hstmt is the returned handle to the statement execution. This identifies the statement and is a parameter to other functions. If *hstmt* is 0, the statement could not be executed.

Example

To execute a Select statement on a dBASE file:

```
hdbc = qeConnect ("DSN=QEDBF")   ;
...
hstmt = qeExecSQL (hdbc, "SELECT * FROM emp")    ;
...
res_code = qeEndSQL (hstmt)   ;
res_code = qeDisconnect (hdbc)   ;
```

Notes

For Select statements, use the qeCol functions to get information about the columns in the statement, the qeFetch functions to retrieve records, and the qeVal functions to retrieve column values.

For all statements, call qeEndSQL to terminate execution of the statement.

See Also

qeAppendSQL, qeEndSQL, qeSetSQL, qeCol functions, qeFetchNext, qeFetchPrev, qeFetchRandom, qeSetSelectOptions, qeNumCols, and the qeVal functions.

qeFetchLogClose

qeFetchLogClose closes the log files used by DTK's fetching functions.

Syntax

```
int16 res_code qeFetchLogClose (int16  hstmt)
```

Description

qeFetchLogClose closes the temporary log files used to support the qeFetchPrev, qeFetchRandom, and qeFetchNumRecs functions. Temporary files are created only if qeSetSelectOptions has been called with options that require them.

See qeSetSelectOptions for more information.

qeFetchLogClose does not delete the temporary log files. DTK automatically reopens the files when you call qeFetchNext, qeFetchPrev, qeFetchRandom, or qeFetchNumRecs.

Parameters

hstmt is the handle to the statement returned by qeExecSQL or qeSQLPrepare.

res_code is the result code returned by qeFetchLogClose, which returns the same set of result codes as qeErr. See Appendix D, "Result and Error Message Codes," on page 537 for a list of these result codes.

Example

To close the temporary log files while fetching records from the employee database file:

```
hdbc=qeConnect ("DSN=QESS;UID=sa;SRVR=PION1")    ;
res_code = qeSetSelectOptions (hdbc, qeLOG_ALWAYS)    ;
hstmt = qeExecSQL (hdbc, "SELECT * FROM emp")    ;
while (qeFetchNext (hstmt) == 0)    {
    ...
    res_code = qeFetchLogClose (hstmt)   ;
}
res_code = qeEndSQL (hstmt)   ;
res_code = qeDisconnect (hdbc)   ;
```

See Also

qeSetSelectOptions.

qeFetchNext

qeFetchNext retrieves the next record from the database.

Syntax

```
int16 res_code qeFetchNext (int16  hstmt)
```

Description

qeFetchNext retrieves the next record from a database system. If this is the first call to qeFetchNext following qeExecSQL, this function retrieves the first record. The retrieved record becomes the current record.

If a qeBindCol function was not called before qeFetchNext, this function gets a record from the database system and stores it in DTK's current record buffer. The record is not returned to your application. To get the column values from the current record, use the qeVal functions.

If a qeBindCol function was called before qeFetchNext, this function gets a record from the database system and puts the column values into the variables specified by the qeBindCol function.

If qeSetAutoUpdate has been called to enable auto-updating, and changes have been made to the current record via calls to qePut functions, a call to qeFetchNext updates the current record.

A result of qeEOF (-5) is returned if an attempt is made to read past the last record returned by the Select statement.

Parameters

hstmt is the handle to the statement returned by qeExecSQL or qeSQLPrepare.

res_code is the result code returned by qeFetchNext, which returns the same set of result codes as qeErr. See Appendix D, "Result and Error Message Codes," on page 537 for a list of these result codes.

Example

To fetch all records from the employee database file:

```
hdbc = qeConnect ("DSN=QEDBF")   ;
...
hstmt = qeExecSQL (hdbc, "SELECT * FROM emp")    ;
while (qeFetchNext (hstmt) == 0)    {
...
}
res_code = qeEndSQL (hstmt)   ;
res_code = qeDisconnect (hdbc)   ;
```

See Also

qeVal functions, qePut functions, qeBindCol functions, qeFetchPrev, qeFetchRandom, qeSetSelectOptions.

qeFetchNumRecs

qeFetchNumRecs returns the number of records chosen by the Select statement.

Syntax

```
int32 num_recs qeFetchNumRecs (int16   hstmt)
```

Description

qeFetchNumRecs returns the number of records chosen by the Select statement. This function can be used only if qeSetSelectOptions has been called to enable it.

To determine the number of records selected, DTK fetches all rows from the result set. If you have not enabled backward fetching, calling qeFetchNumRecs causes an error to be returned. If you have selected a large number of records, this function may work slowly, and may create large temporary log files.

Parameters

hstmt is the handle to the statement returned by qeExecSQL or qeSQLPrepare.

num_recs is the number of records selected by the SQL statement.

Example

To get the number of records in the employee database file:

```
hdbc = qeConnect ("DSN=QEDBF")   ;
res_code = qeSetSelectOptions (hdbc, qeFETCH_ANY_DIR)     ;
hstmt = qeExecSQL (hdbc, "SELECT * FROM emp")    ;
num_recs = qeFetchNumRecs (hstmt)   ;
res_code = qeEndSQL (hstmt)   ;
res_code = qeDisconnect (hdbc)   ;
```

See Also

qeSetSelectOptions.

qeFetchPrev

qeFetchPrev retrieves the previous record from the database.

Syntax

`int16 res_code qeFetchPrev (int16 hstmt)`

Description

qeFetchPrev retrieves the previous record from a database system. The retrieved record becomes the current record. This function can be used only if qeSetSelectOptions has been called to enable it.

If a qeBindCol function was not called before fetching records, this function gets a record from the database system and stores it in DTK's current record buffer. The record is not returned to your application. To get the column values from the current record, use the qeVal functions.

If a qeBindCol function was called, this function gets a record from the database system and puts the column values into the variables specified by the qeBindCol function.

If qeSetAutoUpdate has been called to enable auto-updating, and changes have been made to the current record via calls to qePut functions, a call to qeFetchPrev updates the current record.

When qeFetchPrev attempts to fetch a record before the first record returned by the Select statement, it returns a result of qeEOF (-5).

Parameters

hstmt is the handle to the statement returned by qeExecSQL or qeSQLPrepare.

res_code is the result code returned by qeFetchPrev, which returns the same set of result codes as qeErr. See Appendix D, "Result and Error Message Codes," on page 537 for a list of these result codes.

Example

To fetch a record that has already been read from the employee database file:

```
hdbc = qeConnect ("DSN=QEDBF")   ;
res_code = qeSetSelectOptions (qeFETCH_ANY_DIR)    ;
hstmt = qeExecSQL (hdbc, "SELECT * FROM emp")    ;
res_code = qeFetchNext (hstmt)   ;
/* This is repeated to read other records *    /
...
res_code = qeFetchPrev (hstmt)   ;
...
res_code = qeEndSQL (hstmt)   ;
res_code = qeDisconnect (hdbc)   ;
```

See Also

qeVal functions, qePut functions, qeBindCol functions, qeFetchNext, qeFetchRandom, qeSetSelectOptions.

qeFetchRandom

qeFetchRandom retrieves a designated record from the database.

Syntax

```
int16 res_code qeFetchRandom (int16  hstmt, int32 rec_num)
```

Description

qeFetchRandom retrieves a designated record from a database system, which becomes the current record. This function returns EOF if the designated record is not in the result set.

This function can be used only if qeSetSelectOptions has been called to enable it.

If a qeBindCol function was not called before fetching records, this function gets a record from the database system and stores it in DTK's current record buffer. The record is not returned to your application. To get the column values from the current record, use the qeVal functions.

If a qeBindCol function was called, this function gets a record from the database system and puts the column values into the variables specified by the qeBindCol function.

If qeSetAutoUpdate has been called to enable auto-updating, and changes have been made to the current record via calls to qePut functions, a call to qeFetchRandom updates the current record.

Parameters

hstmt is the handle to the statement returned by qeExecSQL or qeSQLPrepare.

rec_num is the record number to be read. The first record is 1.

res_code is the result code returned by qeFetchRandom, which returns the same set of result codes as qeErr. See Appendix D, “Result and Error Message Codes,” on page 537 for a list of these result codes.

Example

To fetch the last record from the employee database file:

```
hdbc = qeConnect ("DSN=QEDBF")   ;
res_code = qeSetSelectOptions (qeFETCH_ANY_DIR)    ;
hstmt = qeExecSQL (hdbc, "SELECT * FROM emp")    ;
num_recs = qeFetchNumRecs (hstmt)   ;
res_code = qeFetchRandom (hstmt, num_recs)    ;
...
res_code = qeEndSQL (hstmt)   ;
res_code = qeDisconnect (hdbc)   ;
```

See Also

qeVal functions, qePut functions, qeBindCol functions, qeFetchNext, qeFetchPrev, qeSetSelectOptions.

qeForeignKeys

qeForeignKeys creates a statement execution (*hstmt*) that returns information on the set of columns that compose a table's foreign keys.

Syntax

```
int16  hstmt = qeForeignKeys   (
    int16    hdbc,
    ptrstr   pk_table_name,
    ptrstr   fk_table_name)
```

Description

qeForeignKeys returns one record per column in the primary key. Each record contains the following columns:

Column	Type	Description
PK Table Qualifier	Char(128)	Primary key table qualifier. May be NULL
PK Table User	Char(128)	Primary key table user. May be NULL
PK Table Name	Char(128)	Primary key table name.
PK Column Name	Char(128)	Primary key column name.
FK Table Qualifier	Char(128)	Foreign key table qualifier. May be NULL
FK Table User	Char(128)	Foreign key table user. May be NULL
FK Table Name	Char(128)	Foreign key table name.
FK Column Name	Char(128)	Foreign key column name.
Sequence No	Int16	Column sequence number, which is the number of this column within the foreign key. For example, for the foreign key LAST_NAME, FIRST_NAME, the Sequence No would be 1 in the row returned for LAST_NAME and 2 in the row returned for FIRST_NAME.

Column	Type	Description
Update Action	Int16	Action applied to the foreign key when an UPDATE is performed. Values: 0 = qeCascade 1 = qeRestrict 2 = qeSetNull
Delete Action	Int16	Action applied to the foreign key when a DELETE is performed. Values: 0 = qeCascade 1 = qeRestrict 2 = qeSetNull
FK Index Name	Char(128)	Foreign key name. NULL if not applicable to the data source.
PK Index Name	Char(128)	Primary key name. NULL if not applicable to the data source.

Not all database systems support foreign keys. You should include error-checking code to handle those database systems that do not.

Parameters

hstmt is the handle to the statement returned by qeForeignKeys.

hdbc is the handle to a database connection obtained from qeConnect.

pk_table_name is the table whose primary keys are to be returned.

fk_table_name is the table whose foreign keys are to be returned.

Example

```
hdbc = qeConnect ("DSN=QESS;DLG=1")    ;
hstmt = qeForeignKeys (hdbc, "DEPT", "EMP");
while (qeFetchNext (hstmt) == qeSUCCESS)     {
    /* Get info about Foreign Keys *   /
}
```

See Also

qePrimaryKeys.

qeGetAutoUpdate

qeGetAutoUpdate returns the auto-update setting.

Syntax

```
int16 option qeGetAutoUpdate (int16  hdbc)
```

Parameters

option reports whether DTK automatically generates Update or Insert statements when you move off a changed or inserted row. It has one of the following values:

Constant	Value	Action
qeAUTOUPD_DISCARD	1	DTK discards changes or insertions. This is the default.
qeAUTOUPD_DEFER	2	DTK saves the changes but does not update the database. This option enables you to use the qeApplyAll and qeUndoAll functions.
qeAUTOUPD_UPDATE	3	DTK updates the changed or inserted record.

hdbc is the handle to the database connection returned by qeConnect.

Example

```
hdbc = qeConnect ("DSN=QEDBF")   ;
AutoUpdate = qeGetAutoUpdate (hdbc)   ;
/* Value will be default of qeAUTOUPD_DISCARD *    /
res_code = qeDisconnect (hdbc)   ;
```

See Also

qeSetAutoUpdate.

qeGetIsolationLevel

qeGetIsolationLevel returns the isolation level for the connection.

Syntax

```
int16 level qeGetIsolationLevel(int16 hdbc)
```

Parameters

level is the isolation level currently set in the database. It has one of the following values:

Constant	Value	Description
qeISO_READ_ UNCOMMITTED	0x0001	Read uncommitted (0) isolation level. Locks are obtained on modifications to the database and held until end of transaction (EOT). Reading from the database does not involve any locking.
qeISO_READ_ COMMITTED	0x0002	Read committed (1) isolation level. Locks are acquired for reading and modifying the database. Locks are released after reading but locks on modified objects are held until EOT.
qeISO_REPEATABLE_R EAD	0x0004	Repeatable read (2) isolation level. Locks are obtained for reading and modifying the database. Locks on all modified objects are held until EOT. Locks obtained for reading data are held until EOT. Locks on non-modified access structures (indexes, hashing structures, etc.) are released after reading.

Constant	Value	Description
qeISO_SERIALIZABLE	0x0008	Serializable (3) isolation level. All data read or modified is locked until EOT. All access structures that are modified are locked until EOT. Access structures used by the query are locked until EOT.
qeISO_VERSIONING	0x0010	Versioning (4) isolation level. Similar to isolation level 3, serializable, but provides greater concurrence through the use of non-locking “record versioning” protocols.

hdbc is the handle to the database connection returned by qeConnect.

Example

```
hdbc = qeConnect ("DSN=QESS")   ;
levels = qeGetSupportedIsolationLevels (hdbc)    ;
cur_level = qeGetIsolationLevel (hdbc)    ;
if (levels & qeISO_READ_COMMITTED    )
    res_code = qeSetIsolationLevel (hdbc,
             qeISO_READ_COMMITTED)  ;
res_code = qeDisconnect (hdbc)   ;
```

See Also

qeSetIsolationLevel.

qeGetLockOptions

qeGetLockOptions returns the current lock options.

Syntax

```
int16 option qeGetLockOptions (int16   hdbc)
```

Parameters

option is the current lock options setting. It has one of the following values:

Constant	Value	Description
qeLOCK_NO_OPTIONS	0	Default; DTK neither compares nor refreshes the record in the log file.
qeLOCK_COMPARE	1	When locking, DTK compares the record in the log file to the corresponding record in the database, and raises a warning if they are different.
qeLOCK_REFRESH	2	When locking, DTK automatically refreshes the record in the log file with new column values.

hdbc is the handle to the database connection returned by qeConnect.

Example

```
hdbc = qeConnect ("DSN=QEDBF")   ;
lock_options = qeGetLockOptions (hdbc)    ;
/* This will return 0 (the default, *    /
/* no lock options set. *   /
res_code = qeDisconnect (hdbc)   ;
```

See Also

qeSetLockOptions.

qeGetLoginTimeout

qeGetLoginTimeout returns the login timeout.

Syntax

```
int32 timeout qeGetLoginTimeout ( )
```

Description

qeGetLoginTimeout returns the login timeout, in seconds.

Parameters

timeout is the login timeout set in the last call to qeSetLoginTimeout. If the login timeout has not been set, qeGetLoginTimeout returns the default.

Example

```
timeout = qeGetLoginTimeout ()   ;
/* Default is 0, which indicates to wait indefinitely *     /
```

See Also

qeSetLoginTimeout.

qeGetMaxRows

qeGetMaxRows returns either the current value of the maximum number of rows that should be returned for the query as specified in the last call to qeSetMaxRows, or it returns the default of 0.

Syntax

```
int32 max_rows qeGetMaxRows (int16   hdbc)
```

Parameters

max_rows is the maximum number of rows that should be returned for the query as specified in the last call to qeSetMaxRows. If qeSetMaxRows has not been called, the default of 0 is returned, indicating that all rows are to be returned.

hdbc is the handle to the database connection returned by qeConnect.

Example

```
/* Return all rows from the query. *    /
hdbc = qeConnect ("DSN=QEDBF")   ;
max_rows = qeGetMaxRows ( hdbc )   ;
/* Returns 0, indicating no limit on rows returned. *     /
res_code = qeDisconnect (hdbc)   ;
```

See Also

qeSetMaxRows.

qeGetODBCHdbc

qeGetODBCHdbc returns the ODBC *hdbc* that corresponds to the DTK *hdbc*.

Important: This function is potentially dangerous. Using the ODBC *hdbc* to change the state of the ODBC connection may create situations that trap. Use at your own risk.

Syntax

```
int32 ODBCHdbc qeGetODBCHdbc (int16   hdbc)
```

Parameters

hdbc is the handle to the database connection returned by qeConnect.

ODBCHdbc is the handle used as a pointer to information about the ODBC connection.

Example

```
hdbc = qeConnect ("DSN=QESS")   ;
odbc_hdbc = qeGetODBCHdbc (hdbc)   ;
...
/* Use odbc_hdbc in direct calls to ODBC functions. *     /
...
res_code = qeDisconnect (hdbc)   ;
```

qeGetODBCHenv

qeGetODBCHenv returns the ODBC environment handle associated with the instance of DTK.

Syntax

```
int32 ODBCHenv qeGetODBCHenv ( )
```

Description

qeGetODBCHenv returns the ODBC environment handle associated with the instance of DTK.

The *ODBCHenv* is a handle to the implied environment that is created between calls to qeLibInit and qeLibTerm. When you call qeLibInit, DTK closes any currently allocated *ODBCHenv* and opens a new one. A call to qeLibTerm closes this *ODBCHenv*. Therefore, the current *ODBCHenv* becomes invalid when you call either qeLibInit or qeLibTerm.

Important: This function is potentially dangerous. Using the ODBC *hdbc* to change the state of the ODBC connection may create situations that trap. Use at your own risk.

Parameters

ODBCHenv is an environment handle used as a pointer to information about the ODBC environment.

Example

```
res_code = qeLibInit ()  ;
odbc_henv = qeGetODBCHenv ()   ;
...
/* Use odbc_henv in direct calls to ODBC functions. *     /
...
res_code = qeLibTerm ()  ;
```

qeGetODBCHstmt

qeGetODBCHstmt returns the ODBC *hstmt* that corresponds to the DTK *hstmt*.

Important: This function is potentially dangerous. Using the ODBC *hdbc* to change the state of the ODBC connection may create situations that trap. Use at your own risk.

Syntax

```
int32 ODBCHstmt qeGetODBCHstmt (int16   hstmt)
```

Parameters

hstmt is the handle to the statement returned by qeExecSQL or qeSQLPrepare.

ODBCHstmt is the handle used as a pointer to information about the ODBC statement.

Example

```
hdbc = qeConnect ("DSN=QEDBF")   ;
hstmt = qeExecSQL (hdbc, "SELECT * FROM dept")    ;
odbc_hstmt = qeGetODBCHstmt (hstmt)   ;
...
/* Use odbc_hstmt in direct calls to ODBC functions. *     /
...
res_code = qeDisconnect (hdbc)   ;
```

qeGetODBCInfoChar and qeGetODBCInfoCharBuf

These functions return information about an ODBC connection.

Syntax

```
ptrstr  char_val qeGetODBCInfoChar (
    int16    hdbc,
    int16    option)

int16  res_code qeGetODBCInfoCharBuf (
    int16    hdbc,
    int16    option,
    ptrstr   char_val)
```

Description

qeGetODBCInfoChar returns a pointer to the information string. This string is stored in a buffer maintained by DTK. You must copy the string out of this buffer before you call another DTK function, because the next function may use the same buffer.

With qeGetODBCInfoCharBuf, you pass in a pointer to a buffer you have allocated. The information string is put in the buffer. You must make sure that the buffer is large enough to hold the returned string.

Parameters

hdbc is a connection returned from qeConnect.

option is either one of the following constants, or one of the constants defined by ODBC for use with the ODBC SQLGetInfo function that returns a character string.

Constant	Value	Description
qeINFO_DRIVER_VER	7	A character string specifying the version, and optionally a description, of the driver. The form is *aa.bb.cccc*, where *aa* is the major version, *bb* is the minor version, and *cccc* is the release version.
qeINFO_SEARCH_PATTERN_ESCAPE	14	A character string specifying the escape character the driver supports for the pattern-matching characters underscore (_) and percent (%).
qeINFO_DATA_SOURCE_READ_ONLY	25	A character string: Y if the data source is read only; otherwise N.
qeINFO_EXPRESSIONS_IN_ORDERBY	27	A character string: Y if the data source supports ORDER BY expressions; N if not.
qeINFO_IDENTIFIER_QUOTE_CHAR	29	The character string that surrounds a delimited identifier; blank if none.
qeINFO_OUTER_JOINS	38	A character string: Y if the data source supports outer joins and the driver supports the ODBC outer join request syntax; N if not.
qeINFO_OWNER_TERM	39	A character string that contains the data source vendor's name for an owner; for example, "owner" or "Authorization ID."
qeINFO_PROCEDURE_TERM	40	A character string that contains the data source vendor's name for a procedure; for example, "database procedure" or "stored procedure."

Constant	Value	Description
qeINFO_QUALIFIER_ NAME_SEPARATOR	41	A character string that contains the separators the data source uses between the qualifier name and the qualified name element.
qeINFO_TABLE_TERM	45	A character string that contains the data source vendor’s name for a table; for example, “table” or “file.”
qeINFO_QUALIFIER_ TERM	42	A character string that contains the data source vendor’s name for a qualifier; for example, “database” or “directory.”

char_val is a pointer to a string that is the connection information.

res_code is the result code returned by qeGetInfoCharBuf, which returns the same set of result codes as qeErr. See Appendix D, “Result and Error Message Codes,” on page 537 for a list of these result codes.

Example

```
hdbc = qeConnect ("DSN=QESS")   ;
version = qeGetODBCInfoChar (hdbc, qeINFO_DRIVER_VER)     ;
res_code = qeDisconnect (hdbc)   ;
```

qeGetODBCInfoLong

qeGetODBCInfoLong returns information about an ODBC connection.

Syntax

```
int32 long_val qeGetODBCInfoLong (int16  hdbc, int16
option)
```

Parameters

hdbc is a connection returned from qeConnect.

option is one of the following constants, or any other constant defined by ODBC for use with the ODBC SQLGetInfo function that returns a 4-byte integer.

Constant	Value	Description
qeINFO_ACTIVE_ CONNECTIONS	0	An integer specifying the number of active *hdbc*s that the driver can support. Zero indicates no specified limit or the limit is unknown.
qeINFO_ACTIVE_ STATEMENTS	1	An integer specifying the number of active *hstmt*s that the driver can support for an *hdbc*. Zero indicates no specified limit or the limit is unknown.
qeINFO_IDENTIFIER_ CASE	28	An integer indicating the forms of names: 1 =must be uppercase 2 = must be lowercase 3 = case sensitive; can contain upper and lowercase 4 =not case sensitive
qeINFO_MAX_ COLUMN_NAME_LEN	30	An integer specifying the maximum length of a column name.
qeINFO_MAX_ CURSOR_NAME_LEN	31	An integer specifying the maximum length of a cursor name.

Constant	Value	Description
qeINFO_MAX_ OWNER_NAME_LEN	32	An integer specifying the maximum length of an owner name.
qeINFO_MAX_ PROCEDURE_NAME_LEN	33	An integer specifying the maximum length of a procedure name. Zero means procedures are not supported.
qeINFO_MAX_ QUALIFIER_NAME_LEN	34	An integer specifying the maximum length of a qualifier name. Zero means qualifier names are not supported.
qeINFO_MAX_TABLE_ NAME_LEN	35	An integer specifying the maximum length of a table name.
qeINFO_CONVERT_ FUNCTIONS	48	A mask enumerating the scalar conversion functions supported by the driver and data source. The mask qeSQL_FN_CVT_CONVERT determines which conversion functions are supported.

Constant	Value	Description
qeINFO_NUMERIC_ FUNCTIONS	49	A mask enumerating the numeric functions supported by the driver and data source. The following masks are used: qeSQL_FN_NUM_ABS qeSQL_FN_NUM_ACOS qeSQL_FN_NUM_ASIN qeSQL_FN_NUM_ATAN qeSQL_FN_NUM_ATAN2 qeSQL_FN_NUM_CEILING qeSQL_FN_NUM_COS qeSQL_FN_NUM_COT qeSQL_FN_NUM_EXP qeSQL_FN_NUM_FLOOR qeSQL_FN_NUM_LOG qeSQL_FN_NUM_MOD qeSQL_FN_NUM_RAND qeSQL_FN_NUM_PI qeSQL_FN_NUM_SIGN qeSQL_FN_NUM_SIN qeSQL_FN_NUM_SQRT qeSQL_FN_NUM_TAN

Constant	Value	Description
qeINFO_STRING_ FUNCTIONS	50	A mask enumerating the scalar string functions supported by the driver and data source. The following masks are used: qeSQL_FN_STR_ASCII qeSQL_FN_STR_CHAR qeSQL_FN_STR_CONCAT qeSQL_FN_STR_INSERT qeSQL_FN_STR_LEFT qeSQL_FN_STR_LTRIM qeSQL_FN_STR_LENGTH qeSQL_FN_STR_LOCATE qeSQL_FN_STR_LCASE qeSQL_FN_STR_REPEAT qeSQL_FN_STR_REPLACE qeSQL_FN_STR_RIGHT qeSQL_FN_STR_RTRIM qeSQL_FN_STR_SUBSTRING qeSQL_FN_STR_UCASE
qeINFO_SYSTEM_ FUNCTIONS	51	At mask enumerating the scalar string functions supported by the driver and data source. The following masks are used: qeSQL_FN_SYS_USERNAME qeSQL_FN_SYS_DBNAME qeFN_SYS_IFNULL

Constant	Value	Description
qeINFO_TIMEDATE_ FUNCTIONS	52	A mask enumerating the scalar date and time functions supported by the driver and data source. The following masks are used: qeSQL_FN_TD_NOW qeSQL_FN_TD_CURDATE qeSQL_FN_TD_DAYOFMONTH qeSQL_FN_TD_DAYOFWEEK qeSQL_FN_TD_DAYOFYEAR qeSQL_FN_TD_MONTH qeSQL_FN_TD_QUARTER qeSQL_FN_TD_WEEK qeSQL_FN_TD_YEAR qeSQL_FN_TD_CURTIME qeSQL_FN_TD_HOUR qeSQL_FN_TD_MINUTE qeSQL_FN_TD_SECOND

Example

```
hdbc = qeConnect ("DSN=QESS")   ;
num_connects = qeGetODBCInfoLong (hdbc,
    qeINFO_ACTIVE_CONNECTIONS)  ;
res_code = qeDisconnect (hdbc)   ;
```

qeGetOneHstmtPerHdbcOptions

qeGetOneHstmtPerHdbcOptions returns the settings that determine which fetching commands and statement behaviors are allowed by DTK when connected to statements that support only one statement per connection. For more information, see Appendix C, “Coding for Single Statement Database Systems,” on page 529.

Syntax

```
int16 flags qeGetOneHstmtPerHdbcOption   s (int16 hdbc)
```

Parameters

flags is a set of option flags that specifies the read-ahead activity, statement routing, and *hstmt* behavior in effect when DTK uses multiple connections to databases that support only one statement per connection. It returns one read-ahead, routing, and *hstmt* option from among the following:

Constant	Value	Description
qeREADAHEAD_AT_ EXEC	0x0001	DTK reads the statement's entire result set into the log file when the statement executes.
qeREADAHEAD_AT_ UPDATE	0x0002	DTK reads the remainder of the result set into the log file whenever a record is locked, updated, or deleted. This is the default read-ahead option.
qeREADAHEAD_ COMMIT_UPDATES	0x0003	DTK avoids all read-ahead activity by requiring you to commit all updates before fetching any more records.
qeROUTING_READ	0x0008	DTK routes this statement through a connection used for read-only statements.
qeROUTING_UPDATE	0x0010	DTK routes this statement through a connection used for statements that modify the database.

Constant	**Value**	**Description**
qeROUTING_DEFAULT	0x0018	This option lets DTK decide which connection to send the statement to. This is the default routing option.
qeHSTMT_LOCAL	0x0020	Tells DTK that this *hstmt* cannot affect any other active *hstmt* in the same application.
qeHSTMT_NONLOCAL	0x0040	Tells DTK that this *hstmt* may affect other *hstmt*s in the same application. This is the default *hstmt* behavior.

hdbc is the handle to the database connection returned by qeConnect.

Example

```
hdbc = qeConnect ("DSN=QESS")   ;
options = qeGetOneHstmtPerHdbcOptions (hdbc)    ;
res_code = qeDisconnect (hdbc)   ;
```

qeGetParamBinary and qeGetParamBinaryBuf

These functions are used with stored procedures and return an output or input/output parameter's value as a binary value.

Syntax

```
ptrstr  bin_val qeGetParamBinary   (
    int16    hstmt,
    int16    param_num,
    ptrint32 max_len)

int16  res_code qeGetParamBinaryBuf   (
    int16    hstmt,
    ptrstr   bin_val,
    int16    param_num,
    ptrint32 max_len)
```

Description

qeGetParamBinary and qeGetParamBinaryBuf return the value of a stored procedure's output or input/output parameter as a binary value. If the parameter's data type is not binary, the value is converted to binary.

The qeGetParamBinary function returns a pointer to the binary value, which is stored in a buffer maintained by DTK. Copy the value out of this buffer before you call another DTK function, because the next function may use the same buffer.

The qeGetParamBinaryBuf function passes a pointer to a buffer you have allocated, and the value is put in the buffer. Make sure the buffer is large enough to hold the returned value.

If the parameter's data type is a character string (type 1 or 2) or a binary value (type 101 or 102), you may specify the maximum length of data to be returned.

Not all database systems support stored procedures, and some that support stored procedures do not support output parameters. You should include error-checking code to handle those database systems that do not support output and input/output parameters in stored procedures.

Parameters

hstmt is the handle to the statement returned by qeSQLPrepare.

param_num is the position of the parameter value to be returned.

max_len is the maximum number of characters returned if the parameter's data type is character string (type 1 or 2) or binary (type 101 or 102). If *max_len* is zero, the entire string is returned (up to 1000 characters). If *max_len* is not zero and the parameter's data type is not 1, 2, 101, or 102, an error is returned.

max_len is typically used either because your macro language limits character strings to a size that is less than the size of the values of the parameters, or because the parameter values are very large and you want to retrieve only part of the value.

If the value of the parameter is too large to retrieve with one call to qeGetParamBinary, you can call qeGetParamBinary again and again on the same parameter to retrieve more of the value.

bin_val is the returned binary value.

res_code is the result code returned by qeGetParamBinaryBuf, which returns the same set of result codes as qeErr. See Appendix D, "Result and Error Message Codes," on page 537 for a list of these result codes.

Example

```
hdbc = qeConnect ("DSN=QEORA;DLG=2")    ;
hstmt = qeSQLPrepare (hdbc, "{call getPicture (?)}")     ;
res_code = qeSetParamIOType (hstmt, 1, qePARAM_OUTPUT)     ;
res_code = qeSetParamDataType (hstmt, 1, qeBINARY, 100,
0);
res_code = qeSQLExecute (hstmt)   ;
binValue = qeGetParamBinary (hstmt, 1)    ;
...
res_code = qeEndSQL (hstmt)   ;
res_code = qeDisconnect (hdbc)   ;
```

qeGetParamBit

qeGetParamBit is used with stored procedures and returns an output or input/output parameter's value as a bit in a 2-byte integer.

Syntax

```
int16 bit_val qeGetParamBit  (
    int16    hstmt,
    int16    param_num
```

Description

qeGetParamBit returns the parameter's value as a bit in a 2-byte integer. If the parameter's data type is not bit (type 110), the value is converted to this data type.

Not all database systems support stored procedures, and some that support stored procedures do not support output parameters. You should include error-checking code to handle those database systems that do not support output and input/output parameters in stored procedures.

Parameters

hstmt is the handle to the statement returned by qeSQLPrepare.

param_num is the position of the parameter value to be returned.

bit_val is the returned bit value.

Example

```
hdbc = qeConnect ("DSN=QEORA;DLG=2")   ;
hstmt = qeSQLPrepare (hdbc, "{call IsExempt (?)}")    ;
res_code = qeSetParamIOType (hstmt, 1, qePARAM_OUTPUT)    ;
res_code = qeSetParamDataType (hstmt, 1, qeBIT, 0, 0)    ;
res_code = qeSQLExecute (hstmt)   ;
bitValue = qeGetParamBit (hstmt, 1)   ;
...
res_code = qeEndSQL (hstmt)   ;
res_code = qeDisconnect (hdbc)   ;
```

qeGetParamChar and qeGetParamCharBuf

These functions are used with stored procedures and return a character string containing the value from an output or input/output parameter.

Syntax

```
ptrstr char_val qeGetParamChar (
    int16    hstmt,
    int16    param_num,
    ptrstr   fmt_string,
    ptrint32 max_len)

int16 res_code qeGetParamCharBuf (
    int16    hstmt,
    ptrstr   char_val,
    int16    param_num,
    ptrstr   fmt_string,
    ptrint32 max_len)
```

Description

qeGetParamChar and qeGetParamCharBuf return the value of a stored procedure's output or input/output parameter as a character string. If the parameter's data type is not character string, the value is converted to a character string.

The qeGetParamChar function returns a pointer to the string, which is stored in a buffer maintained by DTK. Copy the string out of this buffer before you call another DTK function, because the next function may use the same buffer.

The qeGetParamCharBuf function passes a pointer to a buffer you have allocated, and the string is put in the buffer. Make sure the buffer is large enough to hold the returned string.

Format number and date values by providing a format string (see "Format Strings" on page 59).

If the parameter's data type is a character string (type 1 or 2), you may specify the maximum length of data to be returned.

Not all database systems support stored procedures, and some that support stored procedures do not support output parameters. You should include error-checking code to handle those database systems that do not support output and input/output parameters in stored procedures.

Parameters

hstmt is the handle to the statement returned by qeSQLPrepare.

param_num is the position of the parameter value to be returned.

fmt_string is the format string.

max_len is the maximum number of characters returned if the parameter's data type is character string (type 1 or 2) or binary (type 101 or 102). If *max_len* is zero, the entire string is returned (up to 1000 characters). If *max_len* is not zero and the parameter's data type is not 1, 2, 101, or 102, an error is returned.

max_len is typically used either because your macro language limits character strings to a size that is less than the size of the values of the parameters, or because the parameter values are very large and you want to retrieve only part of the value.

If the value of the parameter is too large to retrieve with one call to qeGetParamChar, you can call qeGetParamChar again and again on the same parameter to retrieve more of the value.

char_val is the returned character value.

res_code is the result code returned by qeGetParamCharBuf, which returns the same set of result codes as qeErr. See Appendix D, "Result and Error Message Codes," on page 537 for a list of these result codes.

Example

```
hdbc = qeConnect ("DSN=QEORA;DLG=2")    ;
hstmt = qeSQLPrepare (hdbc, "{call GetDeptName (?)}")     ;
res_code = qeSetParamIOType (hstmt, 1, qePARAM_OUTPUT)     ;
res_code = qeSetParamDataType (hstmt, 1, qeCHAR, 10, 0)     ;
res_code = qeSQLExecute (hstmt)   ;
charValue = qeGetParamChar (hstmt, 1)    ;
...
res_code = qeEndSQL (hstmt)   ;
res_code = qeDisconnect (hdbc)   ;
```

See Also

qeValChar and qeValCharBuf.

qeGetParamDate and qeGetParamDateBuf

These functions are used with stored procedures and return an output or input/output parameter's value as a date value.

Syntax

```
ptrstr  date_val qeGetParamDate   (
    int16    hstmt,
    int16    param_num

int16  res_code qeGetParamDateBuf   (
    int16    hstmt,
    ptrstr   date_val,
    int16    param_num
```

Description

qeGetParamDate and qeGetParamDateBuf return the value of a stored procedure's output or input/output parameter as a date value. If the parameter's data type is not date, the value is converted to date.

The qeGetParamDate function returns a pointer to the date value, which is stored in a buffer maintained by DTK. Copy the value out of this buffer before you call another DTK function, because the next function may use the same buffer.

The qeGetParamDateBuf function passes a pointer to a buffer you have allocated, and the value is put in the buffer. Make sure the buffer is large enough to hold the returned value.

Not all database systems support stored procedures, and some that support stored procedures do not support output parameters. You should include error-checking code to handle those database systems that do not support output and input/output parameters in stored procedures.

Parameters

hstmt is the handle to the statement returned by qeSQLPrepare.

param_num is the position of the parameter value to be returned.

date_val is the returned date value.

res_code is the result code returned by qeGetParamDateBuf, which returns the same set of result codes as qeErr. See Appendix D, "Result and Error Message Codes," on page 537 for a list of these result codes.

Example

```
hdbc = qeConnect ("DSN=QEORA;DLG=2")    ;
hstmt = qeSQLPrepare (hdbc, "{call LastHireDate (?)}")     ;
res_code = qeSetParamIOType (hstmt, 1, qePARAM_OUTPUT)     ;
res_code = qeSetParamDataType (hstmt, 1, qeDATE, 0, 0)     ;
res_code = qeSQLExecute (hstmt)   ;
dateValue = qeGetParamDate (hstmt, 1)    ;
...
res_code = qeEndSQL (hstmt)   ;
res_code = qeDisconnect (hdbc)   ;
```

qeGetParamDateTime and qeGetParamDateTimeBuf

These functions are used with stored procedures and return an output or input/output parameter's value as a date-time value.

Syntax

```
ptrstr datetime_val qeGetParamDateTime   (
    int16    hstmt,
    int16    param_num

int16 res_code qeGetParamDateTimeBuf   (
    int16    hstmt,
    ptrstr   datetime_val,
    int16    param_num
```

Description

qeGetParamDateTime and qeGetParamDateTimeBuf return the value of a stored procedure's output or input/output parameter as a date-time value. If the parameter's data type is not date-time, the value is converted to date-time.

The qeGetParamDateTime function returns a pointer to the date-time value, which is stored in a buffer maintained by DTK. Copy the value out of this buffer before you call another DTK function, because the next function may use the same buffer.

The qeGetParamDateTimeBuf function passes a pointer to a buffer you have allocated, and the value is put in the buffer. Make sure the buffer is large enough to hold the returned value.

Not all database systems support stored procedures, and some that support stored procedures do not support output parameters. You should include error-checking code to handle those database systems that do not support output and input/output parameters in stored procedures.

Parameters

hstmt is the handle to the statement returned by qeSQLPrepare.

param_num is the position of the parameter value to be returned.

datetime_val is the returned date-time value.

res_code is the result code returned by qeGetParamDateBuf, which returns the same set of result codes as qeErr. See Appendix D, “Result and Error Message Codes,” on page 537 for a list of these result codes.

Example

```
hdbc = qeConnect ("DSN=QEORA;DLG=2")    ;
hstmt = qeSQLPrepare (hdbc, "{call LastHireDate (?)}")     ;
res_code = qeSetParamIOType (hstmt, 1, qePARAM_OUTPUT)     ;
res_code = qeSetParamDataType (hstmt, 1, qeDATETIME, 26,
0);
res_code = qeSQLExecute (hstmt)   ;
datetimeValue = qeGetParamDateTime (hstmt, 1)    ;
...
res_code = qeEndSQL (hstmt)   ;
res_code = qeDisconnect (hdbc)   ;
```

qeGetParamDecimal and qeGetParamDecimalBuf

These functions are used with stored procedures and return an output or input/output parameter's value as a decimal value.

Syntax

```
ptrstr  dec_val qeGetParamDecimal   (
    int16    hstmt,
    int16    param_num,
    int16    precision,
    int16    scale

int16  res_code qeGetParamDecimalBuf   (
    int16    hstmt,
    ptrstr   dec_val,
    int16    param_num,
    int16    precision,
    int16    scale
```

Description

The qeGetParamDecimal function returns a pointer to the decimal value, which is stored in a buffer maintained by DTK. Copy the value out of this buffer before you call another DTK function, because the next function may use the same buffer.

The qeGetParamDecimalBuf passes a pointer to a buffer you have allocated, and the value is put in the buffer. Make sure the buffer is large enough to hold the returned value.

qeGetParamDecimal and qeGetParamDecimalBuf return the parameter value as a decimal number. If the parameter's data type is not a decimal number, the value is converted to a decimal number (type 3).

If the parameter's data type is a character string (type 1 or 2) and the parameter's value is not a number, 0 is returned.

Not all database systems support stored procedures, and some that support stored procedures do not support output parameters. You should include error-checking code to handle those database systems that do not support output and input/output parameters in stored procedures.

Parameters

hstmt is the handle to the statement returned by qeSQLPrepare.

param_num is the number of the parameter whose value is to be returned.

precision is the total number of digits to be returned in the decimal value.

scale is the number of digits right of the decimal point to be returned in the decimal value.

dec_val is the returned decimal value.

res_code is the result code returned by qeGetParamDecimalBuf, which returns the same set of result codes as qeErr. See Appendix D, “Result and Error Message Codes,” on page 537 for a list of these result codes.

Example

```
hdbc = qeConnect ("DSN=QEORA;DLG=2")    ;
hstmt = qeSQLPrepare (hdbc, "{call TotEmpSalary (?)}}")     ;
res_code = qeSetParamIOType (hstmt, 1, qePARAM_OUTPUT)     ;
res_code = qeSetParamDataType (hstmt, 1, qeDECIMAL, 10,
2);
res_code = qeSQLExecute (hstmt)   ;
decValue = qeGetParamDecimal (hstmt, 1)    ;
...
res_code = qeEndSQL (hstmt)   ;
res_code = qeDisconnect (hdbc)   ;
```

See Also

qeValDecimal and qeValDecimalBuf.

qeGetParamDouble

qeGetParamDouble is used with stored procedures and returns an output or input/output parameter's value as a double-precision floating point number.

Syntax

```
float64  param_val qeGetParamDouble   (
    int16    hstmt,
    int16    param_num
```

Description

qeGetParamDouble returns the parameter's value as a double-precision floating-point number. If the parameter's data type is not double-precision floating-point (type 7), the value is converted to this data type.

If the parameter's data type is a character string (type 1 or 2) and the parameter's value is not a number, 0 is returned.

Not all database systems support stored procedures, and some that support stored procedures do not support output parameters. You should include error-checking code to handle those database systems that do not support output and input/output parameters in stored procedures.

Parameters

hstmt is the handle to the statement returned by qeSQLPrepare.

param_num is the number of the parameter whose value is to be returned.

param_val is the returned value.

Example

```
hdbc = qeConnect ("DSN=QEORA;DLG=2")    ;
hstmt = qeSQLPrepare (hdbc, "{call TotEmpSalary (?)}")     ;
res_code = qeSetParamIOType (hstmt, 1, qePARAM_OUTPUT)     ;
res_code = qeSetParamDataType   (
                    hstmt, 1, qeDOUBLEPRECISION, 0, 0)    ;
res_code = qeSQLExecute (hstmt)   ;
dblValue = qeGetParamDouble (hstmt, 1)    ;
...
res_code = qeEndSQL (hstmt)   ;
res_code = qeDisconnect (hdbc)   ;
```

See Also

qeValDouble.

qeGetParamFloat

qeGetParamFloat is used with stored procedures and returns an output or input/output parameter's value as a single-precision floating point number.

Syntax

```
float32  param_val qeGetParamFloat   (
    int16    hstmt,
    int16    param_num
```

Description

qeGetParamFloat returns the parameter's value as a floating-point number. If the parameter's data type is not floating-point (type 6), the value is converted to this data type.

If the parameter's data type is character string (type 1 or 2) and the parameter's value is not a number, 0 is returned.

Not all database systems support stored procedures, and some that support stored procedures do not support output parameters. You should include error-checking code to handle those database systems that do not support output and input/output parameters in stored procedures.

Parameters

hstmt is the handle to the statement returned by qeSQLPrepare.

param_num is the number of the parameter whose value is to be returned.

param_val is the returned value.

Example

```
hdbc = qeConnect ("DSN=QEORA;DLG=2")    ;
hstmt = qeSQLPrepare (hdbc, "{call TotEmpSalary (?)}")     ;
res_code = qeSetParamIOType (hstmt, 1, qePARAM_OUTPUT)     ;
res_code = qeSetParamDataType (hstmt, 1, qeFLOAT, 0, 0)     ;
res_code = qeSQLExecute (hstmt)   ;
floatValue = qeGetParamFloat (hstmt, 1)    ;
...
res_code = qeEndSQL (hstmt)   ;
res_code = qeDisconnect (hdbc)   ;
```

See Also qeValFloat.

qeGetParamInt

qeGetParamInt is used with stored procedures and returns an output or input/output parameter's value as a 2-byte integer.

Syntax

```
int16 param_val qeGetParamInt  (
    int16    hstmt,
    int16    param_num
```

Description

qeGetParamInt returns the parameter's value as a 2-byte integer. If the parameter's data type is not 2-byte integer (type 5), the value is converted to this data type.

If the parameter's data type is character string (type 1 or 2) and the parameter's value is not a number, 0 is returned.

Not all database systems support stored procedures, and some that support stored procedures do not support output parameters. You should include error-checking code to handle those database systems that do not support output and input/output parameters in stored procedures.

Parameters

hstmt is the handle to the statement returned by qeSQLPrepare.

param_num is the number of the parameter whose value is to be returned.

param_val is the returned value.

Example

```
hdbc = qeConnect ("DSN=QEORA;DLG=2")    ;
hstmt = qeSQLPrepare (hdbc, "{call TotNumEmp (?)}")     ;
res_code = qeSetParamIOType (hstmt, 1, qePARAM_OUTPUT)     ;
res_code = qeSetParamDataType (hstmt, 1, qeINTEGER, 0,0)     ;
res_code = qeSQLExecute (hstmt)   ;
intValue = qeGetParamInteger (hstmt, 1)    ;
...
res_code = qeEndSQL (hstmt)   ;
res_code = qeDisconnect (hdbc)   ;
```

See Also

qeValInt.

qeGetParamLong

qeGetParamLong is used with stored procedures and returns an output or input/output parameter's value as a 4-byte integer.

Syntax

```
int32 param_val qeGetParamLong (
    int16    hstmt,
    int16    param_num
```

Description

qeGetParamLong returns the parameter's value as a 4-byte integer. If the parameter's data type is not a 4-byte integer (type 4), the value is converted to this data type.

If the parameter's data type is character string (type 1 or 2) and the parameter's value is not a number, 0 is returned.

Not all database systems support stored procedures, and some that support stored procedures do not support output parameters. You should include error-checking code to handle those database systems that do not support output and input/output parameters in stored procedures.

Parameters

hstmt is the handle to the statement returned by qeSQLPrepare.

param_num is the number of the parameter whose value is to be returned.

param_val is the returned value.

Example

```
hdbc = qeConnect ("DSN=QEORA;DLG=2")    ;
hstmt = qeSQLPrepare (hdbc, "{call TotNumEmp (?)}")     ;
res_code = qeSetParamIOType (hstmt, 1, qePARAM_OUTPUT)     ;
res_code = qeSetParamDataType (hstmt, 1, qeLONG, 0, 0)     ;
res_code = qeSQLExecute (hstmt)   ;
longValue = qeGetParamLong (hstmt, 1)    ;
...
res_code = qeEndSQL (hstmt)   ;
res_code = qeDisconnect (hdbc);hdbc = qeConnect
("DSN=QEORA;DLG=2")  ;
```

See Also

qeValLong.

qeGetParamTime and qeGetParamTimeBuf

These functions are used with stored procedures and return an output or input/output parameter's value as a time value.

Syntax

```
ptrstr time_val qeGetParamTime (
    int16   hstmt,
    int16   param_num

int16 res_code qeGetParamTimeBuf (
    int16   hstmt,
    ptrstr  time_val,
    int16   param_num
```

Description

qeGetParamTime and qeGetParamTimeBuf return the value of a stored procedure's output or input/output parameter as a time value. If the parameter's data type is not time, the value is converted to time.

The qeGetParamTime function returns a pointer to the time value, which is stored in a buffer maintained by DTK. Copy the value out of this buffer before you call another DTK function, because the next function may use the same buffer.

The qeGetParamTimeBuf function passes a pointer to a buffer you have allocated, and the value is put in the buffer. Make sure the buffer is large enough to hold the returned value.

Not all database systems support stored procedures, and some that support stored procedures do not support output parameters. You should include error-checking code to handle those database systems that do not support output and input/output parameters in stored procedures.

Parameters

hstmt is the handle to the statement returned by qeSQLPrepare.

param_num is the position of the parameter value to be returned.

time_val is the returned time value.

res_code is the result code returned by qeGetParamDateBuf, which returns the same set of result codes as qeErr. See Appendix D, "Result and Error Message Codes," on page 537 for a list of these result codes.

Example

```
hstmt = qeSQLPrepare (hdbc, "{call GetStartTime (?) }")     ;
res_code = qeSetParamIOType (hstmt, 1, qePARAM_OUTPUT)     ;
res_code = qeSetParamDataType (hstmt, 1, qeTIME, 0, 0)     ;
res_code = qeSQLExecute (hstmt)   ;
timeValue = qeGetParamTime (hstmt, 1)    ;
...
res_code = qeEndSQL (hstmt)   ;
res_code = qeDisconnect (hdbc)   ;
```

qeGetQueryTimeout

qeGetQueryTimeout returns the query timeout.

Syntax

```
int32 timeout qeGetQueryTimeout (int16    hdbc)
```

Parameters

timeout is the query timeout set in the last call to qeSetQueryTimeout. If a query timeout has not been set, the default of 0 (wait indefinitely) is returned.

hdbc is the handle to the database connection returned by qeConnect.

Example

```
hdbc = qeConnect ("DSN=QESS")   ;
time_secs = qeGetQueryTimeout (hdbc)    ;
/* Will return default of 0 (wait indefinitely). *     /
```

See Also

qeSetQueryTimeout.

qeGetSelectOptions

qeGetSelectOptions returns the option flag settings that determine fetching behavior during the current database connection. These options affect the level of fetching allowed in the current connection, whether logging is used when not made necessary by the database system, and the extent to which the result set persists after a transaction ends.

Syntax

```
int32 flags qeGetSelectOptions (int16   hdbc)
```

Parameters

hdbc is a connection returned from qeConnect.

flags is the set of option flags, which can include the following:

Constant	Value	Description
qeFETCH_FORWARD_DIR	0x0001	Only forward fetching is allowed. This is the default fetching behavior option.
qeFETCH_ANY_DIR	0x0002	Random and previous fetching is enabled.
qeLOG_IF_NEEDED	0x0008	Use log file only as needed to enable previous and random fetching. This is the default logging behavior.
qeLOG_ALWAYS	0x0010	Force use of log file when it is not required. (This does not activate random fetching if it is not explicitly set with qeFETCH_ANY_DIR).
qeSELECT_INVALIDATE	0x0020	Disable fetching at the end of transaction (EOT). Calls made after a commit or rollback to any function except qeEndSQL cause an error.

Constant	Value	Description
qeSELECT_TRUNCATE	0x0040	Truncate the result set at EOT. This option lets you continue fetching only those records that have already been read from the database (if qeFETCH_ANY_DIR is set).
qeSELECT_PERSIST	0x0060	The result set persists at EOT. This is the default behavior, which lets you continue fetching from the entire set of records returned by the Select statement. To enable this behavior for databases that invalidate the *hstmt* at commit or rollback, the records in the result set that have not been fetched by EOT are written to a log file.

These values can be combined by adding them together or joining them with an OR clause.

Example

```
hdbc = qeConnect ("DSN=QESS")   ;
options = qeGetSelectOptions (hdbc)   ;
res_code = qeDisconnect (hdbc)   ;
```

See Also

qeSetSelectOptions.

qeGetSupportedIsolationLevels

qeGetSupportedIsolationLevels returns the isolation levels supported by the database system.

Syntax

```
int16 levels qeGetSupportedIsolationLevels (int16 hdbc)
```

Parameters

hdbc is the handle to the database connection returned by qeConnect.

levels is the set of isolation levels supported by the database system. One of the following flags is set for each isolation level supported:

Constant	Value	Description
qeISO_READ_UNCOMMITTED	0x0001	Read uncommitted (0) isolation level. Locks are obtained on modifications to the database and held until end of transaction (EOT). Reading from the database does not involve any locking.
qeISO_READ_COMMITTED	0x0002	Read committed (1) isolation level. Locks are acquired for reading and modifying the database. Locks are released after reading but locks on modified objects are held until EOT.
qeISO_REPEATABLE_READ	0x0004	Repeatable read (2) isolation level. Locks are obtained for reading and modifying the database. Locks on all modified objects are held until EOT. Locks obtained for reading data are held until EOT. Locks on non-modified access structures (indexes, hashing structures, etc.) are released after reading.

Constant	Value	Description
qeISO_SERIALIZABLE	0x0008	Serializable (3) isolation level. All data read or modified is locked until EOT. All access structures that are modified are locked until EOT. Access structures used by the query are locked until EOT.
qeISO_VERSIONING	0x0010	Versioning (4) isolation level. Similar to isolation level 3, serializable, but provides greater concurrence through the use of non-locking “record versioning” protocols.

The isolation levels supported and default isolation level are database-dependent.

Example

```
hdbc = qeConnect ("DSN=QESS")   ;
levels = qeGetSupportedIsolationLevels (hdbc)    ;
cur_level = qeGetIsolationLevel (hdbc)    ;
if (levels & qeISO_READ_COMMITTED    )
    res_code = qeSetIsolationLevel (hdbc,
             qeISO_READ_COMMITTED)  ;
res_code = qeDisconnect (hdbc)   ;
```

See Also

qeGetIsolationLevel, qeSetIsolationLevel.

qeGetTableCaching

qeGetTableCaching returns the caching setting specified in the last call to qeSetTableCaching.

Syntax

```
int16 setting qeGetTableCaching (int16   hdbc)
```

Parameters

setting is one of the following:

Constant	Value	Description
qeCACHE_PERMANENT	1	Turn caching on, and have the cache file remain after the connection terminates. You must specify a file name with the qeSetCacheFileName function when using this option.
qeCACHE_SESSION	2	Turn caching on for this session. The cache file is deleted when the connection terminates. This is the default.
qeCACHE_OFF	3	Turn caching off.

hdbc is the handle to the database connection returned by qeConnect.

Example

```
/* Cache_Permanent *  /
hdbc = qeConnect ("DSN=QEDBF")   ;
setting = qeGetTableCaching (hdbc)   ;
res_code = qeDisconnect (hdbc)   ;
```

See Also

qeSetTableCaching.

qeGetTraceOptions

qeGetTraceOptions returns the current trace options.

Syntax

```
int16 flags qeGetTraceOptions (  )
```

Parameters

flags is a set of option flags that defines the tracing options in effect. They can be:

Constant	Value	Description
qeTRACE_NON_VAL_CALLS	0x0001	Trace all non-qeVal calls.
qeTRACE_USER	0x0002	Trace strings sent via qeTraceUser.
qeTRACE_VAL_CALLS	0x0004	Trace qeVal calls and bound data at fetch time.
qeTRACE_WINDOW	0x0008	Write all trace information (except ODBC calls) to a trace window.
qeTRACE_ODBC	0x0010	Trace ODBC calls.

Example

```
res_code = qeTraceOn ("\\trace.txt")   ;
trc_val = qeGetTraceOptions ()   ;
hdbc = qeConnect ("DSN=QEDBF")   ;
...
res_code = qeDisconnect (hdbc)   ;
res_code = qeTraceOff ()  ;
```

See Also

qeSetTraceOptions.

qeIndexes

qeIndexes creates a statement execution (*hstmt*) that returns information on the set of indexes for a table.

Syntax

```
int16 hstmt = qeIndexes  (
    int16    hdbc,
    ptrstr   table_name,
    int16    flags)
```

Description

qeIndexes returns one record for each column in each index. Each record contains the following columns:

Column	Type	Description
Table Qualifier	Char(128)	Table qualifier. This is a path for file-based databases. May be NULL.
Table User	Char(128)	Table user. May be NULL.
Table Name	Char(128)	Table name.
Nonunique	Int16	Indicates whether every index entry must be unique or not. Values: 0 = FALSE if the index values must be unique. 1 = TRUE if the index values do not have to be unique; can be nonunique.
Index Qualifier	Char(128)	Index qualifier. May be needed in a DROP INDEX statement.
Index Name	Char(128)	Index name.
Index Type	Int16	Type of index. Values: 1 = qeINDEX_CLUSTERED 2 = qeINDEX_HASHED 3 = qeINDEX_OTHER

Column	Type	Description
Sequence No	Int16	The number of this column within the index. For example, for the index LAST_NAME, FIRST_NAME, the Sequence No would be 1 in the row returned for LAST_NAME and 2 in the row returned for FIRST_NAME.
Column Name	Char(128)	Column name.
Collation	Char(1)	Collating sequence. Values: A = qeINDEX_ASCENDING D = qeINDEX_DESCENDING NULL = qeINDEX_ORDER_UNKNOWN
Cardinality	Int32	Number of unique values in index; may be NULL
Pages	Int32	Number of pages used to store index; may be NULL
Filter	Char(128)	The filter condition when one exists. Otherwise, the value is NULL. For example, SALARY > 25000.

Not all database systems support indexes. You should include error-checking code to handle those database systems that do not.

Parameters

hstmt is the handle to the statement returned by qeIndexes.

hdbc is a handle to a database connection obtained from qeConnect.

table_name is the table whose indexes are to be returned.

flags is a set of option flags that control the values returned from qeIndexes. Each of these options overrides the DTK default. They can be combined by adding them together or joining them with an OR clause.

Constant	Value	Description
qeUNIQUE_INDEXES	0x0001	Return only unique indexes; returning all indexes is default
qeACCURATE_STATS	0x0002	Always request statistics from server, even if it takes a long time; quick retrieval is the default.

Example

```
hdbc = qeConnect (“DSN=QESS;DLG=1”)   ;
hstmt = qeIndexes (hdbc, “EMP”, qeACCURATE_STATS);
while (qeFetchNext (hstmt) == qeSUCCESS)     {
    /* Get info about Indexes *   /
}
```

qeLibInit

qeLibInit initializes a DTK program.

Syntax

```
int16 res_code qeLibInit ( )
```

Description

qeLibInit initializes an individual DTK program by allocating memory for that program. Whenever possible, programs that call the DTK API should call this function before making any other calls.

If you write a multi-threaded application, you should call this function to initialize each thread of execution.

Some programming structures make it impossible to call qeLibInit for every instance of DTK calls. For example, a DLL shared by multiple applications cannot know whether or not the calling application had already called qeLibInit or qeLibTerm. Even so, by using these functions whenever possible you can keep more memory available to your applications.

Parameters

res_code is the result code returned by qeLibInit, which returns the same set of result codes as qeErr. See Appendix D, “Result and Error Message Codes,” on page 537 for a list of these result codes.

Example

```
res_code = qeLibInit ()  ;
...
res_code = qeLibTerm ()  ;
```

See Also

qeLibTerm.

qeLibTerm

qeLibTerm terminates a DTK program.

Syntax

```
int16 res_code qeLibTerm ( )
```

Description

qeLibTerm terminates a DTK program and frees the memory allocated for that program by the corresponding call to qeLibInit. Whenever possible, programs that call the DTK API should call this function as the last DTK function call.

If you write a multi-threaded application, you should call this function to terminate each thread of execution.

Some programming structures make it impossible to call qeLibInit for every instance of DTK calls. For example, a DLL shared by multiple applications cannot know whether or not the calling application has already called qeLibInit or qeLibTerm. Even so, by using these functions whenever possible you can keep more memory available to your applications.

Parameters

res_code is the result code returned by qeLibTerm, which returns the same set of result codes as qeErr. See Appendix D, “Result and Error Message Codes,” on page 537 for a list of these result codes.

Example

```
res_code = qeLibInit ()  ;
...
res_code = qeLibTerm ()  ;
```

See Also

qeLibInit.

qeMoreResults

qeMoreResults begins a new result set from statements or stored procedures that return multiple result sets.

Syntax

```
int16 res_code qeMoreResults (int16   hstmt)
```

Description

qeMoreResults ends the current result set and starts a new one. If the *res_code* is qeEOF, then there are no more result sets. Otherwise, *hstmt* represents the new result set. Some drivers do not support this function.

Parameters

hstmt is the handle to the statement returned by qeExecSQL or qeSQLPrepare.

res_code is the result code returned by qeMoreResults, which returns the same set of result codes as qeErr. See Appendix D, "Result and Error Message Codes," on page 537 for a list of these result codes.

Example

```
hdbc = qeConnect ("DSN=QESS")   ;
hstmt = qeExecSQL (hdbc, "sp_empdept")    ;
/* sp_empdept is a stored procedure containing *     /
/* "SELECT * FROM emp;SELECT * FROM dept" *    /

while (qeFetchNext (hstmt) == qeSUCCESS)     {
    /* Get values from emp *   /
    ...
}

res_code = qeMoreResults (hstmt)   ;
if (res_code != EOF)   {
    while (qeFetchNext (hstmt) == qeSUCCESS)     {
             /* Get values from dept *   /
             ...
    }
}
```

```
res_code = qeEndSQL (hstmt)   ;
res_code = qeDisconnect (hdbc)   ;
```

See Also qeProcedureColumns.

qeNativeSQL and qeNativeSQLBuf

These functions return the SQL string as translated by the driver.

Syntax

```
ptrstr native_sql qeNativeSQL (int16   hstmt)
```

```
int16 res_code qeNativeSQLBuf (int16   hstmt, ptrstr
stmt_buf)
```

Description

qeNativeSQL returns a pointer to the translated SQL string. This string is stored in a buffer maintained by DTK. You must copy the string out of this buffer before you call another DTK function, because the next function may use the same buffer.

With qeNativeSQLBuf, you pass in a pointer to a buffer you have allocated. The translated SQL string is put in the buffer. You must make sure that the buffer is large enough to hold the returned string.

This function depends on driver support and returns an error if the driver does not support the ODBC function SQLNativeSql.

Parameters

hstmt is the handle to the statement returned by qeExecSQL or qeSQLPrepare.

stmt_buf points to an allocated buffer for the resulting statement.

res_code is the result code returned by qeNativeSQLBuf, which returns the same set of result codes as qeErr. See Appendix D, "Result and Error Message Codes," on page 537 for a list of these result codes.

Example

```
hdbc = qeConnect ("DSN=QEDBF")   ;
hstmt = qeSQLPrepare (hdbc, "SELECT * FROM em     p
    WHERE LAST_NAME = 'Woltman'")   ;
native = qeNativeSQL (hstmt)   ;
...
res_code = qeEndSQL (hstmt)   ;
res_code = qeDisconnect (hdbc)   ;
```

qeNumCols

qeNumCols returns the number of columns present in a SQL Select statement.

Syntax

```
int16 num_cols qeNumCols (int16  hstmt)
```

Parameters

hstmt is the handle to the statement returned by qeExecSQL or qeSQLPrepare.

num_cols is the returned number of columns. Its value is 0 if the statement is not a Select statement.

Example

To determine the number of columns in the dBASE employee file:

```
hdbc = qeConnect ("DSN=QEDBF")   ;
...
hstmt = qeExecSQL (hdbc, "SELECT * FROM emp")    ;
num_cols = qeNumCols (hstmt)   ;
...
res_code = qeEndSQL (hstmt)   ;
res_code = qeDisconnect (hdbc)   ;
```

See Also

qeExecSQL.

qeNumModRecs

qeNumModRecs returns the number of records modified by the last function called that modified the database.

Syntax

```
int32 num_recs qeNumModRecs (int16  hstmt)
```

Description

qeNumModRecs returns the number of records modified by a SQL Insert, Update, or Delete statement, qeRecUpdate, qeRecDelete, qeApplyAll, or auto-update operation.

Parameters

hstmt is the handle to the statement returned by qeExecSQL.

num_recs is the returned number of records. Returns 0 if the statement is a Select statement.

Example

To determine the number of records modified by an Update statement to the dBASE employee file:

```
hdbc = qeConnect ("DSN=QEDBF")   ;
...
hstmt = qeExecSQL (hdbc, "UPDATE emp.dbf SET
salary=salary*1.1 WHERE dept='D101'")   ;
num_recs = qeNumModRecs (hstmt)   ;
res_code = qeEndSQL (hstmt)   ;
res_code = qeDisconnect (hdbc)   ;
```

See Also

qeExecSQL.

qeNumParams

qeNumParams returns the number of parameters that appeared in the statement.

Syntax

```
int16 num_params qeNumParams (int16  hstmt)
```

Parameters

hstmt is the handle to the statement returned by qeSQLPrepare.

num_params is the number returned by qeNumParams.

Example

```
hdbc = qeConnect ("DSN=QEDBF")   ;
hstmt = qeSQLPrepare (hdbc, "SELECT * FROM em    p
    WHERE last_name = ?")  ;
num_params = qeNumParams (hstmt); /* Will return 1 *     /
res_code = qeSQLExecute (hstmt)   ;
res_code = qeEndSQL (hstmt)   ;
res_code = qeDisconnect (hdbc)   ;
```

qeParamNum

qeParamNum returns the number of the parameter corresponding to a specified name.

Syntax

```
int16 param_num qeParamNum (int16  hstmt, ptrstr
param_name)
```

Description

qeParamNum returns the number of the parameter that corresponds to *param_name*. Use this function to specify parameters by name in functions that take parameters by number.

If a parameter name is used more than once in the statement, the position of the first occurrence is returned. Setting or binding this position binds for all parameters with the same name.

Parameters

hstmt is the handle to the statement returned by qeSQLPrepare.

param_name is the name of a parameter for *hstmt*.

param_num is the parameter number returned by qeParamNum. If the parameter name does not correspond to any of the parameters in *hstmt*, its value is 0.

Example

```
hdbc = qeConnect ("DSN=QEDBF")   ;
hstmt = qeSQLPrepare (hdbc, "SELECT * FROM em    p
    WHERE last_name = ?last")  ;
res_code = qeSetParamChar (hstmt,
    qeParamNum (hstmt, "last"),"Smith", 10)    ;
res_code = qeSQLExecute (hstmt)   ;
res_code = qeEndSQL (hstmt)   ;
res_code = qeDisconnect (hdbc)   ;
```

qeProcedureColumns

qeProcedureColumns returns a description of the parameters to a specified stored procedure and the result columns for that procedure.

Syntax

```
int16 hstmt qeProcedureColumns (int16   hdbc, ptrstr
proc_name)
```

Description

qeProcedureColumns returns an *hstmt* for a result set describing the parameters to a stored procedure and the result columns for that procedure. The resulting records contain the following columns:

Column	Type	Description
Procedure Qualifier	Char(128)	Procedure qualifier identifier
Procedure Owner	Char(128)	Procedure owner identifier
Procedure Name	Char(128)	Procedure identifier
Column Name	Char(128)	Procedure column identifier
Column Type	Int16	Result type: qePARAM_UNKNOWN, qePARAM_INPUT, qePARAM_INOUT, qePARAM_OUT, qeRESULT_COL, qeRETURN_VAL
Data Type	Int16	Data type
DB Type Name	Char(128)	Data source-dependent type name
Width	Int16	Data type size
Attr1	Int16	Precision for decimal types, date start position for dates, null otherwise.
Attr2	Int16	Scale for decimal types, date end position for dates, null otherwise.

Column	Type	Description
Nullable	Int16	Result type: qeCOL_NULLABLE, qeCOL_NOT_NULLABLE, qeCOL_UNKNOWN
Remarks	Char(256)	Description of column (if available).

You retrieve this information like you would other database values—using the qeVal, qeBindCol, and qeFetch functions.

Parameters

hdbc is the handle to a connection returned by qeConnect.

proc_name is a name or pattern of the procedure to find. If the pattern is "%" or "*", all procedures are selected. You can also specify the qualifier name, owner name, or both.

hstmt is the handle to the statement returned by qeProcedureColumns. Its value is null if the database does not store the procedure.

Example

```
hdbc = qeConnect ("DSN=QESS;DLG=1")    ;
hstmt = qeProcedureColumns (hdbc, "sp_who")    ;
while (qeFetchNext (hstmt) == qeSUCCESS)     {
    ...
/* Get info about stored procedure columns. *    /
    ...
}
res_code = qeDisconnect (hdbc)   ;
```

See Also

qeMoreResults.

qePrimaryKeys

qePrimaryKeys creates a statement execution (*hstmt*) that returns information on the set of columns that compose a table's primary keys.

Syntax

```
int16 hstmt = qePrimaryKeys   (
    int16    hdbc,
    ptrstr   table_name)
```

Description

qePrimaryKeys returns one record per column in the primary key. Each record contains the following columns:

Column	Type	Description
Table Qualifier	Char(128)	Table qualifier. This is a path for file-based databases. May be NULL.
Table User	Char(128)	Table user. May be NULL.
Table Name	Char(128)	Table name.
Column Name	Char(128)	Column name.
Sequence No	Int16	Column sequence number, which is the number of this column within the primary key. For example, for the primary key LAST_NAME, FIRST_NAME, the Sequence No would be 1 in the row returned for LAST_NAME and 2 in the row returned for FIRST_NAME.
Index Name	Char(128)	Primary key name. NULL if not applicable to the data source.

Not all database systems support primary keys. You should include error-checking code to handle those database systems that do not.

Parameters

hstmt is the handle to the statement returned by qePrimaryKeys.

hdbc is a handle to a database connection obtained from qeConnect.

table_name is the table whose primary keys are to be returned.

Example

```
hdbc = qeConnect ("DSN=QESS;DLG=1")   ;
hstmt = qePrimaryKeys (hdbc, "EMP");
while (qeFetchNext (hstmt) == qeSUCCESS)     {
    /* Get info about Primary Keys *   /
}
```

See Also

qeForeignKeys.

qePutBinary

qePutBinary updates a column with binary data bytes.

Syntax

```
int16  res_code qePutBinary  (
    int16    hstmt,
    int16    col_num,
    ptrstr   new_val,
    int32    val_len)
```

Description

qePutBinary updates a column value in the current record buffer with a specified number of binary data bytes.

This function does not change the value in the database. The new value is sent to the database when qeRecUpdate is called or if auto-updating has been enabled for the *hstmt* and the current record position changes.

Parameters

hstmt is the statement handle returned by qeExecSQL or qeSQLPrepare.

col_num is the column number whose field is to be modified. The first column number is 1.

new_val is a buffer of binary data.

val_len is the number of bytes to use from the new_val buffer.

res_code is the result code returned by qePutBinary, which returns the same set of result codes as qeErr. See Appendix D, “Result and Error Message Codes,” on page 537 for a list of these result codes.

Example

```
hdbc = qeConnect ("DSN=QEDBF")   ;
hstmt = qeExecSQL (hdbc, "SELECT interests FROM emp")     ;
res_code = qeFetchNext (hstmt)   ;
res_code = qePutBinary (hstmt, 1, bindata, bin_len)     ;
res_code = qeRecUpdate (hstmt)   ;
res_code = qeEndSQL (hstmt)   ;
res_code = qeDisconnect (hdbc)   ;
```

qePutChar

qePutChar updates a column with a character value.

Syntax

```
int16 res_code qePutChar  (
    int16    hstmt,
    int16    col_num,
    ptrstr   fmt_string,
    ptrstr   new_val)
```

Description

qePutChar updates a column value in the current record buffer with a null-terminated character value.

A format string can be used if formatting is desired and the column type is a date/time or a number.

This function does not change the value in the database. The new value is sent to the database when qeRecUpdate is called or if auto updating has been enabled for the *hstmt* and the current record position changes.

Parameters

hstmt is the handle to the statement returned by qeExecSQL or qeSQLPrepare.

col_num is the column number whose field is to be modified. The first column number is 1.

fmt_string is a pointer to a null-terminated format string which controls the formatting of dates and numbers.

new_val is a null-terminated character string which holds the new value for the column.

res_code is the result code returned by qePutChar, which returns the same set of result codes as qeErr. See Appendix D, “Result and Error Message Codes,” on page 537 for a list of these result codes.

Example

```
hdbc = qeConnect ("DSN=QEDBF")   ;
hstmt = qeExecSQL (hdbc, "SELECT first_name FROM emp")     ;
res_code = qeFetchNext (hstmt)   ;

/* Update the record. *  /
res_code = qePutChar (hstmt, 1, "", "Joe")    ;
res_code = qeRecUpdate (hstmt)   ;

res_code = qeEndSQL (hstmt)   ;
res_code = qeDisconnect (hdbc)   ;
```

qePutDecimal

qePutDecimal updates a column with a decimal value.

Syntax

```
int16 res_code qePutDecimal  (
    int16    hstmt,
    int16    col_num,
    int16    precision,
    int16    scale,
    ptrstr   new_val)
```

Description

qePutDecimal updates a column value in the current record buffer with a decimal value.

This function does not change the value in the database. The new value is sent to the database when qeRecUpdate is called or if auto updating has been enabled for the *hstmt* and the current record position changes.

Parameters

hstmt is the handle to the statement returned by qeExecSQL or qeSQLPrepare.

col_num is the column number whose field is to be modified. The first column number is 1.

precision is the number of significant digits in the result.

scale is the number of digits to the right of the decimal point in the result.

new_val is a pointer to a string that holds the new value for the column.

res_code is the result code returned by qePutDecimal, which returns the same set of result codes as qeErr. See Appendix D, "Result and Error Message Codes," on page 537 for a list of these result codes.

Example

```
hdbc = qeConnect ("DSN=QEDBF")   ;
hstmt = qeExecSQL (hdbc, "SELECT salary FROM emp")     ;
res_code = qeFetchNext (hstmt)   ;
/* Update the record. *  /
res_code = qePutDecimal (hstmt, 1, 9, 2, dec_val)     ;
res_code = qeRecUpdate (hstmt)   ;
res_code = qeEndSQL (hstmt)   ;
res_code = qeDisconnect (hdbc)   ;
```

qePutDouble

qePutDouble updates a column with a double-precision floating-point value.

Syntax

```
int16  res_code qePutDouble  (
    int16    hstmt,
    int16    col_num,
    float64  new_val)
```

Description

qePutDouble updates a column value in the current record buffer with a double-precision floating-point value.

This function does not change the value in the database. The new value is sent to the database when qeRecUpdate is called or if auto updating has been enabled for the *hstmt* and the current record position changes.

Parameters

hstmt is the statement handle returned by qeExecSQL or qeSQLPrepare.

col_num is the column number whose field is to be modified. The first column number is 1.

new_val is a double-precision floating-point value which is the new value for the column.

res_code is the result code returned by qePutDouble, which returns the same set of result codes as qeErr. See Appendix D, "Result and Error Message Codes," on page 537 for a list of these result codes.

Example

```
hdbc = qeConnect ("DSN=QEDBF")   ;
hstmt = qeExecSQL (hdbc, "SELECT salary FROM emp")     ;
res_code = qeFetchNext (hstmt)   ;
res_code = qePutDouble (hstmt, 1, 10000.50)    ;
res_code = qeRecUpdate (hstmt)   ;
res_code = qeEndSQL (hstmt)   ;
res_code = qeDisconnect (hdbc)   ;
```

qePutFloat

qePutFloat updates a column with a single-precision floating-point value.

Syntax

```
int16 res_code qePutFloat   (
    int16     hstmt,
    int16     col_num,
    float32   new_val)
```

Description

qePutFloat updates a column value in the current record buffer with a single-precision floating-point value.

This function does not change the value in the database. The new value is sent to the database when qeRecUpdate is called or if auto updating has been enabled for the *hstmt* and the current record position changes.

Parameters

hstmt is the handle to the statement returned by qeExecSQL or qeSQLPrepare.

col_num is the column number whose field is to be modified. The first column number is 1.

new_val is a single-precision floating-point value which is the new value for the column.

res_code is the result code returned by qePutFloat, which returns the same set of result codes as qeErr. See Appendix D, “Result and Error Message Codes,” on page 537 for a list of these result codes.

Example

```
hdbc = qeConnect ("DSN=QEDBF")   ;
hstmt = qeExecSQL (hdbc, "SELECT salary FROM emp")     ;
res_code = qeFetchNext (hstmt)   ;

res_code = qePutFloat (hstmt, 1, 10000.50)    ;
res_code = qeRecUpdate (hstmt)   ;
res_code = qeEndSQL (hstmt)   ;
res_code = qeDisconnect (hdbc)   ;
```

qePutInt

qePutInt updates a column with a 2-byte integer.

Syntax

```
int16 res_code qePutInt  (
    int16    hstmt,
    int16    col_num,
    int16    new_val)
```

Description

qePutInt updates a column value in the current record buffer with a 2 byte signed integer.

This function does not change the value in the database. The new value is sent to the database when qeRecUpdate is called or if auto updating has been enabled for the *hstmt* and the current record position changes.

Parameters

hstmt is the handle to the statement returned by qeExecSQL or qeSQLPrepare.

col_num is the column number whose field is to be modified. The first column number is 1.

new_val is a 2-byte signed integer which is the new value for the column.

res_code is the result code returned by qePutInt, which returns the same set of result codes as qeErr. See Appendix D, “Result and Error Message Codes,” on page 537 for a list of these result codes.

Example

```
hdbc = qeConnect ("DSN=QEDBF")   ;
hstmt = qeExecSQL (hdbc, "SELECT salary FROM emp")     ;
res_code = qeFetchNext (hstmt)   ;

res_code = qePutInt (hstmt, 1, 10000)    ;
res_code = qeRecUpdate (hstmt)   ;

res_code = qeEndSQL (hstmt)   ;
res_code = qeDisconnect (hdbc)   ;
```

qePutLong

qePutLong updates a column with a 4-byte integer.

Syntax

```
int16 res_code qePutLong  (
    int16 hstmt,
    int16 col_num,
    int32 new_val)
```

Description

qePutLong updates a column value in the current record buffer with a 4-byte integer.

This function does not change the value in the database. The new value is sent to the database when qeRecUpdate is called or if auto updating has been enabled for the *hstmt* and the current record position changes.

Parameters

hstmt is the handle to the statement returned by qeExecSQL or qeSQLPrepare.

col_num is the column number whose field is to be modified. The first column number is 1.

new_val is a 4-byte integer that is the new value for the column.

res_code is the result code returned by qePutLong, which returns the same set of result codes as qeErr. See Appendix D, “Result and Error Message Codes,” on page 537 for a list of these result codes.

Example

```
hdbc = qeConnect ("DSN=QEDBF")   ;
hstmt = qeExecSQL (hdbc, "SELECT salary FROM emp")     ;
res_code = qeFetchNext (hstmt)   ;

res_code = qePutLong (hstmt, 1, 10000)    ;
res_code = qeRecUpdate (hstmt)   ;

res_code = qeEndSQL (hstmt)   ;
res_code = qeDisconnect (hdbc)   ;
```

qePutNull

qePutNull updates a column to have the value null.

Syntax

```
int16 res_code qePutNull (int16  hstmt, int16 col_num)
```

Description

qePutNull updates a column value in the current record buffer to have the value null.

This function does not change the value in the database. The new value is sent to the database when qeRecUpdate is called or if auto updating has been enabled for the *hstmt* and the current record position changes.

Parameters

hstmt is the handle to the statement returned by qeExecSQL or qeSQLPrepare.

col_num is the column number whose field is to be modified. The first column number is 1.

res_code is the result code returned by qePutNull, which returns the same set of result codes as qeErr. See Appendix D, “Result and Error Message Codes,” on page 537 for a list of these result codes.

Example

```
hdbc = qeConnect ("DSN=QEDBF")   ;
hstmt = qeExecSQL (hdbc, "SELECT hire_date FROM emp")     ;
res_code = qeFetchNext (hstmt)   ;

res_code = qePutNull (hstmt, 1)   ;
res_code = qeRecUpdate (hstmt)   ;
res_code = qeEndSQL (hstmt)   ;
res_code = qeDisconnect (hdbc)   ;
```

qePutUsingBindColumns

qePutUsingBindColumns updates columns with the values in the bind buffers.

Syntax

```
int16 res_code qePutUsingBindColumns (int16    hstmt)
```

Description

qePutUsingBindColumns updates column values in the current record with the values in the bind buffers.

If the length value of the bound column is set to qeNO_DATA_CHANGE (-9), then the column is not updated. You can use this function to put a value of null by setting the bound column's length value to qeNULL_DATA (-2).

This function does not change the value in the database. The new value is sent to the database when qeRecUpdate is called or if auto updating has been enabled for the *hstmt* and the current record position changes.

Parameters

hstmt is the handle to the statement returned by qeExecSQL or qeSQLPrepare.

res_code is the result code returned by qePutUsingBindColumns, which returns the same set of result codes as qeErr. See Appendix D, "Result and Error Message Codes," on page 537 for a list of these result codes.

Example

```
char   first_name[9] ;
long   fn_length ;

hdbc = qeConnect ("DSN=QEDBF")   ;
hstmt = qeExecSQL (hdbc, "SELECT first_name FROM emp")     ;
fn_length = 9 ;
qeBindCol (hstmt, 1, first_name, &fn_length)    ;
while (qeFetchNext (hstmt) == 0)    {
    /* qeFetchNext has automatically filled   *    /
    /* first_name                             *    /
    /*                                        *    /
    /* If the first name is David then change *    /
    /* to Dave and insert this new value.     *    /

    if (strcmp (first_name, "David") == 0)     {
             strcpy (first_name, "Dave")   ;
             fn_length = 4 ;
             }

    qePutUsingBindColumns (hstmt)   ;
    res_code = qeRecUpdate (hstmt)   ;
}
res_code = qeEndSQL (hstmt)   ;
res_code = qeDisconnect (hdbc)   ;
```

qeQBEPrepare

qeQBEPrepare creates a new *hstmt* that contains the Where clause conditions that were created for the original *hstmt* by calls to the qeRecSetCondition functions.

Syntax

```
int16 new_hstmt qeQBEPrepare (int16   old_hstmt)
```

Description

qeQBEPrepare creates a new *hstmt* that contains the Where clause conditions that were created for the original *hstmt* by calls to qeRecSetCondition functions.

The new *hstmt* inherits the parameters from the original *hstmt*. Make appropriate parameter routine calls to change these parameters.

After you have made one or more calls to qeRecSetCondition, call qeQBEPrepare to add all the conditions to the Select statement's Where clause. Call qeSQLExecute to execute the resulting statement. Subsequent calls to the qeFetch functions retrieve the records that result from the modified Select statement.

Parameters

new_hstmt is the handle to a SQL statement to which a Where clause containing QBE conditions has been added.

old_hstmt is the handle to an existing SQL statement to which you want to add a Where clause containing QBE conditions.

Example

```
hdbc = qeConnect ("DSN=QEDBF")   ;
hstmt = qeExecSQL (hdbc, "SELECT * FROM emp")    ;
res_code = qeRecSetConditionChar (hstmt, 1    ,
    qeFIND_EQUAL, "David", "", FALSE)   ;
new_hstmt = qeQBEPrepare (hstmt)   ;
res_code = qeSQLExecute (new_hstmt)   ;
res_code = qeEndSQL (hstmt)   ;
res_code = qeDisconnect (hdbc)   ;
```

See Also

qeRecSetCondition functions, qeRecFind.

qeQryAllocate

qeQryAllocate builds a query based on a string containing a SQL statement.

Syntax

```
int16 hqry qeQryAllocate (int16  hdbc, ptrstr  statement)
```

Description

qeQryAllocate builds a query based on *statement*, which may be null. It returns a query handle (*hqry*), which may be used to communicate with the Query Builder.

Parameters

hqry is the handle to a query returned by qeQryAllocate.

hdbc is a handle to a database connection obtained from qeConnect.

statement is a pointer to a string containing a SQL statement. It may be Null if no statement is to be associated with the returned *hqry*.

Example

```
hdbc = qeConnect ("DSN=QEDBF")   ;
hqry = qeQryAllocate (hdbc,"SELECT * FROM emp.dbf");
if (hqry == 0 )
    res_code = qeQryBuilder (hqry,0,qeQRY_TABLES,
             qeQRY_DEFAULT) ;
...
res_code = qeQryFree (hqry)   ;
res_code = qeDisconnect (hdbc)   ;
```

See Also

qeQryBuilder, qeQryOpenQueryFile.

qeQryBuilder

qeQryBuilder runs the Query Builder.

Syntax

```
int16 res_code qeQryBuilder  (
    int16    hqry,
    int16    parent_window,
    int16    flags,
    int16    init_dialog)
```

Description

qeQryBuilder runs the Query Builder, based on the query represented by *hqry*. Any editing applied via the Query Builder affects this query.

An *hqry* can be obtained by calling qeQryAllocate or qeQryOpenQueryFile.

Parameters

hqry is a handle to a query obtained from qeQryAllocate or qeQryOpenQueryFile.

parent_window is a handle to the parent window of the calling application. It may be 0.

flags is a set of option flags that control the behavior and appearance of the Query Builder. Each of these options overrides the DTK default. They can be combined by adding them together or joining them with an OR clause. They include the following:

Constant	Value	Description
qeQRY_NO_COL_ALIAS	0x0001	Column aliases not allowed.
qeQRY_EXIT_AFTER_DLG	0x0002	Exit after first dialog box is exited. Valid only if initial dialog specified.
qeQRY_ALLOW_SRC_ CHANGE	0x0004	Source can be changed in file open box.
qeQRY_SYSTABLES	0x0008	List system tables in table dialog box.
qeQRY_SYNONYMS	0x0010	List synonyms in table dialog box.

Constant	Value	Description
qeQRY_TABLES	0x0020	List tables in table dialog box.
qeQRY_VIEWS	0x0040	List views in table dialog box.
qeQRY_NO_PARAMS	0x0080	Disallow parameters.
qeQRY_BIG_ICONS	0x0100	Use big icons in icon bar.
qeQRY_VALIDATE	0x0200	Validate SQL.
qeQRY_SAMPLE	0x0400	Show sample values in conditions dialog box.

init_dialog specifies the initial dialog box to be displayed when the query builder is called. Valid values are:

Constant	Value	Description
qeQRY_DEFAULT	1	Bring up the default initial dialog.
qeQRY_FILE	2	File dialog.
qeQRY_JOIN	3	Join dialog
qeQRY_SELECT	4	Select dialog.
qeQRY_ORDER	5	Order by dialog.
qeQRY_WHERE	6	Where dialog.
qeQRY_GROUP	7	Group by dialog.
qeQRY_HAVING	8	Having dialog.
qeQRY_TEXT	9	Edit query text dialog.

res_code is the result code returned by qeQryBuilder, which returns the same set of result codes as qeErr. See Appendix D, “Result and Error Message Codes,” on page 537 for a list of these result codes.

Example

```
hdbc = qeConnect ("DSN=QEDBF")   ;
if (hdbc == 0 )
    hqry = qeQryAllocate (hdbc,"");
if (hqry == 0 )
    res_code = qeQryBuilder (hqry,0,qeQRY_TABLES,
             qeQRY_DEFAULT) ;
...
res_code = qeQryFree (hqry)   ;
res_code = qeDisconnect (hdbc)   ;
```

See Also qeQryAllocate, qeQryOpenQueryFile.

qeQryFree

qeQryFree frees the memory associated with an *hqry*. It is important to call qeQryFree to free system resources when you are finished using an *hqry*.

Syntax

```
int16  res_code qeQryFree (int16   hqry)
```

Parameters

hqry is the handle to the query which is to be freed, which was obtained from qeQryAllocate or qeQryOpenQueryFile.

res_code is the result code returned by qeQryFree, which returns the same set of result codes as qeErr. See Appendix D, “Result and Error Message Codes,” on page 537 for a list of these result codes.

Example

```
hdbc = qeConnect ("DSN=QEDBF")   ;
hqry = qeQryAllocate (hdbc,"");
res_code = qeQryBuilder (hqry,0,qeQRY_TABLES,
qeQRY_DEFAULT) ;
...
res_code = qeQryFree (hqry)   ;
res_code = qeDisconnect (hdbc)   ;
```

qeQryGetFileName and qeQryGetFileNameBuf

qeQryGetFileName and qeQryGetFileNameBuf return the file name, if any, associated with the query represented in *hqry*.

Syntax

```
ptrstr file_name qeQryGetFileName (int16   hqry)
```

```
int16 res_code qeQryGetFileNameBuf (int16   hqry,
    ptrstr file_name)
```

Description

qeQryGetFileName and qeQryGetFileNameBuf return the file name, if any, associated with the query represented in *hqry*.

qeQryGetFileName returns a pointer to the file name string. This string is stored in a buffer maintained by DTK. You must copy the string out of this buffer before you call another DTK function, because the next function may use the same buffer.

With qeQryGetFileNameBuf, you pass in a pointer to a buffer you have allocated. The file name string is put in the buffer. You must make sure that the buffer is large enough to hold the returned string.

Parameters

hqry is a handle to a query obtained from qeQryAllocate or qeQryOpenQueryFile.

file_name points to a buffer to hold the returned file name.

res_code is the result code returned by qeQryGetFileNameBuf, which returns the same set of result codes as qeErr. See Appendix D, “Result and Error Message Codes,” on page 537 for a list of these result codes.

Example

```
hdbc = qeConnect ("DSN=QEDBF")   ;
hqry = qeQryOpenQueryFile ("query1.qef")    ;
res_code = qeQrySetHdbc (hqry, hdbc);
...
res_code = qeQryGetFileNameBuf (hqry,file_name)    ;
...
res_code = qeQryFree (hqry)   ;
res_code = qeDisconnect (hdbc)   ;
```

See Also

qeQrySetFileName.

qeQryGetFileOffset

qeQryGetFileOffset returns the offset of the extra information that is associated with the query represented by *hqry*. This information is everything in the query file except the query.

Syntax

```
int32 file_offset qeQryGetFileOffset (int16   hqry)
```

Parameters

file_offset is an integer that represents the position of the first byte after the SQL statement in the query file.

hqry is a handle to a query obtained from qeQryAllocate or qeQryOpenQueryFile.

Example

```
hdbc = qeConnect ("DSN=QEDBF")   ;
hqry = qeQryOpenQueryFile ("query1.qef");
res_code = qeQrySetHdbc (hqry, hdbc);
file_offset = qeQryGetFileOffset (hqry)    ;
if (file_offset == -1  )
    printf ("There is no extra information")    ;
...
res_code = qeQryFree (hqry)   ;
res_code = qeDisconnect (hdbc)   ;
```

qeQryGetHdbc

qeQryGetHdbc returns the *hdbc* associated with the query represented by *hqry*.

Syntax

```
int16 hdbc qeQryGetHdbc (int16  hqry)
```

Parameters

hqry is a handle to a query obtained from qeQryAllocate or qeQryOpenQueryFile.

hdbc is the handle to a database connection returned by qeQryGetHdbc.

Example

```
hdbc = qeConnect ("DSN=QEDBF")   ;
hqry = qeQryAllocate (hdbc,"");
...
hdbc_val = qeQryGetHdbc (hqry)   ;
...
res_code = qeQryFree (hqry)   ;
res_code = qeDisconnect (hdbc)   ;
```

qeQryGetNumParams

qeQryGetNumParams returns the number of parameters in the query represented by *hqry*.

Syntax

```
int16 num_params qeQryGetNumParams (int16    hqry)
```

Parameters

num_params is the number of parameters returned by qeQryGetNumParams.

hqry is a handle to a query obtained from qeQryAllocate or qeQryOpenQueryFile.

Example

```
hdbc = qeConnect ("DSN=QEDBF")   ;
hqry = qeQryOpenQueryFile ("query1.qef");
res_code = qeQrySetHdbc (hqry, hdbc);
...
num_params = qeQryGetNumParams (hqry)    ;
if (num_params != 0)
{
    res_code = qeQrySetNumParams (hqry,1    )
    /* Code to set the parameter name, *    /
    /* prompt, format, default, and type *    /
...
}
...
res_code = qeQrySaveQueryFile (hqry,"query2.qef")     ;
res_code = qeQryFree (hqry)   ;
res_code = qeDisconnect (hdbc)   ;
```

See Also

qeQrySetNumParams.

qeQryGetParamDefault and qeQryGetParamDefaultBuf

These functions return the default value of a parameter associated with the specified query.

Syntax

```
ptrstr param_default qeQryGetParamDefault   (
    int16    hqry,
    int16    param_num)

int16 res_code qeQryGetParamDefaultBuf   (
    int16    hqry,
    int16    param_num,
    ptrstr   param_default)
```

Description

qeQryGetParamDefault and qeQryGetParamDefaultBuf return the default value of the *param_num*th parameter associated with the query represented in *hqry*. This value is used for the parameter if the user does not provide one, and is represented as a character string.

These functions return an error if you specify a *param_num* value greater than the value returned by qeQryGetNumParams.

qeQryGetParamDefault returns a pointer to the default value string. This string is stored in a buffer maintained by DTK. You must copy the string out of this buffer before you call another DTK function, because the next function may use the same buffer.

With qeQryGetParamDefaultBuf, you pass in a pointer to a buffer you have allocated. The default value string is put in the buffer. You must make sure that the buffer is large enough to hold the returned string.

Parameters

hqry is a handle to a query obtained from qeQryAllocate or qeQryOpenQueryFile.

param_num is the parameter number whose default is to be returned. The first parameter number is 1.

param_default points to a buffer to hold the returned parameter default value.

res_code is the result code returned by qeQryGetParamDefaultBuf, which returns the same set of result codes as qeErr. See Appendix D, "Result and Error Message Codes," on page 537 for a list of these result codes.

Example

```
hdbc = qeConnect ("DSN=QEDBF")   ;
hqry = qeQryOpenQueryFile ("query1.qef");
res_code = qeQrySetHdbc (hqry, hdbc);
...
param_default = qeQryGetParamDefault (hqry, 1)    ;
if (param_default == "20000"   )
    res_code = qeQrySetParamDefault (hqry, 1, "22000")    ;
    ...
res_code = qeQrySaveQueryFile (hqry, "query2.qef")    ;
res_code = qeQryFree (hqry)   ;
res_code = qeDisconnect (hdbc)   ;
```

See Also

qeQrySetParamDefault.

qeQryGetParamFormat and qeQryGetParamFormatBuf

These functions return the format string to be applied to the value of a parameter associated with the specified query.

Syntax

```
ptrstr  param_fmt qeQryGetParamFormat   (
    int16    hqry,
    int16    param_num)

int16  res_code qeQryGetParamFormatBuf   (
    int16    hqry,
    int16    param_num,
    ptrstr   param_fmt)
```

Description

qeQryGetParamFormat and qeQryGetParamFormatBuf return the format string to be applied to the value of the *param_num*th parameter associated with the query represented in *hqry*.

These functions return an error if you specify a *param_num* value greater than the value returned by qeQryGetNumParams.

qeQryGetParamFormat returns a pointer to the format string. This string is stored in a buffer maintained by DTK. You must copy the string out of this buffer before you call another DTK function, because the next function may use the same buffer.

With qeQryGetParamFormatBuf, you pass in a pointer to a buffer you have allocated. The format string is put in the buffer. You must make sure that the buffer is large enough to hold the returned string.

Parameters

hqry is a handle to a query obtained from qeQryAllocate or qeQryOpenQueryFile.

param_num is the parameter number whose format string is to be returned. The first parameter number is 1.

param_fmt points to a buffer to hold the returned parameter format string.

res_code is the result code returned by qeQryGetParamFormatBuf, which returns the same set of result codes as qeErr. See Appendix D, “Result and Error Message Codes,” on page 537 for a list of these result codes.

Example

```
hdbc = qeConnect ("DSN=QEDBF")   ;
hqry = qeQryOpenQueryFile ("query1.qef")    ;
res_code = qeQrySetHdbc (hqry, hdbc);
...
param_fmt = qeQryGetParamFormat (hqry, 2)    ;
...
res_code = qeQrySaveQueryFile (hqry, "query2.qef")     ;
res_code = qeQryFree (hqry)   ;
res_code = qeDisconnect (hdbc)   ;
```

See Also

qeQrySetParamDefault.

qeQryGetParamName and qeQryGetParamNameBuf

These functions return the name of a parameter associated with the specified query.

Syntax

```
ptrstr param_name qeQryGetParamName   (
    int16    hqry,
    int16    param_num)

int16 res_code qeQryGetParamNameBuf   (
    int16    hqry,
    int16    param_num,
    ptrstr   param_name)
```

Description

qeQryGetParamName and qeQryGetParamNameBuf return the parameter name of the *param_num*th parameter associated with the query represented in *hqry*.

These functions return an error if you specify a *param_num* value greater than the value returned by qeQryGetNumParams.

qeQryGetParamName returns a pointer to the parameter name string. This string is stored in a buffer maintained by DTK. You must copy the string out of this buffer before you call another DTK function, because the next function may use the same buffer.

With qeQryGetParamNameBuf, you pass in a pointer to a buffer you have allocated. The parameter name string is put in the buffer. You must make sure that the buffer is large enough to hold the returned string.

Parameters

hqry is a handle to a query obtained from qeQryAllocate or qeQryOpenQueryFile.

param_num is the parameter number whose name is to be returned. The first parameter number is 1.

param_name points to a buffer to hold the returned parameter name.

res_code is the result code returned by qeQryGetParamNameBuf, which returns the same set of result codes as qeErr. See Appendix D, “Result and Error Message Codes,” on page 537 for a list of these result codes.

Example

```
hdbc = qeConnect ("DSN=QEDBF")   ;
hqry = qeQryOpenQueryFile ("query1.qef")    ;
res_code = qeQrySetHdbc (hqry, hdbc);
...
param_name = qeQryGetParamName (hqry,1)    ;
/* If the parameter name <> "SALARY", then set it. *     /
if (strcmp (param_name,"SALARY") != 0    )
    res_code = qeQrySetParamName (hqry,1,"SALARY")    ;
    ...
res_code = qeQrySaveQueryFile (hqry,"query2.qef")    ;
res_code = qeQryFree (hqry)   ;
res_code = qeDisconnect (hdbc)   ;
```

See Also

qeQrySetParamName.

qeQryGetParamPrompt and qeQryGetParamPromptBuf

These functions return the prompt for a parameter associated with the specified query.

Syntax

```
ptrstr  param_prompt qeQryGetParamPrompt   (
    int16    hqry,
    int16    param_num)

int16  res_code qeQryGetParamPromptBuf   (
    int16    hqry,
    int16    param_num,
    ptrstr   param_prompt)
```

Description

qeQryGetParamPrompt and qeQryGetParamPromptBuf return the prompt for the *param_num*th parameter associated with the query represented in *hqry*. This is the text that appears in the dialog box when the user is prompted to enter a value for the parameter.

These functions return an error if you specify a *param_num* value greater than the value returned by qeQryGetNumParams.

qeQryGetParamPrompt returns a pointer to the parameter prompt string. This string is stored in a buffer maintained by DTK. You must copy the string out of this buffer before you call another DTK function, because the next function may use the same buffer.

With qeQryGetParamPromptBuf, you pass in a pointer to a buffer you have allocated. The parameter prompt string is put in the buffer. You must make sure that the buffer is large enough to hold the returned string.

Parameters

hqry is a handle to a query obtained from qeQryAllocate or qeQryOpenQueryFile.

param_num is the parameter number whose prompt is to be returned. The first parameter number is 1.

param_prompt points to a buffer to hold the returned parameter prompt.

res_code is the result code returned by qeQryGetParamPromptBuf, which returns the same set of result codes as qeErr. See Appendix D, "Result and Error Message Codes," on page 537 for a list of these result codes.

Example

```
hdbc = qeConnect ("DSN=QEDBF")   ;
hqry = qeQryOpenQueryFile ("query1.qef")    ;
res_code = qeQrySetHdbc (hqry, hdbc);
...
param_prompt = qeQryGetParamPrompt (hqry,2)    ;
...
res_code = qeQryFree (hqry)   ;
res_code = qeDisconnect (hdbc)   ;
```

See Also

qeQrySetParamPrompt.

qeQryGetParamType

qeQryGetParamType returns the parameter type associated with the specified query.

Syntax

```
int16 param_type qeQryGetParamType   (
    int16    hqry,
    int16    param_num)
```

Description

qeQryGetParamType returns the parameter type of the *param_num*th parameter associated with the query represented in *hqry*.

This function returns an error if you specify a *param_num* value greater than the value returned by qeQryGetNumParams.

Parameters

hqry is a handle to a query obtained from qeQryAllocate or qeQryOpenQueryFile.

param_num is the parameter number whose type is to be returned. The first parameter number is 1.

param_type is the parameter type returned by qeQryGetParamType. It can have the following values:

- Char
- Numeric
- Date
- Time
- Date-time
- Logical

Example

```
hdbc = qeConnect ("DSN=QEDBF")   ;
hqry = qeQryOpenQueryFile ("query1.qef")    ;
res_code = qeQrySetHdbc (hqry, hdbc);
...
num_params = qeQryGetNumParams (hqry)    ;
if (num_params >= 1  )
{
    for (i=1; i <= num_params; ++i   )
    {
             param_type = qeQryGetParamType (hqry,i)    ;
             /* if param_type is Date or Time *    /
             /* then set to Date-Time *   /
             if (param_type == qeQRYPARM_DATE ||
             param_type == qeQRYPARM_TIME   )
             res_code = qeQrySetParamType (hqry,i    ,
             qeQRYPARM_DATETIME  )
    }
}
...
res_code = qeQrySaveQueryFile (hqry,"query2.qef")     ;
res_code = qeQryFree (hqry)   ;
res_code = qeDisconnect (hdbc)   ;
```

See Also

qeQrySetParamType.

qeQryGetSource and qeQryGetSourceBuf

These functions return the data source name used in the query file.

Syntax

```
ptrstr source_name qeQryGetSource (int16   hqry)
```

```
int16 res_code qeQryGetSourceBuf   (
    int16     hqry,
    ptrstr   source_name)
```

Description

qeQryGetSource returns a pointer to the data source name string. This string is stored in a buffer maintained by DTK. You must copy the string out of this buffer before you call another DTK function, because the next function may use the same buffer.

With qeQryGetSourceBuf, you pass in a pointer to a buffer you have allocated. The source name string is put in the buffer. You must make sure that the buffer is large enough to hold the returned string.

Calling qeQrySetHdbc to set the *hdbc* changes the source name specified in the query file to the one used when the *hdbc* was created. This new source name is the one returned by qeQryGetSource if it is called after qeQrySetHdbc.

Parameters

hqry is a query handle obtained from qeQryAllocate or qeQryOpenQueryFile.

source_name points to a buffer to hold the returned data source name string.

res_code is the result code returned by qeQryGetSourceBuf, which returns the same set of result codes as qeErr. See Appendix D, “Result and Error Message Codes,” on page 537 for a list of these result codes.

Example

```
hqry = qeQryOpenQueryFile ("query1.qef")    ;
hdbc = qeConnect ("DSN=QEDBF")   ;
res_code = qeQrySetHdbc (hqry, hdbc)    ;
...
source_name = qeQryGetSource (hqry)    ;
...
res_code = qeQryFree (hqry)   ;
res_code = qeDisconnect (hdbc)   ;
```

See Also qeQrySetStmt.

qeQryGetStmt and qeQryGetStmtBuf

These functions return the statement associated with the query represented by *hqry*.

Syntax

```
ptrstr  stmt qeQryGetStmt (int16   hqry)
```

```
int16  res_code qeQryGetStmtBuf (int16   hqry, ptrstr  stmt)
```

Description

qeQryGetStmt returns a pointer to the statement string. This string is stored in a buffer maintained by DTK. You must copy the string out of this buffer before you call another DTK function, because the next function may use the same buffer.

With qeQryGetStmtBuf, you pass in a pointer to a buffer you have allocated. The statement string is put in the buffer. You must make sure that the buffer is large enough to hold the returned string.

Parameters

hqry is a handle to a query obtained from qeQryAllocate or qeQryOpenQueryFile.

stmt points to a buffer to hold the returned statement.

res_code is the result code returned by qeQryGetStmtBuf, which returns the same set of result codes as qeErr. See Appendix D, “Result and Error Message Codes,” on page 537 for a list of these result codes.

Example

```
hdbc = qeConnect ("DSN=QEDBF")   ;
hqry = qeQryOpenQueryFile ("query1")    ;
res_code = qeQrySetHdbc (hqry, hdbc);
...
stmt = qeQryGetStmt (hqry)   ;
...
res_code = qeQryFree (hqry)   ;
res_code = qeDisconnect (hdbc)   ;
```

See Also

qeQrySetStmt.

qeQryOpenQueryFile

qeQryOpenQueryFile builds a handle to a query based on the contents of the query file.

Syntax

```
int16 hqry qeQryOpenQueryFile (ptrstr    pathname)
```

Description

qeQryOpenQueryFile reads a query file and builds a handle to a query based on the contents of that file.

The contents of the query file are made available via a series of functions that access the parts of the query file.

Parameters

hqry is the handle to a query returned by the function. Its value is 0 if the file could not be opened and converted to an *hqry*.

pathname points to a string which holds a pathname to the query file.

Example

```
hdbc = qeConnect ("DSN=QEDBF")   ;
hqry = qeQryOpenQueryFile ("query1")    ;
res_code = qeQrySetHdbc (hqry, hdbc);
...
res_code = qeQryFree (hqry)   ;
res_code = qeDisconnect (hdbc)   ;
```

See Also

qeQryBuilder, qeQryAllocate, qeQrySaveQueryFile.

qeQryPrepare

qeQryPrepare prepares a SQL statement, represented by a handle to a query, for execution.

Syntax

```
int16 hstmt qeQryPrepare (int16  hqry)
```

Description

qeQryPrepare prepares the SQL statement represented by *hqry* for execution.

The statement must subsequently be executed using qeSQLExecute.

Parameters

hqry is a handle to a query obtained from qeQryAllocate or qeQryOpenQueryFile.

hstmt is the handle to the statement returned by the function.

Example

```
hdbc = qeConnect ("DSN=QEDBF")   ;
hqry = qeQryAllocate (hdbc, "SELECT * FROM EMP")     ;
hstmt = qeQryPrepare (hqry)   ;
res_code = qeSQLExecute (hstmt)   ;
...
res_code = qeEndSQL (hstmt)   ;
...
res_code = qeQryFree (hqry)   ;
res_code = qeDisconnect (hdbc)   ;
```

See Also

qeSQLExecute.

qeQrySaveQueryFile

qeQrySaveQueryFile writes a query to a query (.QEF) file.

Syntax

```
int16 res_code qeQrySaveQueryFile (int16   hqry, ptrstr
pathname)
```

Description

qeQrySaveQueryFile writes the query associated with the *hqry* as a query (.QEF) file. If *pathname* is null, then *hqry* must have a name for the file associated with it.

If the query was read from a query file initially, the contents of the file that do not correspond to the query or its parameters are preserved.

Parameters

hqry is a handle to a query.

pathname points to a string which holds a pathname for the query file to be written. If null, the pathname is obtained from the *hqry*.

res_code is the result code returned by qeQrySaveQueryFile, which returns the same set of result codes as qeErr. See Appendix D, “Result and Error Message Codes,” on page 537 for a list of these result codes.

Example

```
hdbc = qeConnect ("DRV=QEDBF")   ;
hqry = qeQryOpenQueryFile ("query1.qef");
res_code = qeQrySetHdbc (hqry, hdbc);
...
res_code = qeQrySaveQueryFile (hqry, "newquery.qef")     ;
res_code = qeQryFree (hqry)   ;
res_code = qeDisconnect (hdbc)   ;
```

See Also

qeQryOpenQueryFile, qeQryAllocate, qeQryBuilder.

qeQrySetFileName

qeQrySetFileName sets the file name of a query (.QEF) file associated with *hqry*.

Syntax

```
int16 res_code qeQrySetFileName (int16  hqry, ptrstr
file_name)
```

Parameters

hqry is a handle to a query obtained from qeQryAllocate or qeQryOpenQueryFile.

file_name points to a string with the new file name.

res_code is the result code returned by qeQrySetFileName, which returns the same set of result codes as qeErr. See Appendix D, “Result and Error Message Codes,” on page 537 for a list of these result codes.

Example

```
hdbc = qeConnect ("DRV=QEDBF")   ;
hqry = qeQryOpenQueryFile ("query1.qef");
res_code = qeQrySetHdbc (hqry, hdbc);
res_code = qeQryBuilder (hqry, 0,
qeQRY_TABLES,qeQRY_DEFAULT)   ;
...
res_code = qeQrySetFileName (hqry, "qry.qef")    ;
res_code = qeQrySaveQueryFile (hqry,"")    ;
res_code = qeQryFree (hqry)   ;
res_code = qeDisconnect (hdbc)   ;
```

See Also

qeQryGetFileName and qeQryGetFileNameBuf.

qeQrySetHdbc

qeQrySetHdbc sets the handle to the connection for the query represented by *hqry*.

Syntax

```
int16 res_code qeQrySetHdbc (int16  hqry, int16 hdbc)
```

Description

qeQrySetHdbc sets the handle to the database connection for the query represented by *hqry*.

Calling this function to set the *hdbc* changes the source name specified in the query file to the one used when the connection was created. This new source name is the one returned by qeQrySetSource, and is written in the header of the query file created by qeQrySaveQueryFile.

Parameters

hqry is a handle to a query obtained from qeQryOpenQueryFile.

hdbc is a handle to a database connection returned by qeConnect.

res_code is the result code returned by qeQrySetHdbc, which returns the same set of result codes as qeErr. See Appendix D, “Result and Error Message Codes,” on page 537 for a list of these result codes.

Example

```
hdbc = qeConnect ("DSN=QEDBF")   ;
hqry = qeQryOpenQueryFile ("query1")    ;
res_code = qeQrySetHdbc (hqry, hdbc);
...
param_name = qeQryGetParamName (hqry, 1)    ;
if (strcmp (param_name, "SALARY") != 0    )
    res_code = qeQrySetParamName (hqry, 1, "SALARY")     ;
    ...
res_code = qeQrySaveQueryFile (hqry, "query2")     ;
res_code = qeQryFree (hqry)   ;
res_code = qeDisconnect (hdbc)   ;
```

See Also

qeQryGetHdbc.

qeQrySetNumParams

qeQrySetNumParams sets the number of parameters associated with the query represented by *hqry*.

Syntax

```
int16 res_code qeQrySetNumParams (int16   hqry, int16
num_params)
```

Parameters

hqry is a handle to a query obtained from qeQryAllocate or qeQryOpenQueryFile.

num_params is the new number of parameters to be associated with the query represented by *hqry*. If you increase the number of parameters, the new parameters default to character type.

res_code is the result code returned by qeQrySetNumParams, which returns the same set of result codes as qeErr. See Appendix D, “Result and Error Message Codes,” on page 537 for a list of these result codes.

Example

```
hdbc = qeConnect ("DSN=QEDBF")   ;
hqry = qeQryOpenQueryFile ("query1.qef");
res_code = qeQrySetHdbc (hqry, hdbc);
...
num_params = qeQryGetNumParams (hqry)    ;
if (num_params == 0)
{
    res_code = qeQrySetNumParams (hqry,  1    )
    /* code to set the Parameter name, *    /
    /* prompt, format, default, and type *    /
...
}
...
res_code = qeQrySaveQueryFile (hqry, "query2.qef")     ;
res_code = qeQryFree (hqry)   ;
res_code = qeDisconnect (hdbc)   ;
```

See Also

qeQryGetNumParams.

qeQrySetParamDefault

qeQrySetParamDefault sets the default value of a parameter associated with the specified query.

Syntax

```
int16 res_code qeQrySetParamDefault   (
    int16    hqry,
    int16    param_num,
    ptrstr   param_default)
```

Description

qeQrySetParamDefault sets the default parameter value of the *param_num*th parameter associated with the query represented by *hqry*.

This function returns an error if you specify a *param_num* value greater than the value returned by qeQryGetNumParams.

Parameters

hqry is a query handle obtained from qeQryAllocate or qeQryOpenQueryFile.

param_num is the parameter number for which a default value is to be set.The first parameter number is 1.

param_default points to a string that is the new parameter default value.

res_code is the result code returned by qeQrySetParamDefault, which returns the same set of result codes as qeErr. See Appendix D, “Result and Error Message Codes,” on page 537 for a list of these result codes.

Example

```
hdbc = qeConnect ("DSN=QEDBF")   ;
hqry = qeQryOpenQueryFile ("query1")    ;
res_code = qeQrySetHdbc (hqry, hdbc)    ;
...
param_default = qeQryGetParamDefault (hqry, 1)    ;
if (param_default == "20000"   )
    res_code = qeQrySetParamDefault (hqry, 1, "22000")     ;
    ...
res_code = qeQrySaveQueryFile (hqry, "query2")    ;
res_code = qeQryFree (hqry)   ;
res_code = qeDisconnect (hdbc)   ;
```

See Also

qeQryGetParamDefault and qeQryGetParamDefaultBuf.

qeQrySetParamFormat

qeQrySetParamFormat sets the format string for a parameter associated with the specified query.

Syntax

```
int16 res_code qeQrySetParamFormat   (
    int16    hqry,
    int16    param_num,
    ptrstr   param_fmt)
```

Description

qeQrySetParamFormat sets the parameter format string to be applied to the *param_num*th parameter associated with the query represented by *hqry*.

This function returns an error if you specify a *param_num* value greater than the value returned by qeQryGetNumParams.

Parameters

hqry is a handle to a query obtained from qeQryAllocate or qeQryOpenQueryFile.

param_num is the parameter number for which a format string is to be set. The first parameter number is 1.

param_fmt points to a string that is the new parameter format string.

res_code is the result code returned by qeQrySetParamFormat, which returns the same set of result codes as qeErr. See Appendix D, “Result and Error Message Codes,” on page 537 for a list of these result codes.

Example

```
hdbc = qeConnect ("DRV=QEDBF")   ;
hqry = qeQryOpenQueryFile ("query1.qef");
res_code = qeQrySetHdbc (hqry, hdbc)   ;
...
num_params = qeQryGetNumParams (hqry)   ;
for (i=1; i<=num_params; ++i   )
{
    param_type = qeQryGetParamType (hqry, i)   ;
    /* If the parameter type is Date *   /
    if (param_type == 3  )
             res_code = qeQrySetParamFormat (hqry, i, "m/
d/yy") ;
}
...
res_code = qeQrySaveQueryFile (hqry, "query2.qef")    ;
res_code = qeQryFree (hqry)   ;
res_code = qeDisconnect (hdbc)   ;
```

See Also

qeQryGetParamFormat and qeQryGetParamFormatBuf.

qeQrySetParamName

qeQrySetParamName sets the name of a parameter associated with the specified query.

Syntax

```
int16 res_code qeQrySetParamName   (
    int16    hqry,
    int16    param_num,
    ptrstr   param_name)
```

Description

qeQrySetParamName sets the parameter name of the *param_num*th parameter associated with the query represented by *hqry*.

This function returns an error if you specify a *param_num* value greater than the value returned by qeQryGetNumParams.

Parameters

hqry is a handle to a query obtained from qeQryAllocate or qeQryOpenQueryFile.

param_num is the number of the parameter for which the name is to be set. The first parameter number is 1.

param_name points to a string that is the new parameter name.

res_code is the result code returned by qeQrySetParamName, which returns the same set of result codes as qeErr. See Appendix D, "Result and Error Message Codes," on page 537 for a list of these result codes.

Example

```
hdbc = qeConnect ("DSN=QEDBF")   ;
hqry = qeQryOpenQueryFile ("query1")    ;
res_code = qeQrySetHdbc (hqry, hdbc)    ;
...
param_name = qeQryGetParamName (hqry, 1)    ;
if (strcmp (param_name, "SALARY") != 0    )
    res_code = qeQrySetParamName (hqry, 1, "SALARY")     ;
    ...
res_code = qeQrySaveQueryFile (hqry, "query2")     ;
res_code = qeQryFree (hqry)   ;
res_code = qeDisconnect (hdbc)   ;
```

See Also

qeQryGetParamName and qeQryGetParamNameBuf.

qeQrySetParamPrompt

qeQrySetParamPrompt sets the prompt for a parameter associated with the specified query.

Syntax

```
int16  res_code qeQrySetParamPrompt   (
    int16    hqry,
    int16    param_num,
    ptrstr   param_prompt)
```

Description

qeQrySetParamPrompt sets the parameter prompt of the *param_num*th parameter associated with the query represented by *hqry*.

This function returns an error if you specify a *param_num* value greater than the value returned by qeQryGetNumParams.

Parameters

hqry is a handle to a query obtained from qeQryAllocate or qeQryOpenQueryFile.

param_num is the number of the parameter for which a prompt is to be set. The first parameter number is 1.

param_prompt points to a string that is the new parameter prompt.

res_code is the result code returned by qeQrySetParamPrompt, which returns the same set of result codes as qeErr. See Appendix D, “Result and Error Message Codes,” on page 537 for a list of these result codes.

Example

```
hdbc = qeConnect ("DSN=QEDBF")   ;
hqry = qeQryOpenQueryFile ("query1");
res_code = qeQrySetHdbc (hqry, hdbc)   ;
...
res_code = qeQrySetParamPrompt (hqry, 1, "Salary")    ;
...
res_code = qeQrySaveQueryFile (hqry, "query2")    ;
res_code = qeQryFree (hqry)   ;
res_code = qeDisconnect (hdbc)   ;
```

See Also

qeQryGetParamPrompt and qeQryGetParamPromptBuf.

qeQrySetParamType

qeQrySetParamType sets the data type of a parameter associated with the specified query.

Syntax

```
int16 res_code qeQrySetParamType   (
    int16    hqry,
    int16    param_num,
    int16    param_type)
```

Description

qeQrySetParamType sets the data type of the *param_num*th parameter associated with the query represented by *hqry*.

This function returns an error if you specify a *param_num* value greater than the value returned by qeQryGetNumParams.

Parameters

hqry is a handle to a query obtained from qeQryAllocate or qeQryOpenQueryFile.

param_num is the number of the parameter for which a type is to be set. The first parameter number is 1.

param_type is the new data type. It can have the following values:

- Char
- Numeric
- Date
- Time
- Date-time
- Logical

res_code is the result code returned by qeQrySetParamType, which returns the same set of result codes as qeErr. See Appendix D, "Result and Error Message Codes," on page 537 for a list of these result codes.

Example

```
hdbc = qeConnect ("DSN=QEDBF")   ;
hqry = qeQryOpenQueryFile ("query1.qef")    ;
res_code = qeQrySetHdbc (hqry, hdbc)    ;
...
num_params = qeQryGetNumParams (hqry)    ;
if (num_params >= 1  )
{
    for (i=1; i <= num_params; ++i   )
    {
             param_type = qeQryGetParamType (hqry, i)    ;
             /* if param_type is Date or Time *    /
             /* then set to Date-Time *   /
             if (param_type == qeQRYPARM_DATE ||
             param_type == qeQRYPARM_TIME   )
             res_code = qeQrySetParamType (hqry, i    ,
             qeQRYPARM_DATETIME  )
    }
}
...
res_code = qeQrySaveQueryFile (hqry, "query2.qef")     ;
res_code = qeQryFree (hqry)   ;
res_code = qeDisconnect (hdbc)   ;
```

See Also

qeQryGetParamType.

qeQrySetStmt

qeQrySetStmt sets the statement associated with the query represented by *hqry*.

Syntax

```
int16 res_code qeQrySetStmt (int16  hqry, ptrstr  stmt)
```

Parameters

hqry is a handle to a query obtained from qeQryAllocate or qeQryOpenQueryFile.

stmt is a pointer to a variable containing the text of the statement to be set.

res_code is the result code returned by qeQrySetStmt, which returns the same set of result codes as qeErr. See Appendix D, “Result and Error Message Codes,” on page 537 for a list of these result codes.

Example

```
hdbc = qeConnect ("DSN=QEDBF")   ;
hqry = qeQryOpenQueryFile ("query1.qef")    ;
res_code = qeQrySetHdbc (hqry, hdbc)   ;
...
res_code = qeQrySetStmt (hqry, "SELECT * FROM emp.dbf")     ;
...
res_code = qeQrySaveQueryFile (hqry, "query2.qef")    ;
res_code = qeQryFree (hqry)   ;
res_code = qeDisconnect (hdbc)   ;
```

See Also

qeQryGetStmt and qeQryGetStmtBuf.

qeQrySetSource

qeQrySetSource sets the data source for the query represented by *hqry*.

Syntax

`int16 res_code qeQrySetSource (int16 hqry, ptrstr source)`

Parameters

hqry is a handle to a query obtained from qeQryAllocate or qeQryOpenQueryFile.

source is a new data source for the query that will be saved in the query file.

res_code is the result code returned by qeQrySetSource, which returns the same set of result codes as qeErr. See Appendix D, “Result and Error Message Codes,” on page 537 for a list of these result codes.

Example

```
hdbc = qeConnect ("DSN=QESS")   ;
hqry = qeQryOpenQueryFile ("query1.qef")    ;
res_code = qeQrySetSource (hqry, "QESS")    ;
res_code = qeQrySaveQueryFile ("query2.qef")    ;
res_code = qeQryFree (hqry)   ;
res_code = qeDisconnect (hdbc)   ;
```

Notes

Calling qeQrySetHdbc causes DTK to reset the source name to that used by the query file.

See Also

qeQryGetSource and qeQryGetSourceBuf.

qeRecClearConditions

qeRecClearConditions clears a statement's search conditions.

Syntax

```
int16 res_code qeRecClearConditions (int16   hstmt)
```

Description

qeRecClearConditions clears all search conditions associated with a statement.

This call is necessary only if search conditions have been previously set for a statement with qeRecSetCondition functions. Newly created statements have no search conditions.

Parameters

hstmt is the handle to the statement returned by qeExecSQL or qeSQLPrepare.

res_code is the result code returned by qeRecClearConditions, which returns the same set of result codes as qeErr. See Appendix D, "Result and Error Message Codes," on page 537 for a list of these result codes.

Example

```
hdbc = qeConnect ("DSN=QEDBF")   ;
hstmt = qeExecSQL (hdbc, "SELECT * FROM emp")    ;
res_code = qeRecSetConditionChar (hstmt, 1,
    qeFIND_EQUAL, "Tyler", "", FALSE)   ;
new_hstmt = qeQBEPrepare (hstmt)   ;
res_code = qeSQLExecute (new_hstmt)   ;
while (qeFetchNext (new_hstmt) == qeSUCCESS    )
    ...
/* Get values matching condition. *   /
    ...
res_code = qeEndSQL (new_hstmt)   ;

res_code = qeRecClearConditions (hstmt)    ;
res_code = qeRecSetConditionChar (hstmt, 1,
    qeFIND_NOT_EQUAL, "Tyler", "", FALSE)   ;
new_hstmt = qeQBEPrepare (hstmt)   ;
res_code = qeSQLExecute (new_hstmt)   ;
while (qeFetchNext (new_hstmt) == qeSUCCESS    )
    ...

/* Get values matching condition. *   /
    ...
res_code = qeEndSQL (new_hstmt)   ;
res_code = qeEndSQL (hstmt)   ;
res_code = qeDisconnect (hdbc)   ;
```

qeRecDelete

qeRecDelete deletes the current record.

Syntax

```
int16 res_code qeRecDelete (int16   hstmt)
```

Description

When you call qeRecDelete, DTK removes the current record from the buffer. The next record fills the position of the deleted record, all subsequent records advance by one, and the total number of records in the buffer decreases by one. If the buffer contains 10 records and the *hstmt* is positioned on record 2, then a call to qeRecDelete deletes record 2, record 3 becomes record 2, 4 becomes 3, etc., and the total count for the buffer becomes 9.

When qeRecDelete is invoked during a transaction, record deletions are either written to the database by a call to qeCommit or aborted by a call to qeRollback. Otherwise, deletions resulting from calls to qeRecDelete are made instantly to the database.

After a record is deleted, the current record is positioned between the previous record and the next record in the buffer. You must call qeFetchNext after deleting a record to position on the next record.

You can call qeNumModRecs to determine the number of records deleted by a call to qeRecDelete.

Calling this function causes DTK to generate a unique key if you have not already defined one with qeRecSetKey.

qeRecDelete cannot delete records from joined tables.

Parameters

hstmt is the handle to the statement returned by qeExecSQL or qeSQLPrepare.

res_code is the result code returned by qeRecDelete, which returns the same set of result codes as qeErr. See Appendix D, “Result and Error Message Codes,” on page 537 for a list of these result codes.

Example

```
hdbc = qeConnect ("DSN=QEDBF")   ;
hstmt = qeExecSQL (hdbc, "SELECT * FROM emp")    ;
res_code = qeFetchNext (hstmt)   ;
res_code = qeRecDelete (hstmt)   ;
res_code = qeEndSQL (hstmt)   ;
res_code = qeDisconnect (hdbc)   ;
```

Notes

If you call qeRecDelete without having previously called a qeFetch function to position the *hstmt* on a record, DTK returns an error. For example, if you call qeExecSQL and then immediately call qeRecDelete on the new *hstmt*, DTK cannot delete a record because the *hstmt* is still on record 0 (no record). In order to delete a record, you must first call qeFetchNext to position the *hstmt* on the first record in the buffer (record 1).

Important: To delete the current record, qeRecDelete generates a SQL Delete statement that uses a Where clause to uniquely identify that record. If this Where clause matches multiple records, qeRecDelete deletes all matching records. You can recover from such invalid deletions by using transactions and calling qeNumModRecs after each call to qeRecDelete to verify that multiple records were not deleted. Calling qeRecLock before calls to qeRecDelete also helps prevent multiple deletions, since qeRecLock uses the same Where clause as qeRecDelete and returns a warning if it locks multiple records.

See Also

qeRecSetKey.

qeRecFind

qeRecFind positions to the next row matching the qeRecSetCondition search criteria.

Syntax

```
int32 result qeRecFind (int16  hstmt, int16 start_pos,
    int16 flags)
```

Description

qeRecFind attempts to find the next row matching the search criteria specified by calls to the qeRecSetCondition functions.

If a matching row is found, it becomes the current position in the result set. If not, the position is unchanged.

You can use qeRecFind along with the qeBindCol or qeVal functions to retrieve the set of records that match the qeRecSetCondition search criteria.

Parameters

result is the number of the row matching the search conditions. It is 0 if no row was found.

hstmt is the handle to the statement returned by qeExecSQL or qeSQLPrepare.

start_pos is the starting position for the search. There is no default; you must specify one of the following values:

Constant	Value	Description
qeFIND_BEGIN	1	Start at the beginning of the result set
qeFIND_END	2	Start at the end of the result set
qeFIND_CURRENT	3	Start at the current record of the result set

flags is a set of option flags that controls the way the search is performed:

Constant	Value	Description
qeFIND_BACKWARD	0x0001	The search goes backwards. The default is forward.
qeFIND_SKIP_ROW	0x0002	The search skips the current row if *start_pos* = qeFIND_CURRENT. The default is to start with the current row.

These values can be combined by adding them together or joining them with an OR clause.

Example

```
hdbc = qeConnect ("DSN=QEDBF")   ;
hstmt = qeExecSQL (hdbc, "SELECT * FROM emp")    ;
res_code = qeRecSetConditionChar (hstmt, 1    ,
    qeFIND_EQUAL, "David", "", FALSE)   ;
new_pos = qeRecFind (hstmt, qeFIND_BEGIN, 0)    ;
/* The hstmt is now either on the same record   *    /
/* or on the first occurrence of a record       *    /
/* matching the condition set above.            *    /
res_code = qeEndSQL (hstmt)   ;
res_code = qeDisconnect (hdbc)   ;
```

See Also

qeRecSetCondition functions, qeQBEPrepare.

qeRecGetKey

qeRecGetKey reports whether a column is part of the key used by DTK.

Syntax

```
int16 setting qeRecGetKey (int16  hstmt, int16 col_num)
```

Description

qeRecGetKey returns whether DTK uses the specified column as part of a key.

DTK does not generate a default key until qeRecUpdate, qeRecDelete, qeRecLock, or qeUniqueWhereClause is called for the *hstmt*. Until you call one of these functions (or specify a key by calling qeRecSetKey), *hstmt* will have no key—every column specified in calls to qeRecGetKey returns False (0).

See “Unique Keys” on page 78 for information on DTK’s use of unique keys.

Parameters

setting is True (1) if the column is in the key; otherwise it is False (0).

hstmt is the handle to the statement returned by qeExecSQL or qeSQLPrepare.

col_num is the column number which is to be tested. The first column number is 1.

Example

```
hdbc = qeConnect ("DSN=QEDBF")   ;
hstmt = qeExecSQL (hdbc, "SELECT * FROM emp")     ;
...
/* Check to see if LAST_NAME field is used *    /
/* as part of the primary key. *   /
set_val = qeRecGetKey (hstmt, 2)   ;
...
res_code = qeEndSQL (hstmt)   ;
res_code = qeDisconnect (hdbc)   ;
```

See Also

qeRecSetKey.

qeRecLock

qeRecLock locks the current record during a transaction.

Syntax

```
int16 res_code qeRecLock (int16  hstmt)
```

Description

qeRecLock attempts to lock the current record. It works only if a transaction is currently active. Otherwise, it returns an error.

The lock is freed by a call to qeCommit or qeRollback.

If enabled by options passed to qeSetLockOptions, qeRecLock can compare the record with the log file or refresh the log file.

If 0 records are locked qeRecLock returns an error (qeLOCK_NO_REC (-6)). qeRecLock issues a warning if multiple records are locked (qeLOCK_MULTI_REC (-7)) or the optional log file comparison fails (qeLOCK_CHANGE_REC (-8)). This makes qeRecLock useful for ensuring that only one record is affected by a call to qeRecDelete or qeRecUpdate.

Calling this function causes DTK to generate a unique key if you have not already defined one with qeRecSetKey.

This function has no effect with some databases.

Parameters

hstmt is the handle to the statement returned by qeExecSQL or qeSQLPrepare.

res_code is the result code returned by qeRecLock, which returns the same set of result codes as qeErr. See Appendix D, “Result and Error Message Codes,” on page 537 for a list of these result codes.

Example

```
hdbc = qeConnect ("DSN=QEDBF")   ;
res_code = qeBeginTran (hdbc)   ;
hstmt = qeExecSQL (hdbc, "SELECT * FROM emp")    ;
res_code = qeFetchNext (hstmt)   ;
res_code = qeRecLock (hstmt)   ;
...
res_code = qeEndSQL (hstmt)   ;
res_code = qeCommit (hdbc)   ;
res_code = qeDisconnect (hdbc)   ;
```

See Also

qeSetLockOptions, qeBeginTran, qeCommit, qeRollback.

qeRecNew

qeRecNew creates a buffer for a new record.

Syntax

```
int16 res_code qeRecNew (int16  hstmt, int32  rec_num)
```

Description

qeRecNew creates a buffer to be used for a new record. All column values are initially set to null. The record can then be placed in the buffer by calls to the qePut functions.

To insert the record, call qeRecUpdate. The record is also inserted when the *hstmt* is moved to a different record number, and qeSetAutoUpdate is set to qeAUTOUPD_UPDATE (3).

Parameters

hstmt is the handle to the statement returned by qeExecSQL or qeSQLPrepare.

rec_num is the location where the new record is to be inserted. If random fetching is enabled, *rec_num* can be any number from 1 to the last record fetched plus 1. If random fetching is not enabled, *rec_num* must be the current record number plus 1.

res_code is the result code returned by qeRecNew, which returns the same set of result codes as qeErr. See Appendix D, "Result and Error Message Codes," on page 537 for a list of these result codes.

Example

```
hdbc = qeConnect ("DSN=QEDBF")   ;
res_code = qeBeginTran (hdbc)   ;
hstmt = qeExecSQL (hdbc, "SELECT * FROM emp")    ;
res_code = qeRecNew (hstmt, 1)   ;
res_code = qePutChar (hstmt, 1, "", "Mike")    ;
res_code = qePutChar (hstmt, 2, "", "McGarrah")    ;
res_code = qeRecUpdate (hstmt)   ;
res_code = qeEndSQL (hstmt)   ;
res_code = qeCommit (hdbc)   ;
res_code = qeDisconnect (hdbc)   ;
```

Notes

When auto-updating has not been enabled by qeSetAutoUpdate, if you call qeRecNew and then move off of the current record before calling qeRecUpdate, then the buffer created by the call to qeRecNew is destroyed.

See Also

qeSetAutoUpdate.

qeRecNum

qeRecNum returns the number of the current record in the buffer.

Syntax

```
int32 rec_num qeRecNum (int16 hstmt)
```

Parameters

hstmt is the handle to the statement returned by qeExecSQL or qeSQLPrepare.

rec_num is the number of the current record in the buffer returned for this statement execution. If there is no current record number—that is, when qeRecState returns qeSTATE_NO_REC—then the *hstmt* is positioned between *rec_num* and <*rec_num* + 1>. In this situation, a call to qeFetchPrev returns the *hstmt* to *rec_num*, and a call to qeFetchNext increments *rec_num* by 1.

Example

```
hdbc = qeConnect ("DSN=QEDBF")   ;
hstmt = qeExecSQL (hdbc, "SELECT * FROM emp")    ;
res_code = qeFetchNext (hstmt)   ;
/* Return the record number of the current *    /
/* record in the selected query.           *    /
res_code = qeRecNum (hstmt)   ;
...
res_code = qeEndSQL (hstmt)   ;
res_code = qeDisconnect (hdbc)   ;
```

qeRecSetConditionBinary

qeRecSetConditionBinary adds a search condition to the statement having a binary value to compare.

Syntax

```
int16 res_code qeRecSetConditionBinary    (
    int16    hstmt,
    int16    col_num,
    int16    operator,
    ptrstr   value,
    int32    length)
```

Description

qeRecSetConditionBinary adds a search condition to the statement having a binary value to compare.

For all operators except the IN operator, multiple search conditions for a column are joined with a boolean AND (that is, all conditions must be true for the row to match the search conditions).

For the IN operator, multiple search conditions for a column are joined with a boolean OR (that is, at least one of the IN conditions must be true for the row to match the search conditions).

Parameters

hstmt is the handle to the statement returned by qeExecSQL or qeSQLPrepare.

col_num is the number of the column on which a search condition is being placed. The first column number is 1.

operator is the comparison operator and has one of the following values:

Constant	Value	Operator
qeFIND_LESS_THAN	1	<
qeFIND_LESS_THAN_OR_EQ	2	<=

Constant	Value	Operator
qeFIND_GREATER_ THAN	3	>
qeFIND_GREATER_ THAN_OR_EQ	4	>=
qeFIND_EQUAL	5	=
qeFIND_NOT_EQUAL	6	<>
qeFIND_IN	9	IN

value points to the binary comparison value.

length is the length (in bytes) of the comparison value.

res_code is the result code returned by qeRecSetConditionBinary, which returns the same set of result codes as qeErr. See Appendix D, “Result and Error Message Codes,” on page 537 for a list of these result codes.

Example

```
hdbc = qeConnect ("DSN=QEDBF")   ;
hstmt = qeExecSQL (hdbc, "SELECT * FROM emp")    ;
/* bindata contains the binary value for comparison. *     /
res_code = qeRecSetConditionBinary (hstmt, 8    ,
    qeFIND_NOT_EQUAL, bindata, 10000)   ;
new_hstmt = qeQBEPrepare (hstmt)   ;
res_code = qeSQLExecute (new_hstmt)    ;
while (qeFetchNext (new_hstmt) == qeSUCCESS    )
    ...
/* Get values matching condition. *    /
    ...
res_code = qeEndSQL (new_hstmt)   ;
res_code = qeEndSQL (hstmt)   ;
res_code = qeDisconnect (hdbc)   ;
```

qeRecSetConditionChar

qeRecSetConditionChar adds a search condition to the statement having a character value to compare.

Syntax

```
int16  res_code qeRecSetConditionChar   (
    int16    hstmt,
    int16    col_num,
    int16    operator,
    ptrstr   value,
    ptrstr   fmt_string,
    int16    case_sens)
```

Description

qeRecSetConditionChar adds a search condition to the statement having a character value to compare.

For all operators except the IN operator, multiple search conditions for a column are joined with a boolean AND (that is, all conditions must be true for the row to match the search conditions).

For the IN operator, multiple search conditions for a column are joined with a boolean OR (that is, at least one of the IN conditions must be true for the row to match the search conditions).

Parameters

hstmt is the handle to the statement returned by qeExecSQL or qeSQLPrepare.

col_num is the number of the column on which a search condition is being placed. The first column number is 1.

operator is the comparison operator and has one of the following values:

Constant	Value	Operator
qeFIND_LESS_THAN	1	<
qeFIND_LESS_THAN_OR_EQ	2	<=

Constant	Value	Operator
qeFIND_GREATER_ THAN	3	>
qeFIND_GREATER_ THAN_OR_EQ	4	>=
qeFIND_EQUAL	5	=
qeFIND_NOT_EQUAL	6	<>
qeFIND_LIKE	7	LIKE
qeFIND_NOT_LIKE	8	NOT LIKE
qeFIND_IN	9	IN

value points to the comparison string.

fmt_string is a string used to control formatting of dates and numbers into a character string.

case_sens determines if character comparisons are case-sensitive. Its value must be TRUE for non-character columns.

res_code is the result code returned by qeRecSetConditionChar, which returns the same set of result codes as qeErr. See Appendix D, “Result and Error Message Codes,” on page 537 for a list of these result codes.

Example

```
hdbc = qeConnect ("DSN=QEDBF")   ;
hstmt = qeExecSQL (hdbc, "SELECT * FROM emp")    ;
res_code = qeFetchNext (hstmt)   ;
res_code = qeRecSetConditionChar (hstmt, 2    ,
    qeFIND_LIKE, "Dav%", ""  , FALSE) ;
new_hstmt = qeQBEPrepare (hstmt)   ;
res_code = qeSQLExecute (new_hstmt)   ;
while (qeFetchNext (new_hstmt) == qeSUCCESS    )
    ...
/* Get values matching condition. *   /
    ...
res_code = qeEndSQL (new_hstmt)   ;
res_code = qeEndSQL (hstmt)   ;
res_code = qeDisconnect (hdbc)   ;
```

qeRecSetConditionDecimal

qeRecSetConditionDecimal adds a search condition to the statement having a decimal value to compare.

Syntax

```
int16  res_code qeRecSetConditionDecimal    (
    int16    hstmt,
    int16    col_num,
    int16    operator,
    ptrstr   value,
    int16    precision,
    int16    scale)
```

Description

qeRecSetConditionDecimal adds a search condition to the statement having a decimal value to compare.

For all operators except the IN operator, multiple search conditions for a column are joined with a boolean AND (that is, all conditions must be true for the row to match the search conditions).

For the IN operator, multiple search conditions for a column are joined with a boolean OR (that is, at least one of the IN conditions must be true for the row to match the search conditions).

Parameters

hstmt is the handle to the statement returned by qeExecSQL or qeSQLPrepare.

col_num is the number of the column on which a search condition is being placed. The first column number is 1.

operator is the comparison operator and has one of the following values:

Constant	Value	Operator
qeFIND_LESS_THAN	1	<
qeFIND_LESS_THAN_OR_EQ	2	<=
qeFIND_GREATER_THAN	3	>

Constant	Value	Operator
qeFIND_GREATER_THAN_OR_EQ	4	>=
qeFIND_EQUAL	5	=
qeFIND_NOT_EQUAL	6	<>
qeFIND_IN	9	IN

value points to the decimal comparison value.

precision is the precision of the decimal value.

scale is the scale of the decimal value.

res_code is the result code returned by qeRecSetConditionDecimal, which returns the same set of result codes as qeErr. See Appendix D, “Result and Error Message Codes,” on page 537 for a list of these result codes.

Example

```
hdbc = qeConnect ("DSN=QEDBF")   ;
hstmt = qeExecSQL (hdbc, "SELECT * FROM emp")    ;
res_code = qeRecSetConditionDecimal (hstmt, 5     ,
    qeFIND_GREATER_THAN, dec_val, 8, 2)    ;
new_hstmt = qeQBEPrepare (hstmt)    ;
res_code = qeSQLExecute (new_hstmt)    ;
while (qeFetchNext (new_hstmt) == qeSUCCESS     )
    ...
/* Get values matching condition. *    /
    ...
res_code = qeEndSQL (new_hstmt)    ;
res_code = qeEndSQL (hstmt)   ;
res_code = qeDisconnect (hdbc)   ;
```

qeRecSetConditionDouble

qeRecSetConditionDouble adds a search condition to the statement having a double-precision floating-point value to compare.

Syntax

```
int16  res_code qeRecSetConditionDouble    (
    int16    hstmt,
    int16    col_num,
    int16    operator,
    float64  value)
```

Description

qeRecSetConditionDouble adds a search condition to the statement having a double-precision floating-point value to compare.

For all operators except the IN operator, multiple search conditions for a column are joined with a boolean AND (that is, all conditions must be true for the row to match the search conditions).

For the IN operator, multiple search conditions for a column are joined with a boolean OR (that is, at least one of the IN conditions must be true for the row to match the search conditions).

Parameters

hstmt is the handle to the statement returned by qeExecSQL or qeSQLPrepare.

col_num is the number of the column on which a search condition is being placed. The first column number is 1.

operator is the comparison operator and has one of the following values:

Constant	Value	Operator
qeFIND_LESS_THAN	1	<
qeFIND_LESS_THAN_OR_EQ	2	<=
qeFIND_GREATER_THAN	3	>
qeFIND_GREATER_THAN_OR_EQ	4	>=

Constant	Value	Operator
qeFIND_EQUAL	5	=
qeFIND_NOT_EQUAL	6	<>
qeFIND_IN	9	IN

value points to the double-precision floating-point comparison value.

res_code is the result code returned by qeRecSetConditionDouble, which returns the same set of result codes as qeErr. See Appendix D, “Result and Error Message Codes,” on page 537 for a list of these result codes.

Example

```
hdbc = qeConnect ("DSN=QEDBF") ;
hstmt = qeExecSQL (hdbc, "SELECT * FROM emp") ;
res_code = qeRecSetConditionDouble (hstmt, 5 ,
    qeFIND_GREATER_THAN, 20000.00) ;
new_hstmt = qeQBEPrepare (hstmt) ;
res_code = qeSQLExecute (new_hstmt) ;
while (qeFetchNext (new_hstmt) == qeSUCCESS )
    ...
/* Get values matching condition. * /
    ...
res_code = qeEndSQL (new_hstmt) ;
res_code = qeEndSQL (hstmt) ;
res_code = qeDisconnect (hdbc) ;
```

qeRecSetConditionFloat

qeRecSetConditionFloat adds a search condition to the statement having a single-precision floating-point value to compare.

Syntax

```
int16  res_code qeRecSetConditionFloat    (
    int16    hstmt,
    int16    col_num,
    int16    operator,
    float32  value)
```

Description

qeRecSetConditionFloat adds a search condition to the statement having a single-precision floating-point value to compare.

For all operators except the IN operator, multiple search conditions for a column are joined with a boolean AND (that is, all conditions must be true for the row to match the search conditions).

For the IN operator, multiple search conditions for a column are joined with a boolean OR (that is, at least one of the IN conditions must be true for the row to match the search conditions).

Parameters

hstmt is the handle to the statement returned by qeExecSQL or qeSQLPrepare.

col_num is the number of the column on which a search condition is being placed. The first column number is 1.

operator is the comparison operator and has one of the following values:

Constant	Value	Operator
qeFIND_LESS_THAN	1	<
qeFIND_LESS_THAN_OR_EQ	2	<=
qeFIND_GREATER_THAN	3	>
qeFIND_GREATER_THAN_OR_EQ	4	>=

Constant	Value	Operator
qeFIND_EQUAL	5	=
qeFIND_NOT_EQUAL	6	<>
qeFIND_IN	9	IN

value points to the single-precision floating-point comparison value.

res_code is the result code returned by qeRecSetConditionFloat, which returns the same set of result codes as qeErr. See Appendix D, "Result and Error Message Codes," on page 537 for a list of these result codes.

Example

```
hdbc = qeConnect ("DSN=QEDBF")   ;
hstmt = qeExecSQL (hdbc, "SELECT * FROM emp")    ;
res_code = qeRecSetConditionFloat (hstmt, 5    ,
    qeFIND_GREATER_THAN, 20000.00)   ;
new_hstmt = qeQBEPrepare (hstmt)   ;
res_code = qeSQLExecute (new_hstmt)   ;
while (qeFetchNext (new_hstmt) == qeSUCCESS    )
    ...
/* Get values matching condition. *   /
    ...
res_code = qeEndSQL (new_hstmt)   ;
res_code = qeEndSQL (hstmt)   ;
res_code = qeDisconnect (hdbc)   ;
```

qeRecSetConditionInt

qeRecSetConditionInt adds a search condition to the statement having a 2-byte integer value to compare.

Syntax

```
int16 res_code qeRecSetConditionInt   (
    int16    hstmt,
    int16    col_num,
    int16    operator,
    int16    value)
```

Description

qeRecSetConditionInt adds a search condition to the statement having a 2-byte integer value to compare.

For all operators except the IN operator, multiple search conditions for a column are joined with a boolean AND (that is, all conditions must be true for the row to match the search conditions).

For the IN operator, multiple search conditions for a column are joined with a boolean OR (that is, at least one of the IN conditions must be true for the row to match the search conditions).

Parameters

hstmt is the handle to the statement returned by qeExecSQL or qeSQLPrepare.

col_num is the number of the column on which a search condition is being placed. The first column number is 1.

operator is the comparison operator and has one of the following values:

Constant	Value	Operator
qeFIND_LESS_THAN	1	<
qeFIND_LESS_THAN_OR_EQ	2	<=
qeFIND_GREATER_THAN	3	>
qeFIND_GREATER_THAN_OR_EQ	4	>=

Constant	Value	Operator
qeFIND_EQUAL	5	=
qeFIND_NOT_EQUAL	6	<>
qeFIND_IN	9	IN

value points to the 2-byte integer comparison value.

res_code is the result code returned by qeRecSetConditionInt, which returns the same set of result codes as qeErr. See Appendix D, “Result and Error Message Codes,” on page 537 for a list of these result codes.

Example

```
hdbc = qeConnect ("DSN=QEDBF") ;
hstmt = qeExecSQL (hdbc, "SELECT * FROM emp") ;
res_code = qeRecSetConditionInt (hstmt, 7 ,
    qeFIND_EQUAL, 1) ;
new_hstmt = qeQBEPrepare (hstmt) ;
res_code = qeSQLExecute (new_hstmt) ;
while (qeFetchNext (new_hstmt) == qeSUCCESS )
    ...
/* Get values matching condition. * /
    ...
res_code = qeEndSQL (new_hstmt) ;
res_code = qeEndSQL (hstmt) ;
res_code = qeDisconnect (hdbc) ;
```

qeRecSetConditionLong

qeRecSetConditionLong adds a search condition to the statement having a 4-byte integer value to compare.

Syntax

```
int16 res_code qeRecSetConditionLong   (
    int16    hstmt,
    int16    col_num,
    int16    operator,
    int32    value)
```

Description

qeRecSetConditionLong adds a search condition to the statement having a 4-byte integer value to compare.

For all operators except the IN operator, multiple search conditions for a column are joined with a boolean AND (that is, all conditions must be true for the row to match the search conditions).

For the IN operator, multiple search conditions for a column are joined with a boolean OR (that is, at least one of the IN conditions must be true for the row to match the search conditions).

Parameters

hstmt is the handle to the statement returned by qeExecSQL or qeSQLPrepare.

col_num is the number of the column on which a search condition is being placed. The first column number is 1.

operator is the comparison operator and has one of the following values:

Constant	Value	Operator
qeFIND_LESS_THAN	1	<
qeFIND_LESS_THAN_OR_EQ	2	<=
qeFIND_GREATER_THAN	3	>
qeFIND_GREATER_THAN_OR_EQ	4	>=

Constant	Value	Operator
qeFIND_EQUAL	5	=
qeFIND_NOT_EQUAL	6	<>
qeFIND_IN	9	IN

value points to the 4-byte integer comparison value.

res_code is the result code returned by qeRecSetConditionLong, which returns the same set of result codes as qeErr. See Appendix D, “Result and Error Message Codes,” on page 537 for a list of these result codes.

Example

```
hdbc = qeConnect ("DSN=QEDBF")   ;
hstmt = qeExecSQL (hdbc, "SELECT * FROM emp")    ;
res_code = qeRecSetConditionLong (hstmt, 5    ,
    qeFIND_GREATER_THAN, 20000)   ;
new_hstmt = qeQBEPrepare (hstmt)   ;
res_code = qeSQLExecute (new_hstmt)   ;
while (qeFetchNext (new_hstmt) == qeSUCCESS    )
    ...
/* Get values matching condition. *   /
    ...
res_code = qeEndSQL (new_hstmt)   ;
res_code = qeEndSQL (hstmt)   ;
res_code = qeDisconnect (hdbc)   ;
```

qeRecSetConditionNull

qeRecSetConditionNull adds a search condition to the statement having a value to compare of null.

Syntax

```
int16 res_code qeRecSetConditionNull   (
    int16    hstmt,
    int16    col_num,
    int16    operator)
```

Description

qeRecSetConditionNull adds a search condition to the statement having a value to compare of null.

For all operators except the IN operator, multiple search conditions for a column are joined with a boolean AND (that is, all conditions must be true for the row to match the search conditions).

For the IN operator, multiple search conditions for a column are joined with a boolean OR (that is, at least one of the IN conditions must be true for the row to match the search conditions).

Parameters

hstmt is the handle to the statement returned by qeExecSQL or qeSQLPrepare.

col_num is the number of the column on which a search condition is being placed. The first column number is 1.

operator is the comparison operator and has one of the following values:

Constant	Value	Operator
qeFIND_EQUAL	5	=
qeFIND_NOT_EQUAL	6	<>
qeFIND_IN	9	IN

res_code is the result code returned by qeRecSetConditionNull, which returns the same set of result codes as qeErr. See Appendix D, "Result and Error Message Codes," on page 537 for a list of these result codes.

Example

```
hdbc = qeConnect ("DSN=QEDBF")   ;
hstmt = qeExecSQL (hdbc, "SELECT * FROM emp")     ;
res_code = qeRecSetConditionNull (hstmt, 3,
    qeFIND_NOT_EQUAL) ;
new_hstmt = qeQBEPrepare (hstmt)   ;
res_code = qeSQLExecute (new_hstmt)   ;
while (qeFetchNext (new_hstmt) == qeSUCCESS    )
    ...
/* Get values matching condition. *   /
    ...
res_code = qeEndSQL (new_hstmt)   ;
res_code = qeEndSQL (hstmt)   ;
res_code = qeDisconnect (hdbc)   ;
```

qeRecSetKey

qeRecSetKey determines which columns DTK uses to uniquely identify a row.

Syntax

```
int16 res_code qeRecSetKey  (
    int16    hstmt,
    int16    col_num,
    int16    value)
```

Description

A column that helps uniquely identify records in the database is part of a *primary key* for the database. When qeRecDelete, qeRecUpdate, and qeRecLock are called, the columns specified by qeRecSetKey are used to help identify the record within the result set to be operated on. DTK uses these columns in a Where clause that uniquely identifies the current record in the buffer in the statement it generate for the database operation.

If no columns are flagged as being part of the unique key when qeRecDelete, qeRecUpdate**,** qeRecLock**,** or qeUniqueWhereClause is called, DTK chooses a set of columns as the key. These columns are set as the unique key until the user changes them. A call to qeRecGetKey reports an individual column's presence in the key. To return the complete set of columns that DTK will choose for the key, call qeUniqueWhereClause.

See "Unique Keys" on page 78 for more information on DTK's use of unique keys.

An error is issued if the column is not valid for use in a primary key.

Parameters

hstmt is the handle to the statement returned by qeExecSQL or qeSQLPrepare.

col_num is the column number that is to be included in the primary key. The first column number is 1.

value is TRUE (1) to set the column as a key, and FALSE (0) to exclude the column from the primary key.

res_code is the result code returned by qeRecSetKey, which returns the same set of result codes as qeErr. See Appendix D, “Result and Error Message Codes,” on page 537 for a list of these result codes.

Example

```
hdbc = qeConnect ("DSN=QEDBF")   ;
hstmt = qeExecSQL (hdbc, "SELECT * FROM emp")    ;
/* Make 4th column part of the key *    /
res_code = qeRecSetKey (hstmt, 4, 1)    ;
...
res_code = qeEndSQL (hstmt)   ;
res_code = qeDisconnect (hdbc)   ;
```

See Also

qeRecGetKey, qeSetLockOptions, qeUniqueWhereClause and qeUniqueWhereClauseBuf.

qeRecState

qeRecState returns the state of the current record.

Syntax

```
int16 rec_state qeRecState (int16 hstmt)
```

Parameters

hstmt is the handle to the statement returned by qeExecSQL or qeSQLPrepare.

rec_state is the returned state of the record. It has one of the following values:

Constant	Value	Description
qeSTATE_NEW	1	The record is a new record that has not been sent to the database and contains no fields that have been updated.
qeSTATE_UNCHANGED	2	The record has no changes waiting to be sent to the database.
qeSTATE_CHANGED	3	The record has changes waiting to be sent to the database.
qeSTATE_NOREC	4	The hstmt is not currently positioned on a record.
qeSTATE_NEW_ CHANGED	5	The record is new and has not been sent to the database but has had one or more columns modified by calls to qePut functions.

Example

```
hdbc = qeConnect ("DSN=QEDBF")   ;
hstmt = qeExecSQL (hdbc, "SELECT * FROM emp")    ;
res_code = qeFetchNext (hstmt)   ;
state = qeRecState (hstmt)   ;
...
res_code = qeEndSQL (hstmt)   ;
res_code = qeDisconnect (hdbc)   ;
```

qeRecUndo

qeRecUndo discards changes to the current record that have not been sent to the database.

Syntax

```
int16 res_code qeRecUndo (int16  hstmt)
```

Description

qeRecUndo discards all changes that have been performed on the current record but have not been sent to the database.

Parameters

hstmt is the handle to the statement returned by qeExecSQL or qeSQLPrepare.

res_code is the result code returned by qeRecUndo, which returns the same set of result codes as qeErr. See Appendix D, “Result and Error Message Codes,” on page 537 for a list of these result codes.

Example

```
hdbc = qeConnect ("DSN=QEDBF")   ;
res_code = qeBeginTran (hdbc)   ;
hstmt = qeExecSQL (hdbc, "SELECT * FROM emp")    ;
res_code = qeFetchNext (hstmt)   ;
res_code = qePutChar (hstmt, 1, "", "Mike")    ;
res_code = qePutChar (hstmt, 2, "", "McGarrah")    ;
res_code = qeRecUndo (hstmt)   ;
...
res_code = qeEndSQL (hstmt)   ;
res_code = qeCommit (hdbc)   ;
res_code = qeDisconnect (hdbc)   ;
```

qeRecUpdate

qeRecUpdate updates the current record with the new values set using qePut functions.

Syntax

```
int16 res_code qeRecUpdate (int16   hstmt)
```

Description

qeRecUpdate updates the current record with new values that were set using qePut functions. It also inserts a record that was created by qeRecNew.

You can call qeNumModRecs to determine the number of records affected by a call to qeRecUpdate.

Calling this function causes DTK to generate a unique key if you have not already defined one with qeRecSetKey.

Parameters

hstmt is the handle to the statement returned by qeExecSQL or qeSQLPrepare.

res_code is the result code returned by qeRecUpdate, which returns the same set of result codes as qeErr. See Appendix D, "Result and Error Message Codes," on page 537 for a list of these result codes.

Example

```
hdbc = qeConnect ("DSN=QEDBF")   ;
hstmt = qeExecSQL (hdbc, "SELECT * FROM emp")    ;
res_code = qeFetchNext (hstmt)   ;
res_code = qePutChar (hstmt, 1, "", "Mike")    ;
res_code = qePutChar (hstmt, 2, "", "McGarrah")    ;
res_code = qeRecUpdate (hstmt)   ;
res_code = qeEndSQL (hstmt)   ;
res_code = qeDisconnect (hdbc)   ;
```

Notes

If you call qeRecUpdate without having called a qeFetch function or qeRecNew, DTK returns an error. For example, if you call qeExecSQL and then immediately call qeRecUpdate on the new *hstmt*, DTK cannot update a record because the *hstmt* is still positioned on record 0 (no record). In order to update a record, you must first call qeFetchNext to position on the first record in the buffer (record 1).

Important: To update the current record, qeRecUpdate generates a SQL Update statement that uses a Where clause to uniquely identify that record. If this Where clause matches multiple records, qeRecUpdate updates all matching records. You can recover from such invalid modifications by using transactions and calling qeNumModRecs after each call to qeRecUpdate to verify that multiple records were not affected. Calling qeRecLock before calls to qeRecUpdate can also help prevent multiple modifications, since qeRecLock uses the same Where clause as qeRecUpdate and returns a warning if it locks multiple records.

See Also

qeRecSetKey.

qeRollback

qeRollback ends a database transaction and cancels all changes to the database made during the transaction.

Syntax

```
int16 res_code qeRollback (int16  hdbc)
```

Description

qeRollback discards all changes made on the connection since qeBeginTran was called and removes all locks held in the database system.

The discarded changes include any saved changes on records other than the current record, any records created by calling qeRecNew, and any new values placed in the current record by calls to qePut functions.

After a rollback, DTK is positioned between what was the last current record in the transaction and the next record in the *hstmt*. Before you perform any operations against the records, call one of the qeFetch functions to position on a valid record.

You must call qeBeginTran to start a transaction before you can call qeRollback to undo all changes.

Parameters

hdbc is the handle to the database connection returned by qeConnect.

res_code is the result code returned by qeRollback, which returns the same set of result codes as qeErr. See Appendix D, “Result and Error Message Codes,” on page 537 for a list of these result codes.

Example

To roll back changes made to a SQL Server database:

```
hdbc=qeConnect ("DSN=QESS;UID=sa;SRVR=PION1")    ;
...
res_code = qeBeginTran (hdbc)   ;
hstmt = qeExecSQL (hdbc   ,
    "UPDATE emp SET salary=salary*1.1")    ;
res_code = qeEndSQL (hstmt)   ;
res_code = qeRollback (hdbc)   ;
res_code = qeDisconnect (hdbc)   ;
```

See Also

qeBeginTran, qeCommit.

qeSetAutoUpdate

qeSetAutoUpdate determines what happens when the *hstmt* is moved to a new record before changed values have been updated or inserted.

Syntax

```
int16 res_code qeSetAutoUpdate (int16  hdbc, int16 option)
```

Description

qeSetAutoUpdate determines what happens when the *hstmt* is moved to a new record before changed values have been updated or inserted by a call to qeRecUpdate. When *option* is set to qeAUTOUP_UPDATE (3), a call to qeFetchNext or any other command that changes the current record number causes DTK to automatically update the current record if any changes have been made to it. When *option* is set to qeAUTOUP_DEFER (2), changes can be deferred—saved but not updated in the database—until a call to qeApplyAll, qeUndoAll**,** qeRecUndo**,** or qeRollback updates the database or discards the changes. The default is qeAUTOUPD_DISCARD (1), which causes DTK to discard changes or insertions.

Parameters

hdbc is the handle to the database connection returned by qeConnect.

option determines whether DTK automatically generates Update or Insert statements when you move off a changed or inserted row. It has one of the following values:

Constant	Value	Action
qeAUTOUPD_ DISCARD	1	DTK discards changes or insertions. This is the default.
qeAUTOUPD_ DEFER	2	DTK saves the changes but does not update the database. This option enables you to use the qeApplyAll and qeUndoAll functions.
qeAUTOUPD_ UPDATE	3	DTK updates the changed or inserted record.

res_code is the result code returned by qeSetAutoUpdate, which returns the same set of result codes as qeErr. See Appendix D, "Result and Error Message Codes," on page 537 for a list of these result codes.

Example

```
hdbc = qeConnect ("DSN=QEDBF")   ;
res_code = qeSetAutoUpdate (hdbc, qeAUTOUPD_DEFER)     ;
hstmt = qeExecSQL (hdbc, "SELECT * FROM emp")    ;
res_code = qeFetchNext (hstmt)   ;
res_code = qePutLong (hstmt, 5, 32000)    ;
res_code = qeFetchNext (hstmt)   ;
/* At this point, the change to the previous record *     /
/* has not been sent to the database, but if the user *     /
/* were to position back to the first record, and issue
*/
/* a qeRecUpdate, the modification would be made. *     /
res_code = qeEndSQL (hstmt)   ;
res_code = qeDisconnect (hdbc)   ;
```

See Also

qeGetAutoUpdate, qeApplyAll, qeUndoAll, qeRecUndo, qeRollback.

qeSetCacheFileName

qeSetCacheFileName sets the file name to be used when caching table names.

Syntax

```
int16 res_code qeSetCacheFileName   (
    int16    hdbc,
    ptrstr   file_name)
```

Description

You can call this function to set the file name to be used when caching of table names is enabled.

The qeSetTableCaching function determines whether the results of qeTables calls are cached. You can call qeGetTableCaching to determine the level of caching enabled. If table caching is set to qeCACHE_PERMANENT (1), you can reuse an existing cache file by specifying it in a call to this function.

A cache file is maintained for each connection.

Important: If session caching is in progress when you call qeSetCacheFileName, the existing cache file is deleted.

Parameters

hdbc is the handle to the connection returned by qeConnect.

file_name is the name of the file to use for caching. It must be a valid name for the operating system you are using. A null value results in a system-generated temporary file being used.

res_code is the result code returned by qeSetCacheFileName, which returns the same set of result codes as qeErr. See Appendix D, "Result and Error Message Codes," on page 537 for a list of these result codes.

Example

```
hdbc = qeConnect ("DSN=QEDBF")   ;
res_code = qeSetCacheFileName (hdbc, "CacheF")    ;
...
res_code = qeDisconnect (hdbc)   ;
```

See Also

qeSetTableCaching.

qeSetDB

qeSetDB sets the default database in database systems that allow tables to be stored in separate databases.

Syntax

```
int16 res_code qeSetDB (int16 hdbc, ptrstr database)
```

Description

When using a database system that lets you store tables in separate databases, you can set the default database for your application with a call to qeSetDB. All subsequent SQL statements are sent to this database.

This function is supported by a limited number of database systems.

Parameters

hdbc is the handle to the database connection returned by qeConnect.

database is the name of the database to become the default.

res_code is the result code returned by qeSetDB, which returns the same set of result codes as qeErr. See Appendix D, "Result and Error Message Codes," on page 537 for a list of these result codes.

Example

To change the SQL Server default database:

```
hdbc = qeConnect ("DSN=QESS;UID=sa;SRVR=PION1")    ;
...
res_code = qeSetDB (hdbc, "pubs")   ;
hstmt = qeExecSQL (hdbc   ,
    "SELECT * FROM authors")   ;
...
res_code = qeEndSQL (hstmt)   ;
res_code = qeDisconnect (hdbc)   ;
```

qeSetDriverTracefile

qeSetDriverTracefile specifies a driver trace file.

Syntax

```
int16  res_code qeSetDriverTracefile   (
    int16     hdbc,
    ptrstr    file_name)
```

Description

qeSetDriverTracefile lets you specify a file to which driver tracing is written. This file traces the ODBC calls made by DTK, and so is not the same as the standard DTK trace file.

This function is useful only when ODBC tracing is enabled by a call to qeSetTraceOptions.

Parameters

hdbc is a handle to a database connection obtained from qeConnect.

file_name points to the name of the file to which trace information should be written. It must be a valid name for the operating system you are using. If null, trace information is written to SQL.LOG.

res_code is the result code returned by qeSetDriverTracefile, which returns the same set of result codes as qeErr. See Appendix D, "Result and Error Message Codes," on page 537 for a list of these result codes.

Example

```
res_code = qeTraceOn ("\\trace.txt")    ;
res_code = qeSetTraceOptions (qeTRACE_ODBC)     ;
res_code = qeSetDriverTracefile (hdbc, "\\odbctrc.txt")     ;
hdbc = qeConnect ("DSN=QEDBF")   ;
...
res_code = qeDisconnect (hdbc)   ;
res_code = qeTraceOff ()  ;
```

See Also

qeSetTraceOptions.

qeSetIsolationLevel

qeSetIsolationLevel sets the isolation level for the connection.

Syntax

```
int16 res_code qeSetIsolationLevel(int16   hdbc, int16
level)
```

Description

qeSetIsolationLevel sets the isolation level for the database to which you are connected. An isolation level represents a particular locking strategy employed in the database to improve data consistency. The higher the isolation level, the more complex the locking strategy behind it. The following table shows what data consistency behaviors can occur at each isolation level:

Level	Dirty reads	Non-repeatable reads	Phantom reads
0, Read uncommitted	Yes	Yes	Yes
1, Read committed	No	Yes	Yes
2, Repeatable read	No	No	Yes
3, Serializable (4, Versioning)	No	No	No

These behaviors are described along with other information on isolation levels in "Isolation Levels" on page 85.

The isolation levels supported and default isolation level are database-dependent. Many databases support only a subset of these isolation levels. Call qeGetSupportedIsolationLevels, which returns the set of isolation levels the database supports, before calling qeSetIsolationLevel.

Parameters

hdbc is the handle to the database connection returned by qeConnect.

level is the isolation level that is to be set in the database. It is one of the following values:

Constant	Value	Description
qeISO_READ_UNCOMMITTED	0x0001	Read uncommitted (0) isolation level. Locks are obtained on modifications to the database and held until end of transaction (EOT). Reading from the database does not involve any locking.
qeISO_READ_COMMITTED	0x0002	Read committed (1) isolation level. Locks are acquired for reading and modifying the database. Locks are released after reading but locks on modified objects are held until EOT.
qeISO_REPEATABLE_READ	0x0004	Repeatable read (2) isolation level. Locks are obtained for reading and modifying the database. Locks on all modified objects are held until EOT. Locks obtained for reading data are held until EOT. Locks on non-modified access structures (indexes, hashing structures, etc.) are released after reading.
qeISO_SERIALIZABLE	0x0008	Serializable (3) isolation level. All data read or modified is locked until EOT. All access structures that are modified are locked until EOT. Access structures used by the query are locked until EOT.
qeISO_VERSIONING	0x0010	Versioning (4) isolation level. Similar to isolation level 3, serializable, but provides greater concurrence through the use of non-locking “record versioning” protocols.

res_code is the result code returned by qeSetIsolationLevel, which returns the same set of result codes as qeErr. See Appendix D, “Result and Error Message Codes,” on page 537 for a list of these result codes.

Example

```
hdbc = qeConnect ("DSN=QESS")   ;
levels = qeGetSupportedIsolationLevels (hdbc)     ;
cur_level = qeGetIsolationLevel (hdbc)    ;
if (levels & qeISO_READ_COMMITTED    )
    res_code = qeSetIsolationLevel (hdbc,
             qeISO_READ_COMMITTED)  ;
res_code = qeDisconnect (hdbc)   ;
```

See Also

qeGetIsolationLevel, qeGetSupportedIsolationLevels.

qeSetLockOptions

qeSetLockOptions controls the behavior of qeRecLock in regard to records that may have changed in the database since they were initially read.

Syntax

```
int16 res_code qeSetLockOptions (int16  hdbc, int16
option)
```

Description

qeSetLockOptions sets the behavior of the qeRecLock function, providing options that help you avoid locking and updating records in the log file that have changed in the database since they were first read. By default, you can lock and update such records. However, by setting the qeLOCK_COMPARE or qeLOCK_REFRESH options, you can have DTK either warn you when the locked record has changed or automatically refresh the copy in the log file with the corresponding values from the database so that the values you see are always current.

Calls to qeSetLockOptions are not cumulative; the options it sets are valid for the entire connection or until you change them by calling this function.

Parameters

hdbc is the handle to the database connection returned by qeConnect.

option lets you control DTK's optional locking behavior. You can specify one of the following values:

Constant	Value	Description
qeLOCK_NO_OPTIONS	0	Default; DTK neither compares nor refreshes the record in the log file.
qeLOCK_COMPARE	1	When locking, DTK compares the record in the log file to the corresponding record in the database, and raises a warning if they are different.
qeLOCK_REFRESH	2	When locking, DTK automatically refreshes the record in the log file with new column values.

res_code is the result code returned by qeSetLockOptions, which returns the same set of result codes as qeErr. See Appendix D, “Result and Error Message Codes,” on page 537 for a list of these result codes.

Example

```
hdbc = qeConnect ("DSN=QEDBF")   ;
res_code = qeSetLockOptions (hdbc, qeLOCK_COMPARE)     ;
/* Set locking to compare and raise a *    /
/* warning if buffer differs for log file.*    /
hstmt = qeExecSQL (hdbc, "SELECT * FROM emp")     ;
/* Statement has lock options set to qeLOCK_COMPARE. *     /
res_code = qeDisconnect (hdbc)   ;
```

See Also

qeRecLock, qeGetLockOptions.

qeSetLoginTimeout

qeSetLoginTimeout sets the number of seconds to wait for a login request to complete before returning.

Syntax

```
int16 res_code qeSetLoginTimeout (int32    seconds)
```

Description

qeSetLoginTimeout sets the login timeout, in seconds.

This function has no effect if the driver does not support timeouts.

Parameters

seconds is the number of seconds to wait for a login to complete. The default is 15. If *seconds* is 0, a connection attempt waits indefinitely.

res_code is the result code returned by qeSetLoginTimeout, which returns the same set of result codes as qeErr. See Appendix D, “Result and Error Message Codes,” on page 537 for a list of these result codes.

Example

To have SQL Server wait indefinitely:

```
res_code = qeSetLoginTimeout (0)   ;
hdbc = qeConnect ("DSN=qess")   ;
...
res_code = qeDisconnect (hdbc)   ;
```

See Also

qeGetLoginTimeout.

qeSetMaxRows

qeSetMaxRows sets the maximum number of rows that a statement returns. You can call this function to limit the amount of records that a Select statement will return.

Syntax

```
int16 res_code qeSetMaxRows (int16  hdbc, int32 max_rows)
```

Parameters

hdbc is the handle to the database connection returned by qeConnect.

max_rows is the maximum number of rows that should be returned for the query. 0, the default, indicates that all rows are to be returned.

res_code is the result code returned by qeSetMaxRows, which returns the same set of result codes as qeErr. See Appendix D, “Result and Error Message Codes,” on page 537 for a list of these result codes.

Example

```
hdbc = qeConnect ("DSN=QESS")   ;
res_code = qeSetMaxRows (hdbc, 10)   ;
hstmt = qeExecSQL (hdbc, "SELECT * FROM emp")    ;
res_code = qeEndSQL (hstmt)   ;
res_code = qeDisconnect (hdbc)   ;
```

See Also

qeGetMaxRows.

qeSetODBCHdbc

qeSetODBCHdbc creates a DTK *hdbc* from the ODBC *hdbc*.

Syntax

```
int16 hdbc qeSetODBCHdbc (ptrstr ODBCHdbc)
```

Description

qeSetODBCHdbc creates a DTK *hdbc* from the ODBC *hdbc*. This function is useful when you want to connect to a database using the ODBC SQLDriverConnect or SQLBrowseConnect functions. After establishing a connection via the ODBC function, you can call qeSetODBCHdbc to convert the ODBC connection handle to a handle usable by DTK functions.

Important: This function is potentially dangerous. Using the ODBC *hdbc* to change the state of the ODBC connection may create situations that trap. There is no guarantee of proper behavior when you call qeSetODBCHdbc, because DTK cannot know any information about the *hstmt* or *hdbc* involved. Use at your own risk.

Parameters

hdbc is the handle to the connection returned by qeSetODBCHdbc.

ODBCHdbc is a pointer to the *hdbc* returned by the ODBC SQLConnect, SQLBrowseConnect, or SQLDriverConnect function.

res_code is the result code returned by qeSetODBCHdbc, which returns the same set of result codes as qeErr. See Appendix D, "Result and Error Message Codes," on page 537 for a list of these result codes.

Example

```
...
/* Previous code retrieved an ODBC hdbc */
hdbc = qeSetODBCHdbc (odbc_hdbc);
...
/* Use as a normal hdbc. */
res_code = qeDisconnect (hdbc);
```

qeSetOneHstmtPerHdbcOptions

qeSetOneHstmtPerHdbcOptions sets options that determine which fetching commands and statement behaviors are allowed by DTK.

Note: If the data source to which you are connected supports more than one *hstmt* per *hdbc*, this function has no effect.

Syntax

```
int16  res_code qeSetOneHstmtPerHdbcOptions   (
    int16    hdbc,
    int32    flags)
```

Parameters

hdbc is the handle to the database connection returned by qeConnect.

flags is a set of option flags that controls read-ahead activity, statement routing, and *hstmt* behavior when DTK uses multiple connections to databases that support only one statement per connection. You can set one read-ahead, routing, and *hstmt* option from among the following:

Constant	Value	Description
qeREADAHEAD_AT_EXEC	0x0001	DTK reads the statement's entire result set into the log file when the statement executes. Reading result sets at this time will often free handles for users of databases who have licenses restricting open handles.
qeREADAHEAD_AT_UPDATE	0x0002	DTK reads the remainder of the result set into the log file whenever a record is locked, updated, or deleted. This is the default read-ahead option.
qeREADAHEAD_COMMIT_UPDATES	0x0003	DTK avoids all read-ahead activity by requiring you to commit all updates before fetching any more records.
qeROUTING_READ	0x0008	DTK routes this statement through a connection used for read-only statements.

Constant	Value	Description
qeROUTING_UPDATE	0x0010	DTK routes this statement through a connection used for statements that modify the database.
qeROUTING_DEFAULT	0x0018	This option lets DTK decide which connection to send the statement to. This is the default routing option.
qeHSTMT_LOCAL	0x0020	Tells DTK that this *hstmt* cannot affect any other active *hstmt* in the same application.
qeHSTMT_NONLOCAL	0x0040	Tells DTK that this *hstmt* may affect other *hstmts* in the same application. This is the default *hstmt* behavior.

These values can be combined by adding them together or joining them with an OR clause. For example, the default is qeREADAHEAD_AT_UPDATE + qeROUTING_DEFAULT + qeHSTMT_NONLOCAL.

res_code is the result code returned by qeSetOneHstmtPerHdbc-Options, which returns the same set of result codes as qeErr. See Appendix D, “Result and Error Message Codes,” on page 537 for a list of these result codes.

Example

```
hdbc = qeConnect ("DSN=QESS")   ;
res_code = qeSetOneHstmtPerHdbcOptions (hdbc,
qeREADAHEAD_AT_UPDATE +qeHSTMT_LOCAL)    ;
hstmt = qeExecSQL (hdbc, "SELECT * FROM emp")    ;
...
/* Options will affect what happens if records *    /
/* are modified on this hstmt. *   /
...
res_code = qeEndSQL (hstmt)   ;
res_code = qeDisconnect (hdbc)   ;
```

See Also

For more information on using this function, see Appendix C, “Coding for Single Statement Database Systems,” on page 529.

qeSetParamBinary

qeSetParamBinary sets the value of a binary parameter.

Syntax

```
int16  res_code qeSetParamBinary   (
    int16     hstmt,
    int16     param_num,
    ptrstr    param_val,
    int32     param_len)
```

Description

qeSetParamBinary assigns the value of a parameter in a SQL statement to a binary value.

DTK copies the assigned value, so the pointer need not remain valid after this call. This parameter has this value until qeClearParam or a qeSetParam or qeBindParam function is called again for this parameter. All parameters with the same name as the one identified by *param_num* are affected.

Before calling qeSetParamBinary, you must call qeSQLPrepare. You must give values to all parameters before calling qeSQLExecute.

Parameters

hstmt is the handle to the statement returned by qeSQLPrepare**,** qeQryPrepare**,** or qeQBEPrepare.

param_num is the position of the parameter to be set. The first parameter number is 1.

param_val is the value to be assigned to the parameter.

param_len is the number of valid bytes in *param_val*.

res_code is the result code returned by qeSetParamBinary, which returns the same set of result codes as qeErr. See Appendix D, “Result and Error Message Codes,” on page 537 for a list of these result codes.

Example

```
hdbc = qeConnect ("DSN=QEDBF")   ;
hstmt = qeSQLPrepare (hdbc,
    "INSERT INTO emp (MEMO) VALUES (?)")   ;
/* bindata contains binary information. *    /
res_code = qeSetParamBinary (hstmt, 1, bindata, 10)     ;
res_code = qeSQLExecute (hstmt)   ;
res_code = qeEndSQL (hstmt)   ;
res_code = qeDisconnect (hdbc)   ;
```

qeSetParamChar

qeSetParamChar sets the value of a character parameter.

Syntax

```
int16 res_code qeSetParamChar (
    int16    hstmt,
    int16    param_num,
    ptrstr   param_val,
    int32    max_len)
```

Description

qeSetParamChar assigns the value of a parameter in a SQL statement to a character value.

DTK copies the assigned value, so the pointer need not remain valid after this call. This parameter has this value until qeClearParam or a qeSetParam or qeBindParam function is called again for this parameter. All parameters with the same name as the one identified by *param_num* are affected.

Before calling qeSetParamChar, you must call qeSQLPrepare. You must give values to all parameters before calling qeSQLExecute.

qeSetParamChar may be called multiple times before executing, resulting in the parameter value being set to the concatenation of all values sent. Lengths of zero are ignored.

Parameters

hstmt is the handle to the statement returned by qeSQLPrepare.

param_num is the position of the parameter to be set.

param_val is the character value to be assigned to the parameter.

max_len is the size of the column with which this parameter is associated. This setting determines whether the parameter is of varying character or long varying character type. If *max_len* is less than or equal to the largest character string allowed by the database, then the parameter is varying character type. If greater, it is long varying character type.

Important: A mismatch between the parameter type and the database column type (varying character versus long varying character) may cause unusual problems for some database drivers, for which no errors are returned.

res_code is the result code returned by qeSetParamChar, which returns the same set of result codes as qeErr. See Appendix D, "Result and Error Message Codes," on page 537 for a list of these result codes.

Example

```
hdbc = qeConnect ("DSN=QEDBF")   ;
hstmt = qeSQLPrepare (hdbc, "SELECT * FROM emp WHERE
last_name = ?") ;
res_code = qeSetParamChar (hstmt, 1, "Joe", 10)    ;
res_code = qeSQLExecute (hstmt)   ;
res_code = qeEndSQL (hstmt)   ;
res_code = qeDisconnect (hdbc)   ;
```

qeSetParamDataType

qeSetParamDataType sets the data type of a stored procedure's output parameters.

Syntax

```
int16  res_code qeSetParamDataType   (
    int16    hstmt,
    int16    param_num,
    int16     param_type,
    int32    precision,
    int16    scale)
```

Description

When the qeSetParam and qeGetParam functions are being used in place of the qeBindParam functions, you should call qeSetParamDataType for every output parameter.

This function is used only with output parameters. Thus, before qeSetParamDataType can be called for a parameter, qeSetParamIOType must be called for that parameter to set it as an output parameter.

When binding parameters, you must call a qeBindParam function for each parameter to create a buffer to pull the input value from or put the output value into; since the qeBindParam functions set the data type for all parameters, you do not need to call qeSetParamDataType when you bind parameters.

When using the qeSetParam and qeGetParam functions instead of binding, you must call qeSetParam for all input and all input/output parameters. Because the qeSetParam functions cannot set the data type for output parameters, you must use qeSetParamDataType for output parameters.

Calling both qeSetParamDataType and a qeBindParam/qeSetParam function for the same parameter does not result in an error as long as the data type and data size passed for the parameter are the same in both calls; if the parameter's data type or data size conflicts between the two calls, an error is issued.

Calling this function on an input or an input/output parameter results in an error.

Parameters

hstmt is the handle to the statement returned by qeSQLPrepare.

param_num is the position of the parameter to be set.

param_type is the data type of the specified parameter. It can have one of the following values:

Constant	Value	Description
qeCHAR	1	Blank-padded, fixed-length string.
qeVARCHAR	2	Variable-length string.
qeDECIMAL	3	BCD number.
qeINTEGER	4	4-byte signed integer.
qeSMALLINT	5	2-byte signed integer.
qeFLOAT	6	4-byte floating-point number.
qeDOUBLEPRECISION	7	8-byte floating-point number.
qeDATETIME	8	26-byte date time value. Example: YYYY-MM-DD HH:MM:SS:FFFFFF
qeBINARY	101	Binary string.
qeVARBINARY	102	Variable-length binary string.
qeBIT	110	Bit value.
qeDATE	111	26-byte date value.
qeTIME	112	26-byte time value.
qeNO_DATA_TYPE	0	No data type.

precision varies by data type. For a decimal value, it is the total number of digits returned. For a character string or binary value, it is the maximum number of characters returned. For a date-time value, it is the number of characters from the returned value to actually use (16, 19, 23, or 26). This value is required only if applicable to the parameter whose data type is being set.

scale is a decimal value's scale. This value is required only if applicable to the parameter whose data type is being set.

res_code is the result code returned by the qeSetParamIOType function, which returns the same set of result codes as qeErr. See Appendix D, "Result and Error Message Codes," on page 537 for a list of these result codes.

Example

```
hdbc = qeConnect (“DSN=QEORA;DLG=2”)    ;
hstmt = qeSQLPrepare (hdbc, “{call = GetDeptName(?)}”)     ;
res_code = qeSetParamIOType (hstmt, 1, qePARAM_OUTPUT)     ;
res_code = qeSetParamDataType (hstmt, 1, qeCHAR, 10, 0)     ;
res_code = qeSQLExecute (hstmt)   ;
dept_name = qeGetParamChar (hstmt, 1, “”, 10    )
...
res_code = qeEndSQL (hstmt)   ;
res_code = qeDisconnect (hstmt)   ;
```

qeSetParamDate

qeSetParamDate sets the value of a date parameter.

Syntax

```
int16  res_code qeSetParamDate   (
    int16     hstmt,
    int16     param_num,
    ptrstr    param_val)
```

Description

qeSetParamDate assigns the value of a parameter in a SQL statement to a date value.

DTK copies the assigned value, so the pointer need not remain valid after this call. This parameter has this value until qeClearParam or a qeSetParam or qeBindParam function is called again for this parameter. All parameters with the same name as the one identified by *param_num* are affected.

Before calling qeSetParamDate, you must call qeSQLPrepare. You must give values to all parameters before calling qeSQLExecute.

Parameters

hstmt is the handle to the statement returned by qeSQLPrepare.

param_num is the position of the parameter to be set.

param_val is the 26-byte date value to be assigned to the parameter.

res_code is the result code returned by qeSetParamDate, which returns the same set of result codes as qeErr. See Appendix D, "Result and Error Message Codes," on page 537 for a list of these result codes.

Example

```
hdbc = qeConnect ("DSN=QEDBF")   ;
hstmt = qeSQLPrepare (hdbc, "SELECT * FROM emp WHERE
    hire_date = ?") ;
res_code = qeSetParamDate (hstmt, 1,
    "1983-06-01 00:00:00:000000")   ;
res_code = qeSQLExecute (hstmt)   ;
res_code = qeEndSQL (hstmt)   ;
res_code = qeDisconnect (hdbc)   ;
```

qeSetParamDateTime

qeSetParamDateTime sets the value of a date-time parameter.

Syntax

```
int16 res_code qeSetParamDateTime   (
    int16    hstmt,
    int16    param_num,
    ptrstr   param_val,
    int16    precision)
```

Description

qeSetParamDateTime assigns the value of a parameter in a SQL statement to a date-time value. DTK copies the assigned value, so the pointer need not remain valid after this call. This parameter has this value until qeClearParam or a qeSetParam or qeBindParam function is called again for this parameter. All parameters with the same name as the one identified by *param_num* are affected.

Before calling qeSetParamDateTime, you must call qeSQLPrepare. You must give values to all parameters before calling qeSQLExecute.

Parameters

hstmt is the handle to the statement returned by qeSQLPrepare.

param_num is the position of the parameter to be set.

param_val is the 26-byte date-time value to be assigned.

precision is the length of the date-time value to be assigned. It is a 2-byte integer giving the number of characters in *param_val* to use: 16, 19, 23, or 26.

res_code is the result code returned by qeSetParamDateTime, which returns the same set of result codes as qeErr. See Appendix D, “Result and Error Message Codes,” on page 537 for a list of these result codes.

Example

```
hdbc = qeConnect ("DSN=QEDBF")   ;
hstmt = qeSQLPrepare (hdbc,"SELECT * FROM emp WHERE
hire_date = ?") ;
res_code = qeSetParamDateTime (hstmt, 1,
    "1983-06-01 12:00:00:000000", 26)   ;
res_code = qeSQLExecute (hstmt)   ;
res_code = qeEndSQL (hstmt)   ;
res_code = qeDisconnect (hdbc)   ;
```

qeSetParamDecimal

qeSetParamDecimal sets the value of a decimal parameter.

Syntax

```
int16  res_code qeSetParamDecimal   (
    int16     hstmt,
    int16     param_num,
    ptrstr    param_val,
    int16     precision,
    int16     scale)
```

Description

qeSetParamDecimal assigns the value of a parameter in a SQL statement to a decimal value. The value is formatted based on the values of *precision* and *scale*.

DTK copies the assigned value, so the pointer need not remain valid after this call. This parameter has this value until qeClearParam or a qeSetParam or qeBindParam function is called again for this parameter. All parameters with the same name as the one identified by *param_num* are affected.

Before calling qeSetParamDecimal, you must call qeSQLPrepare. You must give values to all parameters before calling qeSQLExecute.

Parameters

hstmt is the handle to the statement returned by qeSQLPrepare.

param_num is the position of the parameter to be set.

param_val is the value to be assigned to the parameter.

precision is the number of digits in the value.

scale is the number of digits to the right of the decimal point.

res_code is the result code returned by qeSetParamDecimal, which returns the same set of result codes as qeErr. See Appendix D, “Result and Error Message Codes,” on page 537 for a list of these result codes.

Example

```
hdbc = qeConnect ("DSN=QEDBF")   ;
hstmt = qeSQLPrepare (hdbc, "SELECT * FROM emp
    WHERE salary = ?")  ;
res_code = qeSetParamDecimal (hstmt, 5, dec_val, 9, 2)     ;
res_code = qeSQLExecute (hstmt)   ;
res_code = qeEndSQL (hstmt)   ;
res_code = qeDisconnect (hdbc)   ;
```

qeSetParamDouble

qeSetParamDouble sets the value of a double-precision floating-point parameter.

Syntax

```
int16  res_code qeSetParamDouble   (
    int16    hstmt,
    int16     param_num,
    float64  param_val)
```

Description

qeSetParamDouble assigns the value of a parameter in a SQL statement to a double-precision floating-point value.

DTK copies the assigned value, so the pointer need not remain valid after this call. This parameter has this value until qeClearParam or a qeSetParam or qeBindParam function is called again for this parameter. All parameters with the same name as the one identified by *param_num* are affected.

Before calling qeSetParamDouble, you must call qeSQLPrepare. You must give values to all parameters before calling qeSQLExecute.

Parameters

hstmt is the handle to the statement returned by qeSQLPrepare.

param_num is the position of the parameter to be set.

param_val is the double-precision floating-point value to be assigned.

res_code is the result code returned by qeSetParamDouble, which returns the same set of result codes as qeErr. See Appendix D, “Result and Error Message Codes,” on page 537 for a list of these result codes.

Example

```
hdbc = qeConnect ("DSN=QEDBF")   ;
hstmt = qeSQLPrepare (hdbc, "SELECT * FROM emp
    WHERE salary = ?")  ;
res_code = qeSetParamDouble (hstmt, 1, 32000.00)    ;
res_code = qeSQLExecute (hstmt)   ;
res_code = qeEndSQL (hstmt)   ;
res_code = qeDisconnect (hdbc)   ;
```

qeSetParamFloat

qeSetParamFloat sets the value of a single-precision floating-point parameter.

Syntax

```
int16  res_code qeSetParamFloat   (
    int16     hstmt,
    int16     param_num,
    float32   param_val)
```

Description

qeSetParamFloat assigns the value of a parameter in a SQL statement to a single-precision floating-point value.

DTK copies the assigned value, so the pointer need not remain valid after this call. This parameter has this value until qeClearParam or a qeSetParam or qeBindParam function is called again for this parameter. All parameters with the same name as the one identified by *param_num* are affected.

Before calling qeSetParamFloat, you must call qeSQLPrepare. You must give values to all parameters before calling qeSQLExecute.

Parameters

hstmt is the handle to the statement returned by qeSQLPrepare.

param_num is the position of the parameter to be set.

param_val is the single-precision floating-point value to be assigned to the parameter.

res_code is the result code returned by qeSetParamFloat, which returns the same set of result codes as qeErr. See Appendix D, “Result and Error Message Codes,” on page 537 for a list of these result codes.

Example

```
hdbc = qeConnect ("DSN=QEDBF")   ;
hstmt = qeSQLPrepare (hdbc, "SELECT * FROM emp
    WHERE salary = ?")  ;
res_code = qeSetParamFloat (hstmt, 1, 32000.00)     ;
res_code = qeSQLExecute (hstmt)   ;
res_code = qeEndSQL (hstmt)   ;
res_code = qeDisconnect (hdbc)   ;
```

qeSetParamInt

qeSetParamInt sets the value of a 2-byte integer parameter.

Syntax

```
int16 res_code qeSetParamInt   (
    int16    hstmt,
    int16    param_num,
    int16    param_val)
```

Description

qeSetParamInt assigns the value of a parameter in a SQL statement to a 2-byte integer value.

DTK copies the assigned value, so the pointer need not remain valid after this call. This parameter has this value until qeClearParam or a qeSetParam or qeBindParam function is called again for this parameter. All parameters with the same name as the one identified by *param_num* are affected.

Before calling qeSetParamInt, you must call qeSQLPrepare. You must give values to all parameters before calling qeSQLExecute.

Parameters

hstmt is the handle to the statement returned by qeSQLPrepare.

param_num is the position of the parameter to be set.

param_val is the 2-byte integer value to be assigned to the parameter.

res_code is the result code returned by qeSetParamInt, which returns the same set of result codes as qeErr. See Appendix D, “Result and Error Message Codes,” on page 537 for a list of these result codes.

Example

```
hdbc = qeConnect ("DSN=QEDBF")   ;
hstmt = qeSQLPrepare (hdbc, "SELECT * FROM emp
    WHERE salary = ?")  ;
res_code = qeSetParamInt (hstmt, 1, 32000)    ;
res_code = qeSQLExecute (hstmt)   ;
res_code = qeEndSQL (hstmt)   ;
res_code = qeDisconnect (hdbc)   ;
```

qeSetParamIOType

qeSetParamIOType sets a parameter's input/output (I/O) type.

Syntax

```
int16  res_code qeSetParamIOType   (
    int16    hstmt,
    int16    param_num,
    int16     type_flag)
```

Description

DTK applications should call qeSetParamIOType along with one of the qeBindParam or qeSetParam functions for each parameter in a SQL statement or stored procedure.

If qeSetParamIOType is not called for a parameter, the parameter is assumed to be an input parameter. An error is issued if the application tries to retrieve the output value from a parameter that has not been defined as either qePARAM_INOUT or qePARAM_OUTPUT with qeSetParamIOType.

Parameters

hstmt is the handle to the statement returned by qeSQLPrepare.

param_num is the position of the parameter to be set.

type_flag is a flag to indicate the parameter's IO type. The type flags are:

Constant	Value	Description
qePARAM_INPUT	2	Input parameter.
qePARAM_INOUT	3	Input/Output parameter.
qePARAM_OUTPUT	5	Output parameter.

res_code is the result code returned by the qeSetParamIOType function, which returns the same set of result codes as qeErr. See Appendix D, "Result and Error Message Codes," on page 537 for a list of these result codes.

Example

```
hdbc = qeConnect (“DSN=QEORA;DLG=2”)   ;
hstmt = qeSQLPrepare (hdbc, “{call GetDeptName(?)}”)    ;
char_len = 10 ;
res_code = qeBindParamChar (hstmt, 1, dept, &char_len)    ;
res_code = qeSetParamIOType (hstmt, 1, qePARAM_OUTPUT)    ;
res_code = qeSQLExecute (hstmt)  ;
    /* The value of ?DEPT_NAME is in the dept buffer*    /
res_code = qeEndSQL(hstmt)  ;
res_code = qeDisconnect (hstmt)  ;
```

qeSetParamLong

qeSetParamLong sets the value of a 4-byte integer parameter.

Syntax

```
int16 res_code qeSetParamLong   (
    int16     hstmt,
    int16     param_num,
    int32     param_val)
```

Description

qeSetParamLong assigns the value of a parameter in a SQL statement to a 4-byte integer value.

DTK copies the assigned value, so the pointer need not remain valid after this call. This parameter has this value until qeClearParam or a qeSetParam or qeBindParam function is called again for this parameter. All parameters with the same name as the one identified by *param_num* are affected.

Before calling qeSetParamLong, you must call qeSQLPrepare. You must give values to all parameters before calling qeSQLExecute.

Parameters

hstmt is the handle to the statement returned by qeSQLPrepare.

param_num is the position of the parameter to be set.

param_val is the 4-byte integer value to be assigned to the parameter.

res_code is the result code returned by qeSetParamLong, which returns the same set of result codes as qeErr. See Appendix D, “Result and Error Message Codes,” on page 537 for a list of these result codes.

Example

```
hdbc = qeConnect ("DSN=QEDBF")   ;
hstmt = qeSQLPrepare (hdbc, "SELECT * FROM emp
    WHERE salary = ?")  ;
res_code = qeSetParamLong (hstmt, 1, 32000)    ;
res_code = qeSQLExecute (hstmt)   ;
res_code = qeEndSQL (hstmt)   ;
res_code = qeDisconnect (hdbc)   ;
```

qeSetParamNull

qeSetParamNull sets the value of a parameter in a SQL statement to null.

Syntax

```
int16 res_code qeSetParamNull  (
    int16    hstmt,
    int16    param_num,
    int16    param_type,
    int32    precision,
    int16    scale)
```

Description

qeSetParamNull assigns a null value to a parameter in a SQL statement.

Before calling qeSetParamNull, you must call qeSQLPrepare. You must give values to all parameters before calling qeSQLExecute.

Parameters

hstmt is the handle to the statement returned by qeSQLPrepare.

param_num is the position of the parameter to be set.

param_type is the data type of the specified parameter. It can have one of the following values:

Constant	Value	Description
qeCHAR	1	Blank-padded, fixed-length string.
qeVARCHAR	2	Variable-length string.
qeDECIMAL	3	BCD number.
qeINTEGER	4	4-byte signed integer.
qeSMALLINT	5	2-byte signed integer.
qeFLOAT	6	4-byte floating-point number.
qeDOUBLEPRECISION	7	8-byte floating-point number.
qeDATETIME	8	26-byte date time value. Example: YYYY-MM-DD HH:MM:SS.FFFFFF

Constant	Value	Description
qeDATE	111	26-byte date value.
qeTIME	112	26-byte time value.
qeNO_DATA_TYPE	0	No data type.

You can specify qeNO_DATA_TYPE only if the specified parameter has already been assigned a data type by a previous call to a qeSetParam or qeBindParam function.

precision is a decimal value's precision, the maximum size of a character, or the length (in bytes) of a date-time value. This value is required only if applicable to the parameter being set to null.

scale is a decimal value's scale. This value is required only if applicable to the parameter being set to null.

res_code is the result code returned by qeSetParamNull, which returns the same set of result codes as qeErr. See Appendix D, "Result and Error Message Codes," on page 537 for a list of these result codes.

Example

```
hdbc = qeConnect ("DSN=QEDBF")   ;
hstmt = qeSQLPrepare (hdbc, "SELECT * FROM emp
    WHERE first_name = ?")  ;
res_code = qeSetParamNull (hstmt, 1, qeVARCHAR, 10, 0)     ;
res_code = qeSQLExecute (hstmt)   ;
res_code = qeEndSQL (hstmt)   ;
res_code = qeDisconnect (hdbc)   ;
```

qeSetParamTime

qeSetParamTime sets the value of a time parameter.

Syntax

```
int16  res_code qeSetParamTime   (
    int16     hstmt,
    int16     param_num,
    ptrstr    param_val)
```

Description

qeSetParamTime assigns the value of a parameter in a SQL statement to a 26-byte time value.

DTK copies the assigned value, so the pointer need not remain valid after this call. This parameter has this value until qeClearParam or a qeSetParam or qeBindParam function is called again for this parameter. All parameters with the same name as the one identified by *param_num* are affected.

Before calling qeSetParamTime, you must call qeSQLPrepare. You must give values to all parameters before calling qeSQLExecute.

Parameters

hstmt is the handle to the statement returned by qeSQLPrepare.

param_num is the position of the parameter to be set.

param_val is the 26-byte time value to be assigned to the parameter.

res_code is the result code returned by qeSetParamTime, which returns the same set of result codes as qeErr. See Appendix D, “Result and Error Message Codes,” on page 537 for a list of these result codes.

Example

```
hdbc = qeConnect ("DSN=QEDBF")   ;
hstmt = qeSQLPrepare (hdbc, "SELECT * FROM emp
    WHERE hire_date = ?")  ;
res_code = qeSetParamTime (hstmt, 1,
    "0000-00-00 03:14:12:000000")   ;
res_code = qeSQLExecute (hstmt)   ;
res_code = qeEndSQL (hstmt)   ;
res_code = qeDisconnect (hdbc)   ;
```

qeSetQueryTimeout

qeSetQueryTimeout sets the time to wait for a SQL statement to execute before aborting the query and returning to the application.

Syntax

```
int16 res_code qeSetQueryTimeout (int16  hdbc, int32
seconds)
```

Description

qeSetQueryTimeout sets the timeout for SQL statement execution.

This function depends on driver support, and has no effect if the driver does not support timeouts.

Parameters

hdbc is the handle to the connection returned by qeConnect.

seconds is how many seconds to wait. 0, the default, indicates that no timeout is to occur.

res_code is the result code returned by qeSetQueryTimeout, which returns the same set of result codes as qeErr. See Appendix D, “Result and Error Message Codes,” on page 537 for a list of these result codes.

Example

```
hdbc = qeConnect ("DSN=QESS")  ;
res_code = qeSetQueryTimeout (hdbc, 20)   ;
hstmt = qeExecSQL (hdbc, "SELECT * FROM emp")    ;
/* Query will fail if no response within 20 seconds. *     /
res_code = qeEndSQL (hstmt)  ;
```

See Also

qeGetQueryTimeout.

qeSetSelectOptions

qeSetSelectOptions sets options that determine which fetch commands and positioning behaviors are allowed.

Syntax

```
int16 res_code qeSetSelectOptions (int16   hdbc, int32
flags)
```

Description

qeSetSelectOptions lets you set options that affect fetching behavior during the current database connection. These options affect the level of fetching allowed in the current connection, whether logging is used when not made necessary by the database system, and the extent to which the result set persists after a transaction ends.

Parameters

hdbc is the handle to the database connection returned by qeConnect.

flags is a set of option flags that controls fetching and statement persistence behavior for the current connection. These values can be combined by adding them together or joining them with an OR clause. Possible values include the following:

Constant	Value	Description
qeFETCH_FORWARD_DIR	0x0001	Only forward fetching is allowed. This is the default fetching behavior option.
qeFETCH_ANY_DIR	0x0002	Random and previous fetching is enabled.
qeLOG_IF_NEEDED	0x0008	Use log file only as needed to enable previous and random fetching. This is the default logging behavior.
qeLOG_ALWAYS	0x0010	Force use of log file when it is not required. (This does not activate random fetching if it is not explicitly set with qeFETCH_ANY_DIR).

Constant	Value	Description
qeSELECT_INVALIDATE	0x0020	Disable fetching at the end of transaction (EOT). Calls made after a commit or rollback to any function except qeEndSQL cause an error.
qeSELECT_TRUNCATE	0x0040	Truncate the result set at EOT. This option lets you continue fetching only those records already read from the database (if qeFETCH_ANY_DIR is set).
qeSELECT_PERSIST	0x0060	The result set persists at EOT. This is the default behavior, which lets you continue fetching from the entire set of records returned by the Select statement. To enable this behavior for databases that invalidate the *hstmt* at commit or rollback, the records in the result set that have not been fetched by EOT are written to a log file.

res_code is the result code returned by qeSetSelectOptions, which returns the same set of result codes as qeErr. See Appendix D, “Result and Error Message Codes,” on page 537 for a list of these result codes.

Example

```
hdbc = qeConnect ("DSN=QESS")   ;
res_code = qeSetSelectOptions (hdbc, qeFETCH_ANY_DIR +
qeSELECT_PERSIST)  ;
hstmt = qeExecSQL (hdbc, "SELECT * FROM emp")    ;
...
/* Options affect behavior of this and future hstmts. *     /
...
res_code = qeEndSQL (hstmt)   ;
res_code = qeDisconnect (hdbc)   ;
```

qeSetSQL

qeSetSQL places a partial statement in the SQL buffer.

Syntax

```
int16 res_code qeSetSQL (int16  hdbc, ptrstr  partial_stmt)
```

Description

Some macro languages cannot send an entire SQL statement to qeExecSQL due to limits in the lengths of strings they support. For example, Excel strings are limited to 255 characters. Since many Select statements are longer than 255 characters, Excel cannot send long Select statements to qeExecSQL.

Internally, DTK maintains one SQL buffer per *hdbc*. qeSetSQL replaces the contents of the SQL buffer with the partial statement sent as a parameter. Each subsequent call to qeAppendSQL appends text to the SQL buffer. Once the complete SQL statement has been sent to the DTK API, you can call qeSQLPrepare (with "" as the *sql_stmt* value) or qeExecSQL to use the SQL statement saved in the SQL buffer.

Parameters

hdbc is the handle to the database connection returned by qeConnect.

partial_stmt is the character string that is to replace the contents of the SQL buffer. It must contain the first part of a SQL statement.

res_code is the result code returned by qeSetSQL, which returns the same set of result codes as qeErr. See Appendix D, "Result and Error Message Codes," on page 537 for a list of these result codes.

Example

To send a Select statement in pieces and execute it:

```
hdbc = qeConnect ("DSN=QESS;UID=sa;SRVR=PION1")     ;
...
res_code = qeSetSQL (hdbc, "SELECT *")    ;
res_code = qeAppendSQL (hdbc, " FROM emp")    ;
res_code = qeAppendSQL (hdbc, " ORDER BY last_name"     )
hstmt = qeExecSQL (hdbc, "")   ;
...
res_code = qeEndSQL (hstmt)   ;
res_code = qeDisconnect (hdbc)   ;
```

See Also

qeAppendSQL, qeExecSQL.

qeSetTableCaching

qeSetTableCaching controls whether table information is cached after calls to qeTables.

Syntax

```
int16 res_code qeSetTableCaching (int16  hdbc, int16
setting)
```

Description

qeSetTableCaching controls whether the results of qeTables calls are cached. It can take a noticeable amount of time to retrieve the names of all available tables via qeTables, so caching the table names in a file is a good idea if your application uses them repeatedly. Call qeSetCacheFileName to specifically name a file for table caching. If you do not, DTK stores the table names in a temporary file.

When caching is enabled, only the first call to qeTables returns table names from the database. All subsequent calls to qeTables read table names from the cache file. To reread tables from the database, either turn caching off or delete the cache file before calling qeTables.

You can call qeSetTableCaching to turn caching on for the current session, on for all sessions, or off for all sessions. If enabled for all sessions, the cache file is saved when the connection terminates so that it can be used again when needed. The first time you call this function to set caching to qeCACHE_PERMANENT (1), you must call qeSetCacheFileName to assign a name to the cache file. To reuse the cache file in another session, call qeSetCacheFileName to specify the existing file.

Important: Calling this function to turn caching off deletes the cache file.

Parameters

hdbc is the handle to the connection returned by qeConnect.

setting is one of the following:

Constant	Value	Description
qeCACHE_PERMANENT	1	Turn caching on, and have the cache file remain after the connection terminates. You must specify a file name with the qeSetCacheFileName function when using this option.
qeCACHE_SESSION	2	Turn caching on for this session. The cache file is deleted when the connection terminates. This is the default.
qeCACHE_OFF	3	Turn caching off.

res_code is the result code returned by qeSetTableCaching, which returns the same set of result codes as qeErr. See Appendix D, "Result and Error Message Codes," on page 537 for a list of these result codes.

Example

```
/* Cache_Session * /
hdbc = qeConnect ("DSN=QEDBF")   ;
res_code = qeSetTableCaching (hdbc, qeCACHE_SESSION)     ;
hstmt = qeTables (hdbc, "*", "*", qeTBL_TABLE)     ;
res_code = qeEndSQL (hstmt)   ;
res_code = qeDisconnect (hdbc)   ;
```

See Also

qeTables, qeSetCacheFileName, qeGetTableCaching.

qeSetTraceOptions

qeSetTraceOptions sets the type of information that is sent to the trace file.

Syntax

```
int16 res_code qeSetTraceOptions (int16   flags)
```

Parameters

flags is a set of option flags that defines which tracing options are enabled/disabled. These values can be combined by adding them together or joining them with an OR clause. *flags* can be:

Constant	Value	Description
qeTRACE_NON_VAL_CALLS	0x0001	Trace all non-qeVal calls.
qeTRACE_USER	0x0002	Trace strings sent via qeTraceUser.
qeTRACE_VAL_CALLS	0x0004	Trace qeVal calls and bound data at fetch time.
qeTRACE_WINDOW	0x0008	Write all trace information (except ODBC calls) to a trace window.
qeTRACE_ODBC	0x0010	Trace ODBC calls. Tracing is written to either SQL.LOG or another file that you have specified via the qeSetDriverTracefile function.
qeTRACE_NO_FLUSH	0x0020	Allows faster tracing by writing trace strings to disk in blocks instead of one at a time. Choosing this method can cause some loss of trace information if your program terminates abnormally—use it only when your application is reasonably stable.

The default when qeTraceOn is called is qeTRACE_NON_VAL_CALLS + qeTRACE_USER (0x0001 and 0x0002), unless the Trace section of the QELIB.INI file contains an Options entry.

res_code is the result code returned by qeSetTraceOptions, which returns the same set of result codes as qeErr. See Appendix D, “Result and Error Message Codes,” on page 537 for a list of these result codes.

Example

```
res_code = qeTraceOn ("\\trace.txt")    ;
res_code = qeSetTraceOptions (qeTRACE_NON_VAL_CALL     S
    + qeTRACE_VAL_CALLS)  ;
hdbc = qeConnect ("DSN=QEDBF")   ;
...
res_code = qeDisconnect (hdbc)   ;
res_code = qeTraceOff ()  ;
```

Notes

Calls to this function are not cumulative; only the options set in the last call are valid.

See Also

qeSetDriverTracefile, qeTraceUser.

qeSetupInfo and qeSetupInfoBuf

These functions return the information entered when DTK was installed.

Syntax

```
ptrstr  info qeSetupInfo ( )
```

```
int16  res_code qeSetupInfoBuf (ptrstr   info)
```

Description

qeSetupInfo and qeSetupInfoBuf return the user name, company name, and serial number entered the first time DTK was installed. The first time you run the DTK Setup program, you are prompted for this information.

When you use qeSetupInfo**,** the function returns a pointer to the string. The string is stored in a buffer maintained by DTK. Copy the string out of this buffer before you call another DTK function, because the next function may use the same buffer.

When you use qeSetupInfoBuf**,** you pass in a pointer to a buffer you have allocated. The string is put in the buffer. Make sure the buffer is large enough to hold the returned string.

Parameters

info is the character string containing the user name, company name, and serial number. A Tab character (9) separates the three values and a zero-terminator ends the string. The string may contain up to 128 characters of information.

res_code is the result code returned by qeSetupInfoBuf, which returns the same set of result codes as qeErr. See Appendix D, “Result and Error Message Codes,” on page 537 for a list of these result codes.

Example

To retrieve DTK setup information:

```
setup_info = qeSetupInfo ()   ;
```

qeSources

qeSources returns information on the database sources (systems) that can be accessed.

Syntax

```
int16 hstmt qeSources (int16  option)
```

Description

qeSources creates a statement execution (*hstmt*) that returns information on the database sources (systems) that can be accessed. qeSources returns one record per source. Each record contains the following columns:

Column	Type	Description
Name	Char(32)	Source name.
Extension	Char(32)	File extension. May be null.
DTK *hdbc*	Int16	If qeConnect has been used to connect to this source, the DTK *hdbc*. This is 0 if not currently connected.
Remarks	Char(256)	Comments (if available).

You retrieve this information like you would other database values—using the qeVal, qeBindCol, and qeFetch functions.

Parameters

option determines which sources are returned by the *hstmt* returned by qeSources. There is no default; *option* must contain one of the following values:

Constant	Value	Description
qeSRC_AVAIL_LOGON	1	All sources
qeSRC_CONN_LOGON	2	All connected sources

hstmt is the handle to the statement returned by qeSources.

Example

```
hdbc = qeConnect ("DSN=QEGUP;DLG=1")   ;
hstmt = qeSources (qeSRC_CONN_LOGON)   ;
while (qeFetchNext (hstmt) == qeSUCCESS)     {
    ...
/* Get info about available sources. *    /
    ...
}
res_code = qeDisconnect (hdbc)   ;
```

qeSQLExecute

qeSQLExecute executes a statement previously prepared with qeSQLPrepare, qeQBEPrepare, or qeQryPrepare.

Syntax

```
int16  res_code qeSQLExecute (int16   hstmt)
```

Description

qeSQLExecute executes a statement previously prepared with qeSQLPrepare, qeQBEPrepare, or qeQryPrepare.

This function is also useful for re-executing the active statement without re-parsing.

If the statement contains any parameters that have not been assigned values, qeSQLExecute prompts you for the values.

Parameters

hstmt is the handle to the statement returned by qeSQLPrepare, qeQBEPrepare, or qeQryPrepare.

res_code is the result code returned by qeSQLExecute, which returns the same set of result codes as qeErr. See Appendix D, “Result and Error Message Codes,” on page 537 for a list of these result codes.

Example

```
hdbc = qeConnect ("DSN=QEDBF")   ;
hstmt = qeSQLPrepare (hdbc, "SELECT * FROM emp
    WHERE       first_name = ?")  ;
res_code = qeSetParamChar (hstmt, 1, "Ed", 10)    ;
res_code = qeSQLExecute (hstmt)   ;
res_code = qeFetchNext (hstmt)   ;
res_code = qeEndSQL (hstmt)   ;
res_code = qeDisconnect (hdbc)   ;
```

qeSQLPrepare

qeSQLPrepare prepares a SQL statement for execution.

Syntax

```
int16 hstmt qeSQLPrepare (int16  hdbc, ptrstr  stmt)
```

Description

qeSQLPrepare returns an *hstmt* for a statement and places it in the statement buffer, but does not execute it. Call this function to get a handle for a statement on which you want to do additional processing before you execute it.

qeSQLPrepare is most useful for preparing statements that use parameters, although parameters do not have to be present to use it.

Routines that call this function must call qeSQLExecute to execute.

Parameters

hdbc is the handle to the connection returned from qeConnect.

stmt is a null-terminated character string representing a SQL statement. If *stmt* is null, then the routine uses the statement passed using qeSetSQL and qeAppendSQL.

hstmt is the handle to the statement returned by qeSQLPrepare.

Example

```
hdbc = qeConnect ("DSN=QEDBF")   ;
hstmt = qeSQLPrepare (hdbc, "SELECT * FROM emp
    WHERE    first_name = ?") ;
res_code = qeSetParamChar (hstmt, 1, "Ed", 10)    ;
hstmt = qeSQLExecute (hstmt)   ;
res_code = qeFetchNext (hstmt)   ;
res_code = qeEndSQL (hstmt)   ;
res_code = qeDisconnect (hdbc)   ;
```

qeTables

qeTables returns information on the available database tables.

Syntax

```
int16 hstmt qeTables (
    int16    hdbc,
    ptrstr   qualifier_pattern,
    ptrstr   user_pattern,
    ptrstr   table_pattern,
    int16    flags)
```

Description

qeTables creates a statement execution (*hstmt*) that returns information on the available database tables. qeTables returns one record per table. Each record contains the following columns:

Column	Type	Description
Table Qualifier	Char(128)	Qualifier for returned table.
Table User	Char(128)	A user name (for table-based sources) or directory name (for file-based sources).
Table Name	Char(128)	A table name (for table-based sources) or file name (for file-based sources).
Table Type	Int16	Type of table: qeTBL_TABLE, qeTBL_VIEW, qeTBL_SYNONYM, qeTBL_PROCEDURE, or qeTBL_SYSTABLE.
Remarks	Char(256)	Comments (if available).

You retrieve this information like you would other database values—using the qeVal, qeBindCol, and qeFetch functions.

It can take a noticeable amount of time to retrieve the names of all available tables via qeTables, so caching the table names in a file is a good idea if your application uses them repeatedly. You can call qeSetTableCaching to turn caching on for the current session, on for all sessions, or off for all sessions. You can specifically name a file to use for table caching by calling qeSetCacheFileName. If you do not, DTK stores them in a temporary file.

When caching is enabled, only the first call to qeTables returns table names from the database. All subsequent calls to qeTables read table names from the cache file. To reread tables from the database, either turn caching off or delete the cache file before calling qeTables.

Parameters

hdbc is a handle to a database connection obtained from qeConnect.

qualifier_pattern is a pointer to a string containing a qualifier or path for the set of tables to be selected.

user_pattern is the pattern used for selecting users. If the pattern is null, the current user is assumed. If the pattern is "%" or "*", all users are selected. This parameter is ignored for file-based databases, where the current working directory is assumed.

table_pattern is the pattern used for selecting tables or files. If the pattern is "%" or "*", all tables are selected.

flags is a set of option flags that specifies the types of tables to be returned. The value sent determines the types of items to be returned by the *hstmt*. These are also the values returned in the Type column.

flags has no default value; you must specify at least one of the following values:

Constant	Value	Description
qeTBL_TABLE	0x0001	Get table names.
qeTBL_VIEW	0x0002	Get view names.
qeTBL_PROCEDURE	0x0004	Get stored procedure names.
qeTBL_SYSTABLE	0x0008	Get system table names.
qeTBL_SYNONYM	0x0010	Get synonym names.
qeTBL_DATABASE	0x0080	Get database names.

Note: qeTBL_DATABASE cannot be combined with the other values. All other values can be combined by adding them together or joining them with an OR clause.

hstmt is the handle to the statement returned by qeTables.

Example

```
hdbc = qeConnect ("DSN=QEINF;DLG=1")   ;
hstmt = qeTables (hdbc, "%", "SYS%", "%", qeTBL_TABLE |
qeTBL_SYSTABLE) ;
while (qeFetchNext (hstmt) == qeSUCCESS)    {
    ...
/* Get info about tables. *  /
    ...
}
res_code = qeDisconnect (hdbc)   ;
```

qeTraceOff

qeTraceOff closes the trace file opened by qeTraceOn and discontinues the tracing of calls to the DTK API.

Syntax

```
int16 res_code qeTraceOff ( )
```

Parameters

res_code is the result code returned by qeTraceOff, which returns the same set of result codes as qeErr. See Appendix D, “Result and Error Message Codes,” on page 537 for a list of these result codes.

Example

```
res_code = qeTraceOn ("\\trace.txt")   ;
hdbc = qeConnect ("DSN=QEDBF")   ;
...
res_code = qeDisconnect (hdbc)   ;
res_code = qeTraceOff ()  ;
```

See Also

qeTraceOn.

qeTraceOn

qeTraceOn initiates tracing of DTK functions.

Syntax

```
int16 res_code qeTraceOn (ptrstr  file_pathname)
```

Description

qeTraceOn starts tracing calls to the DTK API by writing debugging information to a trace file. Tracing helps you debug programs that call the DTK API by writing a log of the function calls to the DTK API, as well as the parameters to each call, and the returned value.

The trace file is an ASCII text file that can be edited with Notepad or any other text editor.

DTK continues to write to the Trace file until you call qeTraceOff.

Parameters

file_pathname is the pathname to the trace file you want DTK to write to. It must be a valid pathname for the operating system you are using.

res_code is the result code returned by qeTraceOn, which returns the same set of result codes as qeErr. See Appendix D, “Result and Error Message Codes,” on page 537 for a list of these result codes.

Example

```
res_code = qeTraceOn ("\\trace.txt")   ;
hdbc = qeConnect ("DSN=QEDBF")   ;
...
res_code = qeDisconnect (hdbc)   ;
res_code = qeTraceOff ()  ;
```

Notes

Tracing can also be enabled by the QELIB.INI file. See Appendix F, “The QELIB.INI File,” on page 565 for more information.

See Also

qeTraceOff, qeSetTraceOptions.

qeTraceUser

qeTraceUser sends a user-defined string to the tracefile.

Syntax

```
int16 res_code qeTraceUser (ptrstr tracestring)
```

Parameters

tracestring is a string written to the trace file if qeSetTraceOptions is not called to disable such writing.

res_code is the result code returned by qeTraceUser, which returns the same set of result codes as qeErr. See Appendix D, "Result and Error Message Codes," on page 537 for a list of these result codes.

Example

```
res_code = qeTraceOn ("\\trace.txt")    ;
res_code = qeSetTraceOptions (qeTRACE_NON_VAL_CALL     S
    + qeTRACE_USER) ;
res_code = qeTraceUser ("This is the beginnin    g
    of the trace file.")  ;
hdbc = qeConnect ("DSN=QEDBF")   ;
...
res_code = qeDisconnect (hdbc)   ;
res_code = qeTraceUser ("This is the end of the
    trace file.") ;
res_code = qeTraceOff ()  ;
```

Notes

This function is useful only when tracing is on and user string tracing (the default) is enabled by a call to qeSetTraceOptions.

qeTypeInfo

qeTypeInfo returns information about the data types supported by a database.

Syntax

```
int16 hstmt qeTypeInfo (int16  hdbc)
```

Description

qeTypeInfo creates a statement execution (*hstmt*) that returns information about the types supported on a particular database. The resulting records contain the following columns:

Column	Type	Description
Type Name	Char(128)	Data source-dependent data type name.
Type	Int16	DTK type.
DB Type	Int16	Database type.
Width	Int32	Size of type in bytes.
Attr1	Int16	Precision for decimal types, date start position for dates, null otherwise.
Attr2	Int16	Scale for decimal types, date end position for dates, null otherwise.
Literal Prefix	Char(128)	Characters used to prefix a literal. Null if not applicable.
Literal Suffix	Char(128)	Characters used to terminate a literal. Null if not applicable.
Create Params	Char(128)	The parameters necessary to use the type in a Create Table statement (for Decimal, this would be “precision,scale”).
Nullable	Int16	Whether type can be null. Values: qeCOL_NULLABLE, qeCOL_NOT_NULLABLE, qeCOL_UNKNOWN.

Column	Type	Description
Case Sensitive	Int16	Whether type can be treated as case sensitive for sorting (T/F).
Searchable	Int16	How the type can be used in a WHERE clause. Values: qeCOL_UNSEARCHABLE, qeCOL_LIKE_ONLY, qeCOL_ALL_EXCEPT_LIKE, qeCOL_SEARCHABLE.
Unsigned	Int16	Whether type is unsigned (T/F). Null if not applicable.
Money	Int16	Whether type is a money data type (T/F).
Auto Increment	Int16	Whether type is auto incrementing. Null if not applicable (T/F).
Local Type Name	Char(128)	Localized version of the data source-dependent name of the data type. Null if not supported by the data source.

You retrieve this information like you would other database values—using the qeVal, qeBindCol, and qeFetch functions.

Parameters

hdbc is a handle to a database connection obtained from qeConnect.

hstmt is the handle to the statement returned by qeTypeInfo.

Example

```
hdbc = qeConnect ("DSN=QEINF;DLG=1")   ;
hstmt = qeTypeInfo (hdbc)  ;
while (qeFetchNext (hstmt) == qeSUCCESS)    {
    ...
/* Get info about types. *  /
    ...
}
res_code = qeDisconnect (hdbc)   ;
```

qeUndoAll

qeUndoAll discards all changes to a statement that have not been sent to the database.

Syntax

```
int16 res_code qeUndoAll (int16  hstmt)
```

Description

When qeSetAutoUpdate is set to qeAUTOUPD_DEFER(2) to cause record changes to be deferred (that is, saved but not updated in the database), qeUndoAll discards all record changes performed on the statement but not applied to the database. The changes discarded include any saved changes on records other than the current record, any unsaved records created by calling qeRecNew, and any new values placed in the current record by calls to qePut functions.

Parameters

hstmt is the handle to the statement returned by qeExecSQL or qeSQLPrepare.

res_code is the result code returned by qeUndoAll, which returns the same set of result codes as qeErr. See Appendix D, “Result and Error Message Codes,” on page 537 for a list of these result codes.

Example

```
hdbc = qeConnect ("DSN=QEDBF")   ;
res_code = qeSetAutoUpdate (hdbc, qeAUTOUPD_DEFER)    ;
hstmt = qeExecSQL (hdbc, "SELECT * FROM emp")    ;

res_code = qeFetchNext (hstmt)   ;
res_code = qePutChar (hstmt, 1, "", "Rachel")    ;
res_code = qeFetchNext (hstmt)   ;
res_code = qePutChar (hstmt, 1, "", "Eddie")    ;
res_code = qeFetchNext (hstmt)   ;

res_code = qeUndoAll (hstmt)   ;
res_code = qeEndSQL (hstmt)   ;
res_code = qeDisconnect (hdbc)   ;
```

qeUniqueWhereClause and qeUniqueWhereClauseBuf

These functions return a Where clause that attempts to uniquely identify the current record in an active Select statement.

Syntax

```
ptrstr  where_clause qeUniqueWhereClause (int16   hstmt)

int16  res_code qeUniqueWhereClauseBuf   (
    int16    hstmt,
    ptrstr   clause_buf)
```

Description

qeUniqueWhereClause and qeUniqueWhereClauseBuf return the Where clause being used to identify the current record in an active Select statement. The Where clause attempts to uniquely identify the current record on calls to qeRecUpdate, qeRecDelete, and qeRecLock.

These functions use the columns specified by qeRecSetKey if that function is called. If no columns are specified as a primary key, DTK chooses a key that includes all appropriate columns. For most databases, this includes all searchable, non-character columns and character columns that are not over 256 bytes long.

qeUniqueWhereClause returns a pointer to the Where clause string. This string is stored in a buffer maintained by DTK. You must copy the string out of this buffer before you call another DTK function, because the next function may use the same buffer.

With qeUniqueWhereClauseBuf, you pass in a pointer to a buffer you have allocated. The Where clause string is put in the buffer. You must make sure that the buffer is large enough to hold the returned string.

If you are not currently positioned on a record, these functions return null.

Calling these functions causes DTK to generate a unique key if you have not already defined one with qeRecSetKey.

Parameters

hstmt is the handle to the statement returned by qeExecSQL or qeSQLPrepare.

clause_buf points to an allocated buffer for the resulting clause.

res_code is the result code returned by qeUniqueWhereClauseBuf, which returns the same set of result codes as qeErr. See Appendix D, "Result and Error Message Codes," on page 537 for a list of these result codes.

Example

```
hdbc = qeConnect ("DSN=QEDBF")   ;
hstmt = qeExecSQL (hdbc, "SELECT * FROM emp")    ;
res_code = qeFetchNext (hstmt)   ;
...
unique = qeUniqueWhereClause (hstmt)    ;
...
res_code = qeEndSQL (hstmt)   ;
res_code = qeDisconnect (hdbc)   ;
```

qeValChar and qeValCharBuf

These functions return a column value as a character string.

Syntax

```
ptrstr char_val qeValChar  (
    int16    hstmt,
    int16    col_num,
    ptrstr   fmt_string,
    int16    max_len)

int16 res_code qeValCharBuf  (
    int16    hstmt,
    ptrstr   char_val,
    int16    col_num,
    ptrstr   fmt_string,
    int16    max_len)
```

Description

qeValChar and qeValCharBuf return the value of a column in the current record as a character string. If the data type of the column is not a character string, the value is converted to a character string.

When you use qeValChar, the function returns a pointer to the string. The string is stored in a buffer maintained by DTK. Copy the string out of this buffer before you call another DTK function, because the next function may use the same buffer.

When you use qeValCharBuf, you pass a pointer to a buffer you have allocated. The string is put in the buffer. Make sure the buffer is large enough to hold the returned string.

Format number and date values by providing a format string (see "Format Strings" on page 59).

If the data type of the column is a character string (type 1 or 2), you may specify the maximum length of data to be returned.

Parameters

hstmt is the handle to the statement returned by qeExecSQL or qeSQLPrepare.

col_num is the column number whose value is to be returned. The first column number is 1.

fmt_string is the format string. If the column's data type is numeric or a date-time type (types other than 1 or 2), the format string specifies how to format the value when converting it to a character string. If no format string is given, "GN" is used for numbers and "GD" is used for date-time values.

max_len is the maximum number of characters that are to be returned if the column's data type is character string (type 1 or 2). If *max_len* is zero, the entire string is returned (up to 1000 characters). If *max_len* is not zero and the column's data type is not 1 or 2, an error is returned.

max_len is typically used either because your macro language limits the size of a character string that is less than the size of the values in the database, or because the database values are very large and you want to retrieve only part of the value.

For *max_len* values greater than zero, the actual limit is a little less than 64K (65280 bytes or characters, to be exact). However, if this is not sufficient for your needs, you can make multiple calls to qeValChar and retrieve the value in pieces.

If you use a non-zero *max_len* value to retrieve part of a value, you can call qeValChar again on the same column to retrieve more of the value. For example, you can retrieve a 4000-character value 500 characters at a time by calling qeValChar 8 times, each time setting *max_len* to 500. See "Blobs and Memos" on page 57 for more information.

If you specify a *max_len* of zero, qeValChar returns the entire value with an upper limit of 1000 characters. If the value is longer than 1000 characters, you receive only the first 1000 characters. Call qeValChar again to get the second 1000 characters.

char_val is the returned character value.

res_code is the result code returned by qeValCharBuf, which returns the same set of result codes as qeErr. See Appendix D, “Result and Error Message Codes,” on page 537 for a list of these result codes.

Example

To get the values of the first column for every record in the dBASE employee file:

```
hdbc = qeConnect ("DSN=QEDBF")   ;
...
hstmt = qeExecSQL (hdbc, "SELECT * FROM emp")    ;
while (qeFetchNext (hstmt) == 0)    {
    value = qeValChar (hstmt,1,"",0)   ;
    val_len = qeDataLen (hstmt)   ;
...
}
res_code = qeEndSQL (hstmt)   ;
res_code = qeDisconnect (hdbc)   ;
```

Notes

Column values must be retrieved starting with the first column and stepping through the columns in order (column 1, 2, 3, etc.). You can skip columns, but you cannot get a previous column’s value. You cannot retrieve the same column’s value more than once unless *max_len* is used to retrieve part of a value.

These functions add a zero byte to the end of each character string value. This is the C convention and is supported by most macro languages. Following a call to a qeVal function, qeDataLen returns the actual number of characters returned (not counting the zero byte). If the column value is null, qeDataLen returns qeNULL_DATA (-2). If the entire column value is not returned by qeValChar, qeDataLen indicates that the value was truncated by returning qeTRUNCATION (-1). This occurs if a non-zero *max_len* is specified and the length of the column value is greater than *max_len,* or if a zero *max_len* is specified and the length of the column value is greater than 1000 characters.

Following a call to a qeVal function, qeWarning also returns qeNULL_DATA (-2) if the column value is null, or qeTRUNCATION (-1) if the column value is truncated.

See Also

qeDataLen, qeWarning, qeValMultiChar and qeValMultiCharBuf, qeGetParamChar and qeGetParamCharBuf.

qeValDecimal and qeValDecimalBuf

These functions return a column value as a decimal number.

Syntax

```
ptrstr  dec_val qeValDecimal  (
    int16    hstmt,
    int16    col_num,
    int16    precision,
    int16    scale)

int16  res_code qeValDecimalBuf   (
    int16    hstmt,
    ptrstr   dec_val,
    int16    col_num,
    int16    precision,
    int16    scale)
```

Description

When you use qeValDecimal**,** the function returns a pointer to the value. The value is stored in a buffer maintained by DTK. Copy the value out of this buffer before you call another DTK function, because the next function may use the same buffer.

When you use qeValDecimalBuf**,** you pass in a pointer to a buffer you have allocated. The value is put in the buffer. Make sure the buffer is large enough to hold the returned value.

qeValDecimal and qeValDecimalBuf return the value of a column in the current record as a decimal number. If the data type of the column is not decimal number, the value is converted to a decimal number (type 3).

If the column's data type is character string (type 1 or 2) and the column's value is not a number, the value 0 is returned.

Parameters

hstmt is the handle to the statement returned by qeExecSQL or qeSQLPrepare.

col_num is the column number whose value is to be returned. The first column number is 1.

precision is the total number of digits to be returned in the decimal value.

scale is the number of digits right of the decimal point to be returned in the decimal value.

dec_val is the returned decimal value.

res_code is the result code returned by qeValDecimalBuf, which returns the same set of result codes as qeErr. See Appendix D, “Result and Error Message Codes,” on page 537 for a list of these result codes.

Example

To get the values of the SALARY column for every record in the dBASE employee file:

```
hdbc = qeConnect ("DSN=QEDBF")   ;
...
hstmt = qeExecSQL (hdbc, "SELECT salary FROM emp")    ;
while (qeFetchNext (hstmt) == 0)    {
    value = qeValDecimal (hstmt,1,10,2)   ;
...
}
res_code = qeEndSQL (hstmt)   ;
res_code = qeDisconnect (hdbc)   ;
```

Notes

Column values must be retrieved starting with the first column and stepping through the columns in order (column 1, 2, 3, etc.). You can skip columns, but you cannot get a previous column’s value. You cannot retrieve the same column’s value more than once.

Values are formatted using the Binary Coded Decimal (BCD) format. This format is described in “Decimal Number Format” on page 55.

Since most macro languages do not support the BCD format, you may find it more convenient to retrieve decimal columns as floating-point numbers using qeValFloat or qeValDouble, or as character strings using qeValChar or qeValCharBuf.

Following a call to a qeVal function, qeDataLen returns the actual number of bytes returned. If the column value is null, qeDataLen returns qeNULL_DATA (-2).

Following a call to a qeVal function, qeWarning also returns qeNULL_DATA (-2) if the column value is null.

See Also

qeDataLen, qeWarning, qeGetParamDecimal and qeGetParamDecimalBuf.

qeValDouble

qeValDouble returns a column's value as a double-precision floating-point number.

Syntax

```
float64 dbl_val qeValDouble (int16  hstmt, int16  col_num)
```

Description

qeValDouble returns the value of a column in the current record as double-precision floating-point. If the data type of the column is not double-precision floating-point (type 7), the value is converted to this data type.

If the column's data type is character string (type 1 or 2) and the column's value is not a number, the value 0 is returned.

Parameters

hstmt is the handle to the statement returned by qeExecSQL or qeSQLPrepare.

col_num is the column number whose value is to be returned. The first column number is 1.

dbl_val is the returned value.

Example

To get the values of the SALARY column for every record in the dBASE employee file:

```
hdbc = qeConnect ("DSN=QEDBF")   ;
...
hstmt = qeExecSQL (hdbc, "SELECT salary FROM emp")    ;
while (qeFetchNext (hstmt) == 0)    {
    value = qeValDouble (hstmt,1)   ;
...
}
res_code = qeEndSQL (hstmt)   ;
res_code = qeDisconnect (hdbc)   ;
```

Notes

Column values must be retrieved starting with the first column and stepping through the columns in order (column 1, 2, 3, etc.). You can skip columns, but you cannot get a previous column's value. You cannot retrieve the same column's value more than once.

Following a call to a qeVal function, qeDataLen returns the actual number of bytes returned. If the column value is null, qeDataLen returns qeNULL_DATA (-2).

Following a call to a qeVal function, qeWarning also returns qeNULL_DATA (-2) if the column value is null.

See Also

qeDataLen, qeWarning, qeGetParamDouble.

qeValFloat

qeValFloat returns a column's value as a floating-point number.

Syntax

```
float32  flt_val qeValFloat (int16  hstmt, int16  col_num)
```

Description

qeValFloat returns the value of a column in the current record as floating-point. If the data type of the column is not floating-point (type 6), the value is converted to this data type.

If the column's data type is character string (type 1 or 2) and the column's value is not a number, the value 0 is returned.

Parameters

hstmt is the handle to the statement returned by qeExecSQL or qeSQLPrepare.

col_num is the column number whose value is to be returned. The first column number is 1.

flt_val is the returned value.

Example

To get the values of the SALARY column for every record in the dBASE employee file:

```
hdbc = qeConnect ("DSN=QEDBF")   ;
...
hstmt = qeExecSQL (hdbc, "SELECT salary FROM emp")     ;
while (qeFetchNext (hstmt) == 0)    {
    value = qeValFloat (hstmt,1)   ;
...
}
res_code = qeEndSQL (hstmt)   ;
res_code = qeDisconnect (hdbc)   ;
```

Notes

Column values must be retrieved starting with the first column and stepping through the columns in order (column 1, 2, 3, etc.). You can skip columns, but you cannot get a previous column's value. You cannot retrieve the same column's value more than once.

Following a call to a qeVal function, qeDataLen returns the actual number of bytes returned. If the column value is null, qeDataLen returns qeNULL_DATA (-2).

Following a call to a qeVal function, qeWarning also returns qeNULL_DATA (-2) if the column value is null.

See Also

qeDataLen, qeWarning, qeGetParamFloat.

qeValInt

qeValInt returns a column's value as a 2-byte integer.

Syntax

```
int16 int_val qeValInt (int16  hstmt, int16  col_num)
```

Description

qeValInt returns the value of a column in the current record as a 2-byte integer. If the data type of the column is not 2-byte integer (type 5), the value is converted to this data type.

If the column's data type is character string (type 1 or 2) and the column's value is not a number, the value 0 is returned.

Parameters

hstmt is the handle to the statement returned by qeExecSQL or qeSQLPrepare.

col_num is the column number whose value is to be returned. The first column number is 1.

int_val is the returned value.

Example

To get the values of the SALARY column for every record in the dBASE employee file:

```
hdbc = qeConnect ("DSN=QEDBF")   ;
...
hstmt = qeExecSQL (hdbc, "SELECT salary FROM emp")     ;
while (qeFetchNext (hstmt) == 0)    {
    value = qeValInt (hstmt,1)   ;
...
}
res_code = qeEndSQL (hstmt)   ;
res_code = qeDisconnect (hdbc)   ;
```

Notes

Column values must be retrieved starting with the first column and stepping through the columns in order (column 1, 2, 3, etc.). You can skip columns, but you cannot get a previous column's value. You cannot retrieve the same column's value more than once.

Following a call to a qeVal function, qeDataLen returns the actual number of bytes returned. If the column value is null, qeDataLen returns qeNULL_DATA (-2).

Following a call to a qeVal function, qeWarning also returns qeNULL_DATA (-2) if the column value is null.

See Also

qeDataLen, qeWarning, qeGetParamInt.

qeValLong

qeValLong returns a column's value as a 4-byte integer.

Syntax

```
int32 long_val qeValLong (int16  hstmt, int16 col_num)
```

Description

qeValLong returns the value of a column in the current record as a 4-byte integer. If the data type of the column is not a 4-byte integer (type 4), the value is converted to this data type.

If the column's data type is character string (type 1 or 2) and the column's value is not a number, the value 0 is returned.

Parameters

hstmt is the handle to the statement returned by qeExecSQL or qeSQLPrepare.

col_num is the column number whose value is to be returned. The first column number is 1.

long_val is the returned value.

Example

To get the values of the SALARY column for every record in the dBASE employee file:

```
hdbc = qeConnect ("DSN=QEDBF")   ;
...
hstmt = qeExecSQL (hdbc, "SELECT salary FROM emp")    ;
while (qeFetchNext (hstmt) == 0)    {
    value = qeValLong (hstmt,1)   ;
...
}
res_code = qeEndSQL (hstmt)   ;
res_code = qeDisconnect (hdbc)   ;
```

Notes

Column values must be retrieved starting with the first column and stepping through the columns in order (column 1, 2, 3, etc.). You can skip columns, but you cannot get a previous column's value. You cannot retrieve the same column's value more than once.

Following a call to a qeVal function, qeDataLen returns the actual number of bytes returned. If the column value is null, qeDataLen returns qeNULL_DATA (-2).

Following a call to a qeVal function, qeWarning also returns qeNULL_DATA (-2) if the column value is null.

See Also

qeDataLen, qeWarning, qeGetParamLong.

qeValMultiChar and qeValMultiCharBuf

These functions return the values of multiple columns as a single character string.

Syntax

```
ptrstr val qeValMultiChar (
    int16    hstmt,
    int16    start_col_num,
    int16    end_col_num,
    ptrstr   num_fmt_string,
    ptrstr   date_fmt_string,
    ptrstr   separator)

int16 res_code qeValMultiCharBuf (
    int16    hstmt,
    ptrstr   val,
    int16    start_col_num,
    int16    end_col_num,
    ptrstr   num_fmt_string,
    ptrstr   date_fmt_string,
    ptrstr   separator)
```

Description

qeValMultiChar and qeValMultiCharBuf return the values of several columns in the current record as a single character string. Each column value is separated by a character you specify, typically either Tab (9) or comma (,).

If the data type of the column is not character string, the value is converted to a character string. Number and date values are formatted by providing a format string (see "Format Strings" on page 59).

When you use qeValMultiChar**,** the function returns a pointer to the string. The string is stored in a buffer maintained by DTK. Copy the string out of this buffer before you call another DTK function, because the next function may use the same buffer.

When you use qeValMultiCharBuf**,** you pass in a pointer to a buffer you have allocated. The string is put in the buffer. Make sure the buffer is large enough to hold the returned string.

You can combine the use of qeValChar and qeValMultiChar to retrieve the values in a record. For example, you can call qeValChar to get the value of column 1, then call qeValMultiChar to retrieve the values of columns 2–4. You should mix calls to qeValChar and qeValMultiChar in the following situations:

- Two or more numeric (or date) columns are in the record and you want to use different format strings for each column. You can specify only one numeric (or date) format for each call to qeValMultiChar.
- One or more columns may contain character strings whose length is greater than 1000 characters. qeValMultiChar truncates column values to 1000 characters. To retrieve larger character strings, use qeValChar with a non-zero *max_len* parameter.
- Your macro language has a limit on the size of character strings, and the sum of the sizes of the columns in the record exceeds this limit.

Parameters

hstmt is the handle to the statement returned by qeExecSQL or qeSQLPrepare.

start_col_num is the column number of the first column whose value is to be returned. Column 1 is the first column.

end_col_num is the column number of the last column whose value is to be returned. Column 1 is the first column.

num_fmt_string is the format string to be used to convert all numeric columns to character strings. If no format string is given, “GN” is used.

date_fmt_string is the format string to be used to convert all date-time columns to character strings. If no format string is given, “GD” is used.

separator is the character to be used to separate the column values in the resulting string.

val is the returned character string containing the values of the specified columns. The last value is followed by a zero rather than a separator character.

res_code is the result code returned by qeValMultiCharBuf, which returns the same set of result codes as qeErr. See Appendix D, "Result and Error Message Codes," on page 537 for a list of these result codes.

Example

To get the FIRST_NAME, LAST_NAME, HIRE_DATE, and SALARY values, separated by Tab characters, for every record in the EMP file:

```
hdbc = qeConnect ("DSN=QEDBF")   ;
...
hstmt = qeExecSQL (hdbc, "SELECT first_name, last_name,
hire_date, salary FROM emp")   ;
while (qeFetchNext (hstmt) == 0)    {
    value = qeValMultiChar (hstmt, 1, 4, "", ""    ,
             "\x09") ;
    /* value points to the string *   /
    /* containing four values *   /
    /* separated by Tabs and zero- *   /
    /* terminated. */
    val_len = qeDataLen (hstmt)   ;
    /* val_len is the length of the *   /
    /* entire string. *  /
    ...
}
res_code = qeEndSQL (hstmt)   ;
res_code = qeDisconnect (hdbc)   ;
```

Notes

Column values must be retrieved starting with the first column and stepping through the columns in order (column 1, 2, 3, etc.). You can skip columns, but you cannot get a previous column's value. You cannot retrieve the same column's value more than once.

These functions add a zero byte to the end of the string. This is the C convention and is supported by most macro languages. Following a call to a qeVal function, qeDataLen returns the actual number of characters returned (including the separator characters but not counting the zero byte).

Unlike qeValChar, you cannot determine if an individual column value was null or truncated by checking if qeDataLen returns qeNULL_DATA (-2) or qeTRUNCATION (-1). qeDataLen never returns these values when calling qeValMultiChar since multiple values are returned in the string.

These functions are very similar to qeValChar and qeValCharBuf. They functions provide better performance if the records you are retrieving contain many columns.

See Also

qeDataLen, qeErr, qeValChar and qeValCharBuf.

qeVerNum and qeVerNumBuf

qeVerNum and qeVerNumBuf return the DTK version number that you are using.

Syntax

```
ptrstr  ver_num qeVerNum ( )
```

```
int16  res_code qeVerNumBuf (ptrstr   ver_num)
```

Description

When you use qeVerNum**,** the function returns a pointer to the string. The string is stored in a buffer maintained by DTK. Copy the string out of this buffer before you call another DTK function, because the next function may use the same buffer.

When you use qeVerNumBuf**,** you pass in a pointer to a buffer you have allocated. The string is put in the buffer. Make sure the buffer is large enough to hold the returned string.

Parameters

ver_num is the DTK version number returned as a zero-terminated character string.

res_code is the result code returned by qeVerNumBuf, which returns the same set of result codes as qeErr. See Appendix D, “Result and Error Message Codes,” on page 537 for a list of these result codes.

Example

To retrieve the DTK version number:

```
ver_num = qeVerNum ()  ;
```

qeWarning

qeWarning returns the warning generated by the last DTK function you called. It is usually called after qeErr to determine if the database system or the last function called returned any warnings.

Syntax

```
int16 res_code qeWarning ( )
```

Parameters

res_code is the result code returned by qeWarning. It is either a warning code returned by the database system or one of the following values:

Constant	Value	Description
qeLOCK_CHANGE_REC	-8	A lock was obtained, but the record has been changed since it was originally read. (This can occur only for database systems that require a log file.)
qeLOCK_MULTI_REC	-7	A lock was obtained, but more than one record was locked. This occurred because the primary key fields caused more than one record to be selected.
qeNULL_DATA	-2	A qeVal function returned a null value. Also returned as the length from a qeDataLen call.
qeTRUNCATION	-1	A qeVal function truncated the returned value because the value's size exceeded the buffer.

Example

```
hdbc = qeConnect ("DSN=QEDBF")   ;
if ((qeErr() == qeSUCCESS) && (qeWarning() == qeSUCCESS))
{
    hstmt = qeExecSQL (hdbc, "SELECT * FROM emp")    ;
    res_code = qeEndSQL (hstmt)   ;
    res_code = qeDisconnect (hdbc)   ;
}
```

See Also

qeErr, qeDBErr.

Go To

Part 3: Appendixes

A Data Conversion Functions

Data conversion functions let you convert values from any data type that DTK supports to any other data type. For example, you can convert long integers to floating-point values. These functions are not tied to a database connection or SQL statement execution. You can call these functions even if you are not using the Database functions described in the previous section.

When converting values to or from character strings, you may specify a format string. When converting to character strings, the format string controls the format of the resulting string. When converting from character strings, the format string gives the format of the character string value to be converted.

Errors may be detected when converting values. Use the qeErr, qeErrMsg, and qeErrMsgBuf functions to determine if any errors have occurred.

Converting Hexadecimal Values to Binary

qeHexToBin and qeHexToBinBuf convert a hexadecimal value into a binary value and place the result in a buffer.

Syntax

```
ptrstr  bin_value qeHexToBin  (
    ptrstr   hex_value,
    int32    length)

int16  res_code qeHexToBinBuf  (
    ptrstr   bin_value,
    ptrstr   hex_value,
    int32    length)
```

Description

qeHexToBin and qeHexToBinBuf convert a hexadecimal value into a binary value, and place the result in a buffer. The buffer must be at least half the size of the hexadecimal value.

qeHexToBin returns a pointer to the binary value. This value is stored in a buffer maintained by DTK. You must copy the value out of this buffer before you call another DTK function, because the next function may use the same buffer.

With qeHexToBinBuf, you pass in a pointer to a buffer you have allocated. The binary value is put in the buffer. You must make sure that the buffer is large enough to hold the returned value.

Parameters

bin_value points to a buffer allocated by the user to accept the converted hexadecimal value. It must be at least length/2 bytes long.

hex_value points to a string of *length* bytes of hexadecimal data. It is not a null-terminated string.

length is the length of the binary string that *hex_value* points to.

res_code is the result code returned by qeHexToBinBuf, which returns the same set of result codes as qeErr. See Appendix D for a list of these result codes.

Example

```
bin_val = qeHexToBin ("0A32B16F1A1A", 12)    ;
```

Converting to Character Strings

These functions convert a value from any of DTK's data types to a character string. You can specify a format string to control the string formatting. The format of decimal numbers is described in "Format Strings" on page 59.

Because these functions return a pointer, they have two forms (see "Parameter Conventions" on page 151). The names are identical, except one is appended with "Buf." In the first form listed, the function returns a pointer to the string. The string is stored in a buffer maintained by DTK. You must copy the string out of this buffer before you call another DTK function, because the next function may use the same buffer.

In the second form listed, appended with "Buf", you pass in a pointer to a buffer you have allocated. The string is put in the buffer. You must make sure that the buffer is large enough to hold the returned string.

Syntax

Converting from Date:

```
ptrstr  char_val qeDateToChar   (
    ptrstr   date_val,
    ptrstr   fmt_string)

int16  res_code qeDateToCharBuf   (
    ptrstr   char_val,
    ptrstr   date_val,
    ptrstr   fmt_string)
```

Converting from Decimal:

```
ptrstr  char_val qeDecimalToChar   (
    ptrstr   dec_val,
    int16    precision,
    int16     scale,
    ptrstr   fmt_string)
```

```
int16  res_code qeDecimalToCharBuf   (
    ptrstr   char_val,
    ptrstr   dec_val,
    int16    precision,
    int16    scale,
    ptrstr   fmt_string)
```

Converting from Double:

```
ptrstr  char_val qeDoubleToChar   (
    float64  dbl_val,
    ptrstr   fmt_string)
```

```
int16  res_code qeDoubleToCharBuf   (
    ptrstr   char_val,
    float64  dbl_val,
    ptrstr   fmt_string)
```

Converting from Float:

```
ptrstr  char_val qeFloatToChar   (
    float32  flt_val,
    ptrstr   fmt_string)
```

```
int16  res_code qeFloatToCharBuf   (
    ptrstr   char_val,
    float32  flt_val,
    ptrstr   fmt_string)
```

Converting from Int:

```
ptrstr  char_val qeIntToChar   (
    int16    int_val,
    ptrstr   fmt_string)
```

```
int16  res_code qeIntToCharBuf   (
    ptrstr   char_val,
    int16    int_val,
    ptrstr   fmt_string)
```

Converting from Long:

```
ptrstr  char_val qeLongToChar   (
    int32     long_val,
    ptrstr    fmt_string)
```

```
int16  res_code qeLongToCharBuf   (
    ptrstr    char_val,
    int32     long_val,
    ptrstr    fmt_string)
```

Parameters

char_val is the returned character string value.

fmt_string is the format string (see “Format Strings” on page 59). If no format string is given, numbers are formatted using GN, and dates are formatted using GD.

date_val, *dec_val*, *dbl_val*, *flt_val*, *int_val*, and *long_val* are the values to be converted.

res_code is the result code returned by the function, which returns the same set of result codes as qeErr. See Appendix D, “Result and Error Message Codes,” on page 537 for a list of these result codes.

precision is the number of significant digits when converting from a decimal number.

scale specifies the location of the decimal point when converting from a decimal number.

Example

To convert 125.3 to a character string formatted as money:

```
string1 = qeDoubleToChar (125.3, "$#,##0.00")    ;
```

To convert a Julian date value to a formatted date:

```
string1 = qeDateToChar (jul, "mm/dd/yyyy")    ;
```

Converting Character Strings to Date Values

These functions convert a character string into a standard date value using format strings that you specify.

Syntax

```
ptrstr  date_value qeCharToDate   (
    ptrstr   char_value,
    ptrstr   fmt_string)

int16  res_code qeCharToDateBuf   (
    ptrstr   date_value,
    ptrstr   char_value,
    ptrstr   fmt_string)
```

Description

qeCharToDate and qeCharToDateBuf convert a character string formatted using the format string into a standard date value using the specified format string.

qeCharToDate returns a pointer to the date value. This value is stored in a buffer maintained by DTK. You must copy the value out of this buffer before you call another DTK function, because the next function may use the same buffer.

With qeCharToDateBuf, you pass in a pointer to a buffer you have allocated. The date value is put in the buffer. You must make sure that the buffer is large enough to hold the returned value.

Parameters

date_value points to a buffer allocated by the user to accept the converted character value.

char_value points to the formatted character value to convert. If this character value is null, the function returns a date value of “01/01/94.”

fmt_string is the string used to format the value pointed to by *char_val*.

res_code is the result code returned by qeCharToDateBuf, which returns the same set of result codes as qeErr. See Appendix D, "Result and Error Message Codes," on page 537 for a list of these result codes.

Example

```
date_val = qeCharToDate (date_string, "mm/dd/yyyy")    ;
```

Converting to Decimal Numbers

These functions convert a value from any of DTK's data types to a decimal number. When converting from a character string, you can specify a format string to give the format of the character string.

Because these functions return a pointer, they have two forms (see "Parameter Conventions" on page 151). The names are identical, except one is appended with "Buf." In the first form listed, the function returns a pointer to the string. The string is stored in a buffer maintained by DTK. You must copy the string out of this buffer before you call another DTK function, because the next function may use the same buffer.

In the second form listed, appended with "Buf", you pass in a pointer to a buffer you have allocated. The string is put in the buffer. You must make sure that the buffer is large enough to hold the returned string.

Syntax

Converting from Char:

```
ptrstr  dec_val qeCharToDecimal (
    int16     precision,
    int16     scale,
    ptrstr    char_val,
    ptrstr    fmt_string)
```

```
int16  res_code qeCharToDecimalBuf (
    ptrstr   dec_val,
    int16    precision,
    int16    scale,
    ptrstr   char_val,
    ptrstr   fmt_string)
```

Converting from Double:

```
ptrstr  dec_val qeDoubleToDecimal (
    int16    precision,
    int16    scale,
    float64  dbl_val)
```

```
int16  res_code qeDoubleToDecimalBuf (
    ptrstr   dec_val,
    int16    precision,
    int16    scale,
    float64  dbl_val)
```

Converting from Float:

```
ptrstr  dec_val qeFloatToDecimal (
    int16    precision,
    int16    scale,
    float32  flt_val)
```

```
int16  res_code qeFloatToDecimalBuf (
    ptrstr   dec_val,
    int16    precision,
    int16    scale,
    float32  flt_val)
```

Converting from Int:

```
ptrstr  dec_val qeIntToDecimal (
    int16    precision,
    int16    scale,
    int16    int_val)
```

```
int16 res_code qeIntToDecimalBuf (
    ptrstr   dec_val,
    int16    precision,
    int16    scale,
    int16    int_val)
```

Converting from Long:

```
ptrstr  dec_val qeLongToDecimal (
    int16    precision,
    int16    scale,
    int32    long_val)
```

```
int16 res_code qeLongToDecimalBuf (
    ptrstr   dec_val,
    int16    precision,
    int16    scale,
    int32    long_val)
```

Parameters

dec_val is the returned decimal number value.

precision is the number of significant digits in the result.

scale specifies the location of the decimal point in the result.

char_val, *dbl_val*, *flt_val*, *int_val*, and *long_val* are the values to be converted to a decimal number.

fmt_string is the format string (see “Format Strings” on page 59). If no format string is given, DTK assumes that the character string contains a number formatted as GN. If the character string contains a date-time value, *fmt_string* can be used to give its format, and the result will be the Julian value represented by the date-time.

res_code is the result code returned by the function, which returns the same set of result codes as qeErr. See Appendix D, “Result and Error Message Codes,” on page 537 for a list of these result codes.

Example

To convert a character string to a decimal number with 8 digits of precision and 2 digits right of the decimal point:

```
string1 = qeCharToDecimal (8, 2, "1500", "")    ;
```

To convert a character string containing a date-time value to a Julian decimal number with 12 digits of precision and 2 digits right of the decimal point:

```
string1 = qeCharToDecimal (12, 2, "04/07/53", "mm/dd/
yy");
```

Converting to Double-Precision Floating-Point Numbers

These functions convert a value from any of DTK's data types to an 8-byte double-precision floating-point number (type 7).

When converting from a character string, you can specify a format string to give the format of the character string.

Syntax

```
float64  dbl_val qeCharToDouble   (
    ptrstr   char_val,
    ptrstr   fmt_string)

float64  dbl_val qeDateToDouble (ptrstr   date_val)

float64  dbl_val qeDecimalToDouble   (
    ptrstr   dec_val,
    int16    precision,
    int16    scale)

float64  dbl_val qeFloatToDouble (float32   flt_val)

float64  dbl_val qeIntToDouble (int16   int_val)

float64  dbl_val qeLongToDouble (int32   long_val)
```

Parameters

dbl_val is the returned double float value.

char_val, *flt_val*, *long_val*, *dec_val*, *int_val*, and *date_val* are the values to be converted to a double float.

fmt_string is the format string (see“Format Strings” on page 59). If no format string is given, DTK assumes that the character string contains a number formatted as “GN.” If the character string contains a date-time value, *fmt_string* can be used to give its format, and the result will be the Julian value represented by the date-time.

precision is the number of significant digits when converting from a decimal number.

scale specifies the location of the decimal point when converting from a decimal number.

Example

To convert a character string to a double float:

```
dbl_val = qeCharToDouble ("1500", "")    ;
```

To convert a character string containing a date-time value to a Julian double float:

```
dbl_val = qeCharToDouble ("04/07/53", "mm/dd/yy")     ;
```

Converting to Floating-Point Numbers

These functions convert a value from any of DTK’s data types to a 4-byte floating-point number (type 6).

When converting from a character string, you can specify a format string to give the format of the character string.

Syntax

```
float32 flt_val qeCharToFloat (
    ptrstr char_val, ptrstr fmt_string)

float32 flt_val qeDecimalToFloat (
    ptrstr   dec_val,
    int16    precision,
    int16    scale)

float32 flt_val qeDoubleToFloat (float64 dbl_val)

float32 flt_val qeIntToFloat (int16 int_val)

float32 flt_val qeLongToFloat (int32 long_val)
```

Parameters

flt_val is the returned floating-point value.

char_val, *dbl_val*, *long_val*, *dec_val*, and *int_val* are the values to be converted to a floating-point number.

fmt_string is the format string (see "Format Strings" on page 59). If no format string is given, DTK assumes that the character string contains a number formatted as GN. If the character string contains a date-time value, *fmt_string* can be used to give its format, and the result will be the Julian value represented by the date-time.

precision is the number of significant digits when converting from a decimal number.

scale specifies the location of the decimal point when converting from a decimal number.

Example

To convert a character string to a floating-point number:

```
flt_val = qeCharToFloat ("1500", "")    ;
```

To convert a character string containing a date-time value to a Julian floating-point number:

```
flt_val = qeCharToFloat ("04/07/53", "mm/dd/yy")    ;
```

Converting Binary Values to Hexadecimal

qeBinToHex and qeBinToHexBuf convert a binary value into a hexadecimal value.

Syntax

```
ptrstr  hex_value qeBinToHex (ptrstr   bin_value, int16
length)
```

```
int16 res_code qeBinToHexBuf   (
    ptrstr    hex_value,
    ptrstr    bin_value,
    int16     length)
```

Description

qeBinToHex and qeBinToHexBuf convert a binary value into a hexadecimal value and place the result in a buffer. The buffer must be twice the size of the binary value.

Because this function returns a pointer, it has two forms (see "Parameter Conventions" on page 151).

qeBinToHex returns a pointer to the hexadecimal value. This value is stored in a buffer maintained by DTK. You must copy the value out of this buffer before you call another DTK function, because the next function may use the same buffer.

With qeBinToHexBuf, you pass in a pointer to a buffer you have allocated. The hexadecimal value is put in the buffer. You must make sure that the buffer is large enough to hold the returned value.

Parameters

hex_value points to a buffer allocated by the user to accept the converted binary value. It must be at least 2 * *length* bytes long.

bin_value points to a string of *length* bytes of binary data. It is not a null-terminated string.

length is the length of the binary string pointed to by *bin_value*.

res_code is the result code returned by qeBinToHexBuf, which returns the same set of result codes as qeErr. See Appendix D, “Result and Error Message Codes,” on page 537 for a list of these result codes.

Example

```
hex_value = qeBinToHex (bin_value, bin_length)    ;
```

Converting to Integers

These functions convert a value from any of DTK’s data types to a 2-byte integer (type 5). When converting from a character string, you can specify a format string to give the format of the character string.

Syntax

```
int16 int_val qeCharToInt (ptrstr   char_val, ptrstr
fmt_string)

int16 int_val qeDecimalToInt   (
    ptrstr   dec_val,
    int16     precision,
    int16     scale)

int16 int_val qeDoubleToInt (float64   dbl_val)

int16 int_val qeFloatToInt (float32   flt_val)

int16 int_val qeLongToInt (int32   long_val)
```

Parameters

int_val is the returned integer value.

char_val, *dbl_val*, *flt_val*, *dec_val*, and *long_val* are the values to be converted to an integer.

fmt_string is the format string (see Chapter 4, “Retrieving and Converting Data,” on page 37). If no format string is given, DTK assumes that the character string contains a number formatted as GN.

precision is the number of significant digits when converting from a decimal number.

scale specifies the location of the decimal point when converting from a decimal number.

Example

To convert a character string to an integer:

```
int_val = qeCharToInt ("1500", "")   ;
```

Notes

You should not attempt to convert date-time values to integers because the resulting Julian value is too large for a 2-byte integer.

The format of decimal numbers is described in "Format Strings" on page 59.

Converting to Long Integers

These functions convert a value from any of DTK's data types to a 4-byte integer (type 4).

When converting from a character string, you can specify a format string to give the format of the character string.

Syntax

```
int32  long_val qeCharToLong   (

ptrstr  char_val, ptrstr  fmt_string)

int32  long_val qeDecimalToLong   (
    ptrstr    dec_val,
    int16     precision,
    int16     scale)

int32  long_val qeDoubleToLong (float64   dbl_val)

int32  long_val qeFloatToLong (float32   flt_val)

int32  long_val qeIntToLong (int16   int_val)

int32  long_val qeDateToLong (ptrstr   date_val)
```

Parameters

long_val is the returned long integer value.

char_val, *dbl_val*, *flt_val*, *dec_val*, *int_val*, and *date_val* are the values to be converted to a long integer.

fmt_string is the format string (see"Format Strings" on page 59). If no format string is given, DTK assumes that the character string contains a number formatted as GN. If the character string contains a date-time value, *fmt_string* can be used to give its format, and the result will be the Julian value represented by the date-time.

precision is the number of significant digits when converting from a decimal number.

scale specifies the location of the decimal point when converting from a decimal number.

Example

To convert a character string to a long integer:

```
long_val = qeCharToLong ("1500", "")   ;
```

To convert a character string containing a date-time value to a Julian long integer:

```
long_val = qeCharToLong ("04/07/53", "mm/dd/yy")    ;
```

Go To

B For Microsoft Visual Basic Users

This appendix explains how to use DTK with Visual Basic (VB), version 2.0 or higher, to develop VB applications that access data from the databases DTK supports.

Using DTK with Visual Basic

You can call DTK's DLL functions directly from VB. For background information, please read Chapter 22, "Calling Procedures in DLLs," in the Visual Basic *Programmer's Guide*.

DTK comes with sample VB applications that can be used as template for developing other VB applications. By default, these examples are installed in subdirectories of the EXAMPLES directory under your install directory.

Every DTK function that can be called from VB is declared in the code module file, QEDEMO.BAS. You can copy these declarations into the code module of your application so that you do not have to enter them by hand.

This appendix contains the following sections:

- "A VB Example" on page 510 shows the code for a sample VB application that calls DTK functions.
- "DTK Functions for Visual Basic Users" on page 512 introduces the three kinds of functions VB can use.

- "Standard DTK Functions" on page 513 explains how to call the majority of DTK functions.
- "VB-Specific Functions" on page 513 covers the four DTK functions designed for VB users only.
- ""Buf" Functions" on page 524 lists the functions VB users use as alternatives to those functions that return a pointer to a value.
- "Data Types" on page 527 gives the VB equivalents for DTK's data types.

A VB Example

The following sample code shows how to use Visual Basic to connect and disconnect from the dBASE database system, use the VB-specific functions to fetch and update a record, and check for errors. The VB-specific functions pass current record information into arrays. The line-continuation arrow (➤) denotes wrapped lines that must be entered as one line of code in the Code window.

```
'Declare arrays to hold current record, format strings, and errors
Dim RecordArray() As Varian  t
Dim FormatStringsArray() As Strin  g
Dim ErrorsArray() As Intege  r
Dim hdbc As Integer, hstmt As Integer, res_code As Integer

'Call qeConnect to connect to a data source. Check to see if hdbc == 0,
'which indicates that the connection failed.
    hdbc = qeConnect("DRV=QEDBF"  )
    'Error-handling routin  e
    If hdbc = 0 The  n
       MsgBox "qeConnect failed, error = " + Str$(qeErr()    )
       Exit Su b
    End I f

    'Call qeExecSQL to select dept and salary values from Tim Grove's recor     d
    hstmt = qeExecSQL(hdbc, "SELECT dept, salary FROM c:\qelib\emp
    ➤WHERE first_name = 'Timothy' AND last_name = 'Grove'"    )

'Error-handling routin  e
```

```
    If hstmt = 0 The n
       err_qe% = qeErr(  )
       If err_qe% = 4 Then err_db& = qeDBErr(   )
       MsgBox "qeExecSQL failed, error = " + Str$(err_qe%) + "dberr = " +
Str$(err_db& )
       res_code = qeDisconnect(hdbc  )
       Exit Su b
    End I f

    'Get the number of columns in the SQL statemen   t
    NumCols% = qeNumCols(hstmt  )
    ReDim RecordArray(1 To NumCols%  )
    ReDim FormatStringsArray(1 To NumCols%   )
    ReDim ErrorsArray(1 To NumCols%  )
    'Call qeVBFetchNext to retrieve the record indicated by the SQ    L
    'statemen t
    'This function stores the record's values and other info in th    e
    'array s
        res_code = qeVBFetchNext(hstmt, RecordArray(), FormatStringsArray(),
         ➤ErrorsArray() )
'Error-handling routin e
        If res_code <> 0 The  n
            MsgBox "qeVBFetchNext failed, error = " + Str$(res_code    )
        Els e
            'Check whether errors occurred while fetching in a colum    n
            For n = 1 To NumCols  %
                If ErrorsArray(n) <> 0 The   n
                    MsgBox "qeVBFetchNext column " + Str$(n) + " error = " +
                     ➤Str$(ErrorsArray(n) )
                End I f
            Next  n
        End I f

    'Set new dept and salary values for the current recor    d
    If res_code = 0 The n
        RecordArray(1) = "D101  "
        RecordArray(2) = "$42000  "
        'Call qeVBPutRecord to put the new values into the current recor    d
        res_code = qeVBPutRecord(hstmt, RecordArray(), FormatStringsArray(),
         ➤ErrorsArray() )
        If res_code <> 0 The  n
            MsgBox "qeVBPutRecord failed, error = " + Str$(res_code    )
        Els e
            For n = 1 To NumCols  %
                If ErrorsArray(n) <> 0 The   n
```

```
                MsgBox "qeVBPutRecord column " + Str$(n) + " error = " +
                 ➤Str$(ErrorsArray(n) )
            End I f
        Next  n
    End I f
'Call qeRecUpdate to update the current record in the databas    e
res_code = qeRecUpdate(hstmt  )
    If res_code <> 0 The  n
        MsgBox "qeRecUpdate failed, error = " + Str$(res_code    )
    End I f
'Call qeEndSQL to end the SQL statemen  t
res_code = qeEndSQL(hstmt  )
If res_code <> 0 The  n
    MsgBox "qeEndSQL failed, error = " + Str$(res_code    )
End I f
'Call qeDisconnect to disconnect from a data source   .
res_code = qeDisconnect(hdbc  )
If res_code <> 0 The  n
    MsgBox "qeDisconnect failed, error = " + Str$(res_code    )
End I f
```

DTK Functions for Visual Basic Users

To Visual Basic users, DTK has three kinds of functions:

- Standard DTK functions. Most of DTK's functions are standard functions that can be called in VB just as they are in other development environments.
- VB-specific functions. For VB users, DTK 2.*x* provides four functions to simplify and speed up fetching and putting records: qeVBFetchNext, qeVBFetchPrev, qeVBFetchRandom, and qeVBPutRecord.
- "Buf" functions. Because VB does not allow functions to return pointers to values, DTK provides a set of alternate, "Buf" functions that fill memory buffers that you must allocate.

The following sections tell you more about these functions.

Standard DTK Functions

The vast majority of DTK functions work the same in VB as they do in other development environments. For example, you can insert, update, or delete records in VB just as you do from Microsoft C++, by issuing a SQL Insert, Update, or Delete statement using qeExecSQL, or by calling qeRecNew, qeRecUpdate, or qeRecDelete. The previous chapters of this manual explain how to call the standard DTK functions.

VB users cannot use the qeBindCol function, nor any of those functions that return a pointer to a value, such as qeValChar. DTK provides the “Buf” functions as alternatives.

VB-Specific Functions

Only VB users can call the four functions described in this section, qeVBFetchNext, qeVBFetchPrev, qeVBFetchRandom, and qeVBPutRecord. These functions provide a much faster and easier way to retrieve database records than the qeValBuf functions. The VB-specific functions treat records as arrays of Variants, which are easier to manipulate than the data returned from qeValMultiCharBuf.

qeVBFetchNext

qeVBFetchNext is used in VB applications to retrieve the next record from the database.

Syntax

```
int16 res_code qeVBFetchNext (int16 hstmt, variant RecordArray(), string FormatStringsArray(), int16 ErrorsArray())
```

Description

qeVBFetchNext retrieves the next record from the database and passes record values into the record array. If this is the first call to qeVBFetchNext following qeExecSQL, this function retrieves the first record. The retrieved record becomes the current record.

qeVBFetchNext passes the current record's values into the record array as Variants. If a column's data type is numeric or date-time, the corresponding element in the format string array can be set to a format string to format the data. If an error occurs while fetching a particular column, the corresponding element in the error array is set to the error returned.

The arrays passed to the function must be declared to contain at least as many field values as there are in the current record, or in other words, the number of columns present in the SQL Select statement.

Parameters

hstmt is the handle to the statement returned by qeExecSQL or qeSQLPrepare.

RecordArray() is a handle to a dynamic array of type Variant. Each element in the array corresponds to a column in the Select statement and therefore a value in the current record. If a column is of type numeric or date-time and a format string is specified for the column, the column value is formatted and converted into a character string.

FormatStringsArray() is a handle to a dynamic array of format strings, one for each column returned. Each array element can be either a format string, which formats date or number data, or a null or empty string value, in which case the corresponding data is returned with no formatting. Numeric and date-time columns with format strings are formatted and converted into character strings. Format strings are ignored for columns of other data types.

ErrorsArray() is a handle to a dynamic array of errors that occur as the function retrieves the current record. If an error occurs as the function fetches the value of a column, the corresponding element contains error values such as those returned by the qeErr function.

res_code is the result code returned by qeVBFetchNext, which returns the same set of result codes as qeErr. See Appendix D, "Result and Error Message Codes," on page 537 for a list of these result codes.

Example

To fetch all records from the employee database file:

```
Dim RecordArray() As Variant
Dim FormatStringsArray() As String
Dim ErrorsArray() As Integer
...
hstmt = qeExecSQL (hdbc, "SELECT * FROM emp");

ReDim RecordArray(1 To NumCols%)
ReDim FormatStringsArray(1 To NumCols%)
ReDim ErrorsArray(1 To NumCols%)

while (qeVBFetchNext(hstmt, RecordArray(),
➤FormatStringsArray(), ErrorsArray() = 0)
    ...
Wend
```

Notes

Whenever you acquire a new *hstmt*, you must call qeVBFetchNext to move the cursor to the first record before you can perform any other operations on the data.

If qeSetAutoUpdate has been called to enable auto-updating, and changes have been made to the current record via calls to qeVBPutRecord, a call to qeVBFetchNext updates the current record.

When qeVBFetchNext reaches the last record returned by the Select statement, it returns a result of qeEOF (-5).

Sometimes, calling the appropriate qeValBuf function is a more efficient way to get the values in the current buffer. For example, when you want to retrieve one value of a character field in the Select statement, call qeFetchNext and qeValCharBuf instead of qeVBFetchNext.

qeVBFetchNext returns values of DTK data type 2 (variable length character string) as strings. Note that Visual Basic strings may have null characters in them.

qeVBFetchNext trims any trailing blanks on data returned as strings, unless the data is of DTK type 2.

Null database values are returned as an empty string for string and date types, or as 0 for numeric types. When qeVBFetchNext returns a null value, the corresponding entry in the error array will be set to qeNULL_DATA (-2).

See Also

qeFetchNext, qeSetSelectOptions, qeVal functions, qeVBFetchPrev, qeVBFetchRandom.

qeVBFetchPrev

qeVBFetchPrev retrieves the previous record from the database in VB applications.

Syntax

```
int16 res_code qeVBFetchPrev (int16  hstmt, varian t
RecordArray(), string FormatStringsArray(), int16
ErrorsArray())
```

Description

qeVBFetchPrev retrieves the previous record from the database and passes record values into the record array. qeVBFetchPrev cannot be called unless qeSetSelectOptions has been called to enable backwards scrolling.

qeVBFetchPrev passes the current record's values into the record array as Variants. If a column's data type is numeric or date-time, the corresponding element in the format array can be set to a format string that formats the data. If an error occurs while fetching a particular column, the corresponding element in the error array is set to the error returned.

The arrays passed to the function must be declared to contain at least as many field values as there are in the current record, or in other words, the number of columns present in the SQL Select statement.

Parameters

hstmt is the handle to the statement returned by qeExecSQL or qeSQLPrepare.

RecordArray() is a handle to a dynamic array of type Variant. Each element in the array corresponds to a column in the Select statement and therefore a value in the current record. If a column is of type numeric or date-time and a format string is specified for the column, the column value is formatted and converted into a character string.

FormatStringsArray() is a handle to a dynamic array of format strings, one for each column returned. Each array element can be either a format string, which formats date or number data, or a null or empty string value, in which case the corresponding data is returned with no formatting. Numeric and date-time columns with format strings are formatted and converted into character strings. Format strings are ignored for columns of other data types.

ErrorsArray() is a handle to a dynamic array of errors that occur as the function retrieves the current record. If an error occurs as the function fetches the value of a column, the corresponding element contains error values such as those returned by the qeErr function.

res_code is the result code returned by qeVBFetchPrev, which returns the same set of result codes as qeErr. See Appendix D, "Result and Error Message Codes," on page 537 for a list of these result codes.

Example

To fetch the previous record from the employee database file:

```
Dim RecordArray() As Variant
Dim FormatStringsArray() As String
Dim ErrorsArray() As Integer
...
hstmt = qeExecSQL (hdbc, "SELECT * FROM emp");
res_code = qeSetSelectOptions (hstmt, 1)

ReDim RecordArray(1 To NumCols%)
ReDim FormatStringsArray(1 To NumCols%)
ReDim ErrorsArray(1 To NumCols%)

qeVBFetchPrev(hstmt, RecordArray(),
FormatStringsArray(),
➤ErrorsArray( )
```

Notes

If qeSetAutoUpdate has been called to enable auto-updating, and changes have been made to the current record via calls to qeVBPutRecord, a call to qeVBFetchPrev updates the current record.

Sometimes, calling the appropriate qeValBuf function is a more efficient way to get the values in the current buffer. For example, when you want to retrieve one value of a character field in the Select statement, call qeFetchPrev and qeValCharBuf instead of qeVBFetchPrev.

qeVBFetchPrev returns values of DTK data type 2 (variable length character string) as strings. Note that Visual Basic strings may have null characters in them.

qeVBFetchPrev trims any trailing blanks on data returned as strings, unless the data is of DTK type 2.

Null database values are returned as an empty string for string and date types, or as 0 for numeric types. When qeVBFetchPrev returns a null value, the corresponding entry in the error array will be set to qeNULL_DATA (-2).

See Also

qeFetchPrev, qeVal functions, qeSetQueryTimeout, qeVBFetchNext, qeVBFetchRandom.

qeVBFetchRandom

qeVBFetchRandom retrieves a specific record from the database in VB applications.

Syntax

```
int16 res_code qeVBFetchRandom (int16  hstmt, int32
rec_num, varian t RecordArray(), string
FormatStringsArray(), int16 ErrorsArray())
```

Description

qeVBFetchRandom retrieves a specified record from the database and passes record values into the record array. qeVBFetchRandom cannot be called unless qeSetSelectOptions has been called to enable backwards scrolling.

qeVBFetchRandom passes the current record's values into the record array as Variants. If a column's data type is numeric or date-time, the corresponding element in the format array can be set to a format string that formats the data. If an error occurs while fetching a particular column, the corresponding element in the error array is set to the error returned.

The arrays passed to the function must be declared to contain at least as many field values as there are in the current record, or in other words, the number of columns present in the SQL Select statement.

Parameters

hstmt is the handle to the statement returned by qeExecSQL or qeSQLPrepare.

rec_num is the record number to be read. The first record is 1.

RecordArray() is a handle to a dynamic array of type Variant. Each element in the array corresponds to a column in the Select statement and therefore a value in the current record. If a column is of type numeric or date-time and a format string is specified for the column, the column value is formatted and converted into a character string.

FormatStringsArray() is a handle to a dynamic array of format strings, one for each column returned. Each array element can be either a format string, which formats date or number data, or a null or empty string value, in which case the corresponding data is returned with no formatting. Numeric and date-time columns with format strings are formatted and converted into character strings. Format strings are ignored for columns of other data types.

ErrorsArray() is a handle to a dynamic array of errors that occur as the function retrieves the current record. If an error occurs as the function fetches the value of a column, the corresponding element contains error values such as those returned by the qeErr function.

res_code is the result code returned by qeVBFetchRandom, which returns the same set of result codes as qeErr. See Appendix D, “Result and Error Message Codes,” on page 537 for a list of these result codes.

Example

To fetch the last record from the employee database file:

```
Dim RecordArray() As Varian   t
Dim FormatStringsArray() As Strin   g
Dim ErrorsArray() As Intege   r
...
hstmt = qeExecSQL (hdbc, "SELECT * FROM emp"     )
res_code = qeSetSelectOptions (hstmt, 1    )

num_recs% = qeFetchNumRecs (hstmt    )

ReDim RecordArray(1 To NumCols%    )
ReDim FormatStringsArray(1 To NumCols%     )
ReDim ErrorsArray(1 To NumCols%    )

qeVBFetchRandom (hstmt, RecordArray(),
➤FormatStringsArray(), ErrorsArray()    )
```

Notes

If qeSetAutoUpdate has been called to enable auto-updating, and changes have been made to the current record via calls to qeVBPutRecord functions, a call to qeVBFetchRandom updates the current record.

Sometimes, calling the appropriate qeValBuf function is a more efficient way to set the values in the current buffer. For example, when you want to retrieve one value of a character field in the Select statement, call qeFetchRandom and qeValCharBuf instead of qeVBFetchRandom.

qeVBFetchRandom returns values of DTK data type 2 (variable length character string) as strings. Note that Visual Basic strings may have null characters in them.

qeVBFetchRandom trims any trailing blanks on data returned as strings, unless the data is of DTK type 2.

Null database values are returned as an empty string for string and date types, or as 0 for numeric types. When qeVBFetchRandom returns a null value, the corresponding entry in the error array is set to qeNULL_DATA (-2).

See Also

qeVal functions, qeVBFetchPrev, qeVBFetchRandom.

qeVBPutRecord

qeVBPutRecord is used to set values in the current record buffer.

Syntax

`int16` ***`res_code`*** `qeVBPutRecord (int16` ***`hstmt,`*** `variant` ***`RecordArray()`***`, string` ***`FormatStringsArray()`***`, int16` ***`ErrorsArray()`***`)`

Description

qeVBPutRecord updates the current record buffer with new values passed in from the record array. Note that qeVBPutRecord does not change the values in the database. To actually modify the database, you must call qeRecUpdate.

If you are putting a value into a date-time column, you must also pass a format string in the corresponding element in the format string array. If you do not know the value's format, you can use the Format$ function to convert the date to a known format. Then you can set the corresponding element in the format string array to match this format, for example,

```
RecordArray(2) = Format$(Text2.text, "m/d/yy"     )
FormatStringsArray(2) = "m/d/yy   "
```

Similarly, if you are putting a string value into a numeric column, you can use VB's conversion functions to convert it to a numeric value. For example, the following code converts the value in the Text1 control to a Variant that is internally represented as an Integer.

```
RecordArray(3) = CVar (CInt (Text1.text)    )
```

Refer to Visual Basic's *Language Reference* for more information on these VB functions.

If an error occurs while putting a value into a particular column, the corresponding element in the error array is set to the error returned.

The arrays passed to the function must be declared to contain at least as many elements as there are columns in the SQL Select statement.

If any element of the record array contains a null value or an empty string, qeVBPutRecord puts a null value into the current record buffer. To avoid overwriting values in the database, be sure to pass values in all elements of the record array, even if they do not differ from the original database values.

Parameters

hstmt is the handle to the statement returned by qeExecSQL or qeSQLPrepare.

RecordArray() is a handle to a dynamic array of type Variant. Each element in the array corresponds to a column in the Select statement and therefore a value in the current record. If a column is of type numeric or date-time and a format string is specified for the column, the column value is formatted and converted into a character string.

FormatStringsArray() is a handle to a dynamic array of format strings, one for each column returned. Each array element can be either a format string, which formats date or number data, or a null or empty string value, in which case the corresponding data is returned with no formatting. Numeric and date-time columns with format strings are formatted and converted into character strings. Format strings are ignored for columns of other data types.

ErrorsArray() is a handle to a dynamic array of errors that occur as the function retrieves the current record. If an error occurs as the function fetches the value of a column, the corresponding element contains error values such as those returned by the qeErr function.

res_code is the result code returned by qeVBPutRecord, which returns the same set of result codes as qeErr. See Appendix D, "Result and Error Message Codes," on page 537 for a list of these result codes.

Example

To put a record into the current record buffer and update the employee database file:

```
Dim RecordArray() As Variant
Dim FormatStringsArray() As String
Dim ErrorsArray() As Integer
...
hstmt = qeExecSQL (hdbc, "SELECT dept, salary FROM
➤d:\qelib\emp WHERE first_name = 'Timothy' AND last_name
➤= 'Grove'") ;

ReDim RecordArray(1 To NumCols%)
ReDim FormatStringsArray(1 To NumCols%)
ReDim ErrorsArray(1 To NumCols%)

res_code = qeVBFetchNext(hstmt, RecordArray(),
FormatStringsArray(), ErrorsArray()
...

If res_code = 0 Then
    RecordArray(1) = "D101"
    RecordArray(2) = "$42,000.00"
    FormatStringsArray(2) = "$###,###.00"
    res_code = qeVBPutRecord(hstmt, RecordArray(),
    ➤FormatStringsArray(), ErrorsArray()
```

Notes

Sometimes it is more efficient to set values in the current buffer by calling the appropriate qePut functions. For example, when updating only one column in each record when several columns have been selected, calling a qePut function is faster than calling qeVBPutRecord.

qeVBPutRecord returns values in DTK data type 2 (variable length character string) columns as strings. Note that Visual Basic strings may have null characters in them. To put a value into a of DTK type 2 column, set the Variant to a string containing the value you wish to put.

Null database values are returned as an empty string for string and date types, or as 0 for numeric types. When qeVBFetchNext returns a null value, the corresponding entry in the error array is set to qeNULL_DATA (-2).

See Also

qePut functions, qeRecUpdate, qeVal functions, qeVBFetchNext, qeVBFetchPrev, qeVBFetchRandom.

"Buf" Functions

Visual Basic does not support DLL functions that return a pointer to a value. Because of this limitation, DTK provides alternative forms of these functions. These forms have the same name as the standard forms and end with "Buf," for example, qeErrMsgBuf.

When using the Buf functions, your VB program must allocate a buffer to hold the value returned by the function. Also, you must pass the pointer to the buffer as an additional parameter to the Buf functions. Make sure that the size of the buffer you allocate is large enough to hold the returned value.

The following table lists the DTK functions not supported and the alternative function that you must use.

Don't Use	Use Instead
qeBindCol functions	the qeVBFetch functions or the qeValBuf functions
qeClauseGet	qeClauseGetBuf
qeColAlias	qeColAliasBuf
qeColDBTypeName	qeColDBTypeNameBuf
qeColExpr	qeColExprBuf
qeColName	qeColNameBuf
qeErrMsg	qeErrMsgBuf

Don't Use	Use Instead
qeGetODBCInfoChar	qeGetODBCInfoCharBuf
qeNativeSQL	qeNativeSQLBuf
qeQryGetFileName	qeQryGetFileNameBuf
qeQryGetParamDefault	qeQryGetParamDefaultBuf
qeQryGetParamFormat	qeQryGetParamFormatBuf
qeQryGetParamName	qeQryGetParamNameBuf
qeQryGetParamPrompt	qeQryGetParamPromptBuf
qeQryGetSource	qeQryGetSourceBuf
qeQryGetStmt	qeQryGetStmtBuf
qeSetupInfo	qeSetupInfoBuf
qeUniqueWhereClause	qeUniqueWhereClauseBuf
qeValChar	the qeVBFetch functions or qeValCharBuf
qeValDecimal	the qeVBFetch functions or qeValDecimalBuf
qeValMultiChar	the qeVBFetch functions or qeValMultiCharBuf
qeVerNum	qeVerNumBuf
qeBinToHex	qeBinToHexBuf
qeHexToBin	qeHexToBinBuf
qeDateToChar	qeDateToCharBuf
qeDecimalToChar	qeDecimalToCharBuf
qeDoubleToChar	qeDoubleToCharBuf
qeFloatToChar	qeFloatToCharBuf

Don't Use	Use Instead
qeIntToChar	qeIntToCharBuf
qeLongToChar	qeLongToCharBuf
qeCharToDate	qeCharToDateBuf
qeCharToDecimal	qeCharToDecimalBuf
qeDoubleToDecimal	qeDoubleToDecimalBuf
qeFloatToDecimal	qeFloatToDecimalBuf
qeIntToDecimal	qeIntToDecimalBuf
qeLongToDecimal	qeLongToDecimalBuf

Allocating Buffers

When you call qeValCharBuf, or any other Buf function, it is critical that you allocate a buffer to hold the value returned by the function. In other words, the string variable you pass as the second parameter must be long enough to hold the column value returned from the database.

There are two ways you can set the size of a string variable. You can declare the variable as a *fixed-length string* using the Dim statement:

```
Dim value As String * 255
```

Or you can use *variable-length strings* by either not declaring them at all—just assign to value$—or declaring them as:

```
Dim value As String
```

If you use variable-length strings, you must immediately precede each call to qeValCharBuf with:

```
value = String$(255,0)
```

Both techniques allocate a string variable of the specified size, 255 in these examples. The required length of your string variables depends on the size of column values that you are retrieving. You can call qeColWidth to get the maximum size of a column.

If your application has a General Protection Fault on a call to a qeVal function, chances are the string variable you sent was not large enough to hold the column value.

Data Types

Each column in a table has a *data type*. The data type determines the type of information that can be stored in the column. See “Data Types in DTK” on page 53 for more information.

With the exception of Decimal numbers, DTK’s data types can be mapped directly to VB data types. The following table shows the DTK data types and the corresponding VB data types.

Identifier	DTK Data Type	VB Data Type
1	Fixed length character string	String
2	Variable length character string	String
3	Decimal number (BCD)	N/A
4	Long integer (4-byte)	Long
5	Integer (2-byte)	Integer
6	Floating-point numbers (4-byte)	Single
7	Double-precision floating-point numbers (8-byte)	Double
8	Date-Time (26-byte character string)	String

DTK automatically converts data when you use the qeVal functions. For example, you can use the qeValCharBuf function to retrieve any database column value, and DTK will convert all values to character strings.

For your convenience, DTK also provides data type conversion functions, listed in Appendix A, “Data Conversion Functions,” on page 493.

If your database contains Decimal numbers, you should use either qeValCharBuf or qeValDouble to convert the numbers to double-precision floating-point numbers.

Go To

C Coding for Single Statement Database Systems

Most database systems allow more than one active statement on a connection. For example, you could read records from Oracle with one statement and update records using a second statement, with both statements sharing a single connection to the database.

Some database systems, however, have the limitation that for each database connection, you can execute only one SQL statement at a time. For convenience, such database systems are called *single-statement* systems in this book. Refer to the *DataDirect ODBC Drivers Reference* for information on whether specific database systems support single or multiple active statements on a connection.

This appendix describes the issues that create special considerations for DTK applications that support single-statement systems, as well as techniques for achieving the best performance possible when writing such applications.

Why Is This an Issue?

The single-statement limitation greatly affects the way that DTK handles the execution of SQL statements, because when your program connects to a single-statement database and issues a Select statement, you cannot issue any other Select, Insert, Update, or Delete statements until you terminate the first Select statement. So if you issue the first Select statement, read a few records, and then want to update a record, you have a problem—you can't update a record until you terminate the Select statement, but if you terminate the Select statement to do the update, then you won't be able to read the rest of the records returned by the Select statement (because you terminated it).

Fortunately, DTK avoids this problem by automatically creating a second connection to the database system whenever it's necessary to execute a SQL statement and the original connection is "busy." This technique is called *cloning connections*.

Suppose your application issues a Select statement, then reads a few records, and then issues an Update (or any other SQL) statement. In this case, DTK detects that your program is trying to issue a second SQL statement while the first one is still active, so it clones the first connection to get a second one, and then executes the Update statement on that second connection.

Note: Many DTK functions create SQL statements as part of their execution—and therefore cause DTK to clone connections to single-statement database systems—so this behavior is not limited to the SQL execution functions. Any DTK function that changes the database can issue SQL statements.

When a statement terminates, the corresponding connection is no longer busy so DTK retains this connection to ensure that it will have one ready when another SQL statement is issued. DTK never retains more than one inactive connection; if an application terminates two statements, it closes one connection and retains the other one for future use.

DTK keeps the number of open connections to a minimum by automatically terminating the Select statement when your application reads the last record. From the viewpoint of your application, its "statement" has not been terminated, because it can still scroll through the records (when random fetching is enabled). But from the viewpoint of the database system, the statement has been closed, so the associated connection can be used again.

DTK is able to continue scrolling through the records after the statement has been terminated because it saves all records in a log file as it reads them. These log files are stored in temporary files on the user's computer.

DTK's use of cloned connections for single statement database systems creates special considerations relative to locking behavior and performance. The following sections describe these considerations, as well as the options that DTK provides for handling them.

Locking Considerations

Database systems generally allow you to have one transaction per connection, and do not let you have one transaction that spans more than one connection. For single-statement systems, two SQL statements running at the same time in separate connections are in separate transactions. So for these database systems, connections and transactions are equivalent.

When a statement causes the database system to lock a record, the lock is acquired in the context of the connection's transaction. If a second statement executing in a second connection attempts to lock the same record, the database system will not let the second lock succeed, because it treats the two transactions as if they were two different users—even though the two transactions were started by the same application. Two transactions cannot lock the same record at the same time, even if the transactions were started by the same application.

Some single-statement databases, when you issue a Select statement, will acquire locks on the pages as they read records from the page. As soon as a such a database reads a record from a different page, it removes the shared lock it had on the first page and acquires a shared lock on the new page. Thus, as an application is reading records from a Select statement, the database system is acquiring and releasing shared locks on the pages as the records are being read. As a result, DTK doesn't know which record's page is currently locked when an application is reading records.

If you issue an Update, Delete, or Insert statement in a single-statement database, it will acquire an exclusive lock on the page containing the changed record. These locks are held until the current transaction ends.

Here is an example of the type of locking problems that may occur. An application issues a Select statement and reads the first record, then attempts to update that record. The Select statement is being executed on one connection, and the Update statement is being executed on a second connection. The first connection has a lock on the page containing the first record, and the Update statement attempts to acquire an exclusive lock on the same page. If these happen to be the same page, the Update statement's attempt to acquire a lock will fail.

DTK avoids this potential problem by reading all of the records from all active Select statements into the log file. This activity is called *read-ahead* in DTK. By default, this read-ahead activity occurs whenever an application attempts to update, delete, or insert a record. When the application attempts to update the first record, DTK will first read all of the records from the Select statement (putting them into the log file), terminate the Select statement, and then execute the Update statement. By doing the read-ahead, DTK guarantees that the Select statement is no longer holding any shared locks, so the Update statement's attempt to get an exclusive lock will not fail because of a conflict with a Select statement in the same application. However, read-ahead cannot prevent a lock conflict between one user's exclusive lock and some other user on a different computer who has a shared or exclusive lock on the same page.

The following sections contain more information on read-ahead behavior and the DTK options you can use to control it.

Performance Considerations

Depending on your network and your server computer, you may notice some delay when connecting to a database; so every time your application issues an SQL statement that causes the database driver to clone a connection, there may be a noticeable delay.

Depending on the installation of your database system, your server may have a limit on the total number of active connections. (For example, you could have only 5 or 10 connections that are shared by all users.) In such a case, you want DTK to use as few connections as possible, and to close them as soon as possible.

One way that you can affect how soon DTK can reuse a connection is to have DTK read all of the records from a Select statement as soon as possible. (This is the same read-ahead activity described in the previous section.) After reading the statement's entire result set into the log file, DTK no longer needs to keep the statement open for you to continue reading, inserting, updating, and deleting records. Once the entire result set is read into the log file, DTK closes the statement and frees the connection it used. So the sooner DTK reads the entire result set returned by the Select statement, the sooner the connection is freed.

Certain events cause DTK to automatically read the entire result set. To optimize the performance and effect of this read-ahead activity, DTK allows you to specify which events trigger it. The following section describes how.

Controlling Read-ahead Activity

The qeSetOneHstmtPerHdbcOptions function lets you specify when DTK will perform read-ahead activity. In addition to other behavior that this function controls, it provides the following read-ahead options:

Constant	Value	Description
qeREADAHEAD_AT_EXEC	0x0001	DTK reads the statement's entire result set into the log file when the statement executes. Reading result sets at this time will often free handles for users of databases who have licenses restricting open handles.
qeREADAHEAD_AT_UPDATE	0x0002	DTK reads the remainder of the result set into the log file whenever a record is locked, updated, or deleted. This is the default read-ahead option.
qeREADAHEAD_COMMIT_UPDATES	0x0003	DTK avoids all read-ahead activity by requiring you to commit all updates before fetching any more records.

The qeREADAHEAD_AT_EXEC option uses the fewest database system resources because it frees the Select statement's connection earliest. It also controls when the read-ahead occurs—any performance lags that the read-ahead may cause occur when the application starts, and not while users are trying to work with the data.

If you know that your users will rarely be updating the database, choose the qeREADAHEAD_AT_UPDATE option to prevent unnecessary read-ahead activity. If your users *will* be updating the database, choosing this option makes any performance penalty caused by read-ahead concurrent with the events that make it necessary—the first time you lock, update, or delete a record. DTK will free the Select statement's connection at that time.

The qeREADAHEAD_COMMIT_UPDATES option allows you to avoid read-ahead activity entirely when using transactions by agreeing to commit all database changes before fetching additional records from the result set. When using this option, your application must not fetch any records from this result set from the time you first update a record until you commit all updates. Note that when you choose this option, DTK uses a different locking protocol to eliminate the need for read-ahead activity—fetching at the wrong time can cause you to hang the database as well as your application. Because it does not read-ahead, DTK cannot free the Select statement's connection until qeEndSQL is called.

Another way to avoid unnecessary read-ahead activity is to tell DTK when the statement you are issuing will not affect other active statements. For example, suppose your application has an active Select statement like

```
SELECT * FROM em p
```

and you want to issue another statement like

```
SELECT * FROM dep t
```

Because these two statements read data from separate tables, updates to the result set from one of these statements cannot affect records in the result set of the other. This means that when you update a record returned by one statement, DTK does not need to read-ahead to free shared locks on both Select statements—only the one that is getting updated. If the two statements read data from the same table, DTK would have to read-ahead on both statements in order to free all shared locks and perform the update.

By default, DTK will always assume that a statement can affect other active statements, and will read-ahead on all active statements when performing an update. However, by calling the qeSetOneHstmtPerHdbcOptions function and setting the qeHSTMT_LOCAL flag (0x0020), you can inform DTK whenever the next statement issued will not affect other active statements, and thereby prevent unnecessary read-ahead activity on those statements.

Preventing Statement Conflicts

When sending multiple statements through cloned connections, it is important to send all statements that modify the database through the same connection. Doing so prevents the locking conflicts that can otherwise occur.

DTK can usually determine whether a statement will modify the database. To do so, it examines the first word of the statement. If that word is any other than “Select,” DTK sends the statement through the connection used for statements that modify the database. However, some statements, such as those that invoke stored procedures, may cause DTK to guess incorrectly. If your application uses such statements, then before issuing them you should call the qeSetOneHstmtPerHdbcOptions function and set one of the following flags:

Constant	Value	Description
qeROUTING_READ	0x0008	DTK will route this statement through a connection used for read-only statements.
qeROUTING_UPDATE	0x0010	DTK will route this statement through a connection used for statements that modify the database.
qeROUTING_DEFAULT	0x0018	This option allows DTK to decide which connection to send the statement to. This is the default routing option.

Go To

D Result and Error Message Codes

This appendix lists the result and error codes returned by qeErr and the error messages returned by qeErrMsg and qeErrMsgBuf.

Result Codes

The following table lists the result codes returned by qeErr and other functions that return result codes.

Constant	Value	Description
qeLOCK_NO_REC	-6	A lock was attempted, but either no record was selected by the primary key, the record has been deleted by another user, or another user has changed the value of a key field.
qeEOF	-5	EOF. Returned by qeFetchNext, qeFetchPrev, or qeFetchRandom when there is no record to return.
qeUSER_CANCELED	-4	User canceled out of the logon dialog box.
qeOUT_OF_MEMORY	-3	Windows or OS/2 is out of memory. This is usually fatal.

Constant	Value	Description
qeSUCCESS	0	Success.
qeSUCCESS_WITH_INFO	1	Success with information (warning).
qeNO_DATA_WITH_INFO	2	EOF with additional information (usually ESC during a fetch).
qeDBSYS_ERROR	4	Database system error. Call qeDBErr to retrieve the database system's error number.
qeLIBSYS_ERROR	5	Returned when the system cannot locate the DTK Dynamic Link Library.

Error Codes and Messages

The following error codes are returned by qeErr or any other function that returns a result code. The corresponding messages can be retrieved with a call to qeErrMsg or qeErrMsgBuf.

Code #	Error Message Text
1100	Error on menu operation. Resources may be getting low.
1500	Not enough memory for data transfer--message truncated.
1501	Cannot create file '*file_name*'.
1502	Cannot delete file: '*file_name*'.
1503	Not enough memory for this command.
1504	Cannot set current working directory to '*dir_name*'.

Code #	Error Message Text
1506	Insufficient disk space.
1507	Invalid file handle.
1508	Access to file denied '*file_name*'.
1509	File not found '*file_name*'.
1510	Path not found '*path_name*'.
1511	You must run SHARE when locking is enabled or you must set Locking=NONE in your ODBC.INI file.
1512	Whole or part of the region has already been locked.
1513	Unable to unlock record.
1514	Lock failed! SHARE buffers have been exceeded.
1515	Unable to load help file.
1516	Not a DOS disk.
1517	Invalid Parameter.
1518	File read locks not supported
1519	Not owner of resource access has been denied
1520	File currently exist.
1521	File dead lock has been detected.
1522	No file lock resource exist.
1523	Unable to load DLL *'dll_name'* because of *'reason'*.
1524	File name is too long: '*file_name*'.
2100	You can only logon to this database once.
2105	Unable to load dynamic link library: '*file_name*'
2106	Connection string must contain a DSN=<driver_name>: '*incorrect_string*'
2108	Transaction processing is not supported for this database driver
2700	Token too big: '*token_name*'

Code #	Error Message Text
2701	Number too large: '*number_string*'
2702	Number contains an invalid character: '*invalid_char*'
2703	Unmatched quote character: '*character*'
2704	Error parsing connect string at offset '*offset*'.
2705	Error parsing '*string*' at offset '*offset*'.
2706	Attribute '*attribute*' specified more than once.
2707	Attribute specified twice using keywords '*keyword1*' and '*keyword2*'.
2708	Invalid hexadecimal character found during conversion
2709	Quicksort stack overflow.
2710	Too many sort keys.
2711	Invalid license file: '*file_name*'
2712	The Beta period for this product has expired. Please contact INTERSOLV to obtain a production version of this product
2713	The evaluation period for this product has expired. Please contact INTERSOLV to obtain a production version of this driver.
2714	The Beta period for this product will expire in less than 15 days. Please contact INTERSOLV to obtain a production version of this product.
2715	The evaluation period for this product will expire in less than 15 days. Please contact INTERSOLV to obtain a production version of this driver.
2716	Cannot handle strings larger than 65500 bytes
2717	Initialization file is not open.
2718	This is a not-for-resale version of a INTERSOLV product. You can order INTERSOLV products by calling 800-547-4000.
2719	Could not create trace window.
2720	Error parsing first line of query file: '*file_name*'.
2721	Could not get needed access to '*problem*'.

Code #	Error Message Text
2722	Can not increase internal array size past 16000.
2723	You are now using the INTERSOLV ODBC Drivers from the Database Library product. These drivers may only be used for developing and testing applications. They may not be distributed for commercial use.
2724	To use or distribute this ODBC-enabled application with drivers from INTERSOLV, you must purchase the appropriate driver distribution license. Please contact INTERSOLV at 800-547-4000 for more information and assistance.
2725	The license file, *'file_name'*, does not authorize you to use this ODBC driver. Please contact INTERSOLV at 800-547-4000 to purchase a license.
2726	The license has expired. Please call INTERSOLV to obtain a production version of this product.
2727	The license will expire in less than 15 days. Please call INTERSOLV to obtain a production version of this product
2728	USA and Canada: 800-547-4000 Asia Pacific: 301-838-5241 United Kingdom: +44 1727 812812 Australia: +61 (3) 9816 9977 France: +33 (1) 49.03.09.99 Germany: +49 (89)962 71-152 Other countries: +44 1727 812812.
3501	Your format mask is too long, the limit is 79 characters
3502	The E format character must be followed by a sign character; + or -. For Example, '0.00E+00'.
3503	The E format character must be followed by one or more digits to display the exponent. For Example, '0.00E+00'.
3504	The quoted string in your format mask is missing the second quotation mark.
3505	The scale command must be followed by '*' for multiply or by '/' for divide. For Example, '[S*1000]'.
3506	The scale operator must be followed by a number that is a power of ten; 10, 100, 1000, etc. For Example, '[S*1000]'.

Code #	Error Message Text
3507	A command in your format mask is missing the ']' end command character. For Example, '[S*1000]'.
3508	Partial values cannot be formatted or converted
3509	You attempted to format or convert a date value that is not a valid date.
3510	Overflow resulted when converting a value to single-precision floating-point.
3511	Overflow resulted when converting a value to short integer.
3512	Overflow resulted when converting a value to decimal format
3513	The value being converted has an exponent that is too large
3514	Overflow resulted when converting a value to long integer.
3515	The date contains an invalid year.
3516	The date contains an invalid month.
3517	The date contains an invalid day.
3518	The date contains an invalid hour.
3519	The date contains an invalid minute.
3520	The date contains an invalid second.
3521	The date contains invalid fractional seconds.
3522	This character cannot appear in a date format string:'*character*'
3523	This character cannot appear in a number format string:'*character*'
3524	This character cannot appear in a general format string:'*character*'
3525	This string cannot be converted to a number:'*character*'
3526	Could not convert to a date value:'*unconverted_value*'
3527	Overflow resulted when converting a value to double-precision floating-point.
3528	Invalid decimal (BCD) digit in nibble'*number*'.
3529	Invalid decimal (BCD) sign.

Code #	Error Message Text
3530	The number *'number'* cannot be converted to a date.
4100	You have exceeded the limit on the number of connection and statement handles.
4101	The connection, statement, or query handle you provided is not valid.
4102	LIKE or NOT LIKE requested for a non-character data type
4103	You provided an invalid column number. Column numbers must be between 1 and the number of columns returned by the SELECT statement.
4104	The information you requested for a column is not relevant given its data type.
4105	You have exceeded the limit on the number of active programs that can use DTK.
4106	You must retrieve the values for columns in increasing column number order, e.g. column 1, then 2, then 3, etc. You cannot retrieve the value for a column more than once
4110	The last parameter to qeValChar or qeValCharBuf must be zero if the underlying data type is not a character string
4111	You cannot call qeBindCol after you have called qeFetchNext, qeFetchPrev, qeFetchRandom, or qeFetchNumRecs
4112	You did not call qeBindCol for every column in the Select statement.
4114	The database system you are connected to does not support transactions.
4117	You must call qeBeginTran to begin a transaction before you can call qeCommit or qeRollback.
4118	You already have an active transaction. Call qeCommit or qeRollback to end the active transaction.
4119	You have not given an SQL statement to be executed
4120	Tracing has already been turned on.

Code #	Error Message Text
4121	The trace file name is not valid.
4122	Tracing is not turned on.
4123	You must call qeFetchSetOptions before calling a qeBindCol or qeFetch function.
4125	You cannot use qeFetchPrev, qeFetchRandom, or qeFetchNumRecs without first calling qeSetSelectOptions to enable random fetching.
4127	You can only call this function if the current statement is a Select statement.
4128	This evaluation copy of DTK has expired. Call INTERSOLV at (800) 547-4000 to purchase the product.
4129	You can only call this function if there is an active record
4130	This evaluation copy of DTK will expire within the next two weeks. Call INTERSOLV at (800) 547-4000 to purchase the product
4131	You cannot change this column's value. '*reason*'
4132	Attempt to get column attribute that does not exist for this table
4133	Dictionary query is not allowed for this function.
4134	Invalid option specified: '*invalid_option*'.
4135	Statement has not been executed or is not positioned on a row
4136	Row to be locked has changed.
4137	Multiple rows locked.
4138	No rows locked.
4139	The specified column is not searchable.
4140	No database source has been specified.
4141	The parameter number supplied '*number*' is invalid.
4142	Field number supplied ('*number*') is invalid.
4143	Missing keyword: '*keyword*'

Code #	Error Message Text
4144	This statement hasn't been executed. Execution required for this operation.
4145	You must call qeBeginTran to begin a transaction before you can call qeRecLock.
4146	No query save file specified.
4147	Cannot insert row '*row_num*' because it isn't within the rows you have fetched ('*num_fetched_rows*') or immediately following the last row you have fetched.
4148	Parameter type ('*param_type*') not in range of 1 to 6.
4149	error_text
4150	warning_text
4151	HSTMT was invalidated at end of transaction.
4152	Not currently positioned on a row.
4153	Fetching on this statement cannot occur until the transaction has been committed or rolled back.
4154	Query does not have a valid hdbc.
4155	Operation only allowed with deferred auto-update.
4156	Parameter *'param'* in the SQL statement is un-named.
4157	DTK parameter '*param'* doesn't have a name.
4158	No DTK parameter for *'param'*.
4159	DTK parameter *'param'* not found in SQL statement.
4160	qeFetchSetOptions is an obsolete function and is not supported for statements with parameters that haven't been bound or set
4161	The specified column may not yield an exact match because of the database's internal data representation.
4162	Locking is not supported. This is due to either a driver limitation or your current isolation level.
4163	Query Builder was canceled.

Code #	Error Message Text
4164	Unable to grow database greater than 30 records using a demo version.
4165	The handle ('*handle*') is being used by another task.
4166	qeVal functions may not be used with this hstmt because qeBindCol has been used on this hstmt.
4167	Unable to create a new handle because handle('*handle*') is still active.
4168	Parameter ('*param*') not found.
4169	Unable to allocate buffer as large as the max_len passed to qeValChar or qeValCharBuf.
4170	Unable to exit after first dialog of the Query Builder for an hqry without a SQL statement.
4171	Unable to qeSetParamDataType parameter('*param*') because it is not of IO type qePARAM_OUTPUT.
4172	qeGetParam functions may not be used with parameter('*param*') because qeBindParam has been used on this parameter.
4173	The last parameter to qeGetParamChar or qeGetParamCharBuf must be zero if the underlying data type is not a character string
4174	Unable to qeGetParam parameter('*param*') because it is of IO type qePARAM_INPUT
4175	Unable to set use the ODBC connection or statement handle
4500	Missing keyword: '*keyword*'
4501	Unexpected text at end of SQL query: '*text*'
4502	Empty SQL clause found.
4503	Missing matching */ in comment.
4504	Improper select list in SELECT statement: '*bad_list*'
4505	You did not give a SQL statement to execute.
4506	Cannot update or delete record, no primary key available

Code #	Error Message Text
4507	Operation aborted!
4508	Duplicate table names not allowed in FROM clause: '*dup_table_names*'
4510	Unable to lock this record. It has been modified or deleted by another user.
4511	Unable to insert record into database.
4513	The number of parameters supplied does not match the number of parameter markers in the statement.
4514	The declared parameter names don't match the statement parameters.
4515	You cannot delete the current record from a query containing a join
4516	You cannot insert a record into a query containing a join
4517	You cannot add another break to this field: '*field_name*'
4518	You cannot read backwards without a logfile.
4519	Unable to build select list.
4520	You cannot modify a read-only query.
4522	Field number supplied ('*field_num*') is too large.
4523	Case-insensitive search requested on a non-character column
4524	This statement hasn't been executed. Execution required for this operation.
4525	Invalid ODBC handle --- internal error.
4526	A table or table alias name exceeds '*max_chars*' characters.
4527	The parameter number supplied '*param_num*' is too large.
4528	At least one parameter has not been supplied a value
4529	Fixing bind and set is not allowed for multiple value parameters
4530	End of results.

Code #	Error Message Text
4531	The value of a field is longer than the maximum length that can be stored in the database or generation of a SQL statement has run out of space (> 65000 bytes).
4532	Fetching no longer allowed on this statement, probably due to a previous update or delete. Enabling logging will probably fix the error.
4533	Parameter '*param_num*' hasn't been given a data type.
4534	Attempt to change the data type of parameter'*param_num*'.
4537	Inserting row '*row_num*' is illegal with current row ('*row_num*').
4538	Attempt to insert '*num*' characters into a column that allows '*num*' characters.
4539	*'statement'* is invalid in a select statement.
4540	Database does not support an uppercase function
4541	Internal error 00 -- contact INTERSOLV Technical Support
4542	Parameter input was canceled--statement was not executed
4543	Internal error 01 -- contact INTERSOLV Technical Support
4544	You have exceeded the limit on SQL statements allowed by this demo version. To reset the SQL statement counter, exit and then restart your application.
4545	You may not modify this column because we were unable to determine the table for this column.
4546	The column's (*'p1'*) precision of *'p2'* exceeds the limit of *'p3'*.
4547	The parameter *'param'* does not appear in the SQL statement.
9000	Attempted linkout from empty list.
9001	Attempted get from empty list.
9002	Attempted update of empty list.
9003	Seek to item not on list.
9004	Not enough memory for List.

Code #	Error Message Text
9006	Unable to complete operation--out of memory!
9007	The Query you entered is either incorrect or too complex to be understood by the Query Builder. Click the Error Check icon to check for errors.
9009	Expression has changed--save?
9010	Memory allocation error in Query parser.
9011	An unterminated comment was found.
9012	SQL statement must begin with SELECT.
9013	No select list in Query.
9014	No from clause in Query.
9015	The BY was missing from GROUP BY clause.
9016	The BY was missing from ORDER BY clause.
9017	A empty clause was found.
9019	An unsupported feature(e.g. UNION) was found.
9020	A wildcard had incorrect format in select list.
9021	Unmatched parens found in ORDER BY clause.
9022	Unmatched parens found in GROUP BY clause.
9023	Format of BETWEEN incorrect in WHERE clause.
9024	Format of BETWEEN incorrect in HAVING clause.
9025	No value found after operator in WHERE clause.
9026	No value found after operator in HAVING clause.
9027	Unmatched parens found in WHERE clause.
9028	Unmatched parens found in HAVING clause.
9029	Warning--no fields found for:
9030	Joins only valid with two or more tables.
9031	Must specify grouping before you can edit grouping conditions

Code #	Error Message Text
9032	Error--incompatible data types.
9033	Error--unmatched quote found in string.
9034	When comparing against NULL, use 'Is' or 'Is not'.
9035	Missing = after CHARSET, DELIMITER, or PARSE
9036	Missing val after headerline.
9037	Missing = after HEADERLINE.
9038	Data type syntax error.
9039	Syntax error in stmt.
9040	Missing right paren.
9041	Missing paren or comma.
9042	Bad parse string.
9043	Unknown data type.
9044	Number width invalid.
9045	Character width invalid.
9046	Decimal greater than width.
9047	File options missing paren or too large (> 1024).
9048	There are currently no parameters to edit.
9049	Parameter markers are only valid in the WHERE and HAVING clauses.
9050	Parameter name must begin with a letter, be alphanumeric, and have no blanks.
9051	No fields have been chosen.
9052	Field aliases are not allowed.
9053	Grouping change requires select list to change.
9054	Operator is required.
9056	Parameter name already used.

Code #	Error Message Text
9057	Parameter must have a name.
9058	To compare against NULL, enter 'NULL' for the value
9059	Queries containing parameters must be validated by executing the statement.
9060	A NULL value is not allowed.
9061	Unable to parse SQL. Database currently unknown to query builder.
9062	Expression is too long.
9063	Query Builder error #
9064	Query Builder warning #
9065	Database error #
9066	Invalid logical value.
9067	Only one table is allowed to be entered at a time
9068	Invalid table entry. There are too many spaces
9069	The alias you specified conflicts with a previously used table. Use another alias.
9070	The table name you specified conflicts with a previously used alias. Use an alias for this table.
9071	The alias you specified has already been used
9072	The table name is too large.
9073	No functions have been defined for this data source
9074	Alias name is invalid.
9075	Warning--the left hand column is ambiguous. First table found was selected.
9076	Warning--the right hand column is ambiguous. First table found was selected.
9077	Warning--the left hand column wasn't found in any table

Code #	Error Message Text
9078	Warning--the right hand column wasn't found in any table
9080	Unmatched parens found in FOR UPDATE OF clause
9081	Invalid number - must be integer between 0 and 65535
9082	Save Failed.
9083	Multi-table queries (joins) are not allowed.
9084	This datasource does not support GROUP BYs.
10028	Cannot access drive.
30040	Cannot open file '*file_name*'.
30041	Error on input or output to a file.
30042	Cannot rename '*file_name1*' to '*file_name2*'.
30043	Not enough memory for this command.
30045	The maximum number of files are already open.
30047	Reserved file name cannot be opened '*file_name*'.
30049	File system is Read Only.
30050	Need Additional Information.
30051	Out of file handles. Cannot open file '*file_name*'.
30190	Out of memory.

Go To

E Compatibility Issues

This appendix contains information about the compatibility of DTK Version 2.*x* with QELIB 1.0, with future versions of DTK, and with ODBC database drivers. It contains the following sections:

- "QELIB 1.0 Compatibility," describes the differences between DTK Version 2.*x* functionality and the functionality provided for applications developed using QELIB Version 1.0.
- "Obsolete QELIB Functions" on page 556 describes the qeFetchGetOptions and qeFetchSetOptions functions, which have been replaced by the new qeSetSelectOptions and qeGetSelectOptions functions. These functions will not be supported in future versions of DTK.
- "ODBC Compatibility" on page 560 lists the ODBC functions that must be supported by database drivers used with DTK and specific DTK functions.

QELIB 1.0 Compatibility

With DTK, applications developed using QELIB 1.0 can be run using version 2.*x*. However, because certain changes made for version 2.*x* create incompatibilities with version 1.0 applications, you must specify when you want to take advantage of version 1.0 compatibility by setting Revision = 1 in the QELIB.INI file. For information on the QELIB.INI file, see Appendix F, "The QELIB.INI File," on page 565.

When you specify Revision 1 compatibility in the QELIB.INI file, DTK 2.*x* functions differently to support QELIB 1.0 behavior in the following areas:

- Native column type support
- Column width support
- Error checking
- SQL compatibility
- Issuing multiple SQL statements
- Character string values returned from SQL Server

The following sections describe each of these differences.

Note to OS/2 users In order to run your existing DTK applications using DTK 2.*x*, you must first recompile them as 32-bit applications.

Native Column Type Suppot

In DTK 2.*x*, the qeColDBType function does not support the data type values of 1000 or greater that were returned by qeDBColType in QELIB version 1.0. However, DTK will support these values when you specify Revision 1 compatibility.

Column Width Suppot

In QELIB 1.0, the qeColWidth function could not return column width values greater than 32K (32,760 bytes). This column width restriction continues to apply when you specify Revision 1 compatibility. In DTK 2.*x*, the qeColWidth function can return column widths up to 2^{31} bytes.

You should note this change if upgrading QELIB 1.0 applications that use the qeColWidth function to allocate memory, since this function can now return width values that exceed the operating system’s ability to allocate memory.

Error Checking

Because the qeWarning function returns the values qeTRUNCATION (-1) and qeNULL_DATA (-2), qeErr does not return them in DTK 2.*x*. When you choose Revision 1 compatibility, the qeErr function returns these values as errors.

SQL Compatibility

The ODBC-compliant drivers used with Version 2.*x* of DTK support ANSI-standard SQL, which they modify into the SQL dialect used in the database system. This makes DTK 2.*x* applications portable among ODBC database drivers. In QELIB 1.0, however, database system-specific SQL statements are passed to the underlying database system without modification. Therefore, SQL statements issued in QELIB 1.0 applications may be incompatible with the ODBC drivers when you specify Revision 2 compatibility. When you specify Revision 1 compatibility, however, DTK adds a connection string setting, MODIFYSQL=0, that allows database-specific SQL to be passed unmodified through the ODBC drivers.

Issuing Multiple SQL Statements

DTK 2.*x* provides the qeMoreResults function for moving to the next set of results from multiple SQL statements and stored procedures. When using a Revision 2 compatibility setting, you must call qeMoreResults to retrieve the results of each statement executed, regardless of whether it was a Select, Update, Delete, or other type of statement. Use the Revision 1 setting to enable the QELIB 1.0 behavior for multiple statement results—where DTK continues to execute SQL statements until it returns a result set from a Select statement.

SQL Server Character Strings

In QELIB 1.0, fixed length character string values were returned from SQL Server as varying length character strings with the blanks removed. DTK continues this behavior when you specify Revision 1 compatibility. In DTK 2.*x*, these fixed-length character strings are returned as fixed length, blank-padded.

Obsolete QELIB Functions

The qeFetchSetOptions and qeFetchGetOptions functions are still supported for compatibility with QELIB 1.0 applications, but will not be supported in future versions of DTK. It is not recommended that you use these functions. Instead, use the qeSetSelectOptions and qeGetSelectOptions functions, which operate on the current *hdbc* instead of the current *hstmt*.

qeFetchGetOptions

qeFetchGetOptions returns the fetch options that were set with the previous call to qeFetchSetOptions.

Syntax

```
int32 options qeFetchGetOptions (int16   hstmt)
```

Parameters

hstmt is the handle to the statement returned by qeExecSQL.

options are the returned option flag values.

Example

To set the fetch options and then retrieve them:

```
hdbc = qeConnect ("DSN=QEDBF")   ;
...
hstmt = qeExecSQL (hdbc, "SELECT * FROM emp")    ;
res_code = qeFetchSetOptions (hstmt,1)   ;
...
options = qeFetchGetOptions (hstmt)   ;
/* Returns 1 in this case *   /
res_code = qeEndSQL (hstmt)   ;
res_code = qeDisconnect (hdbc)   ;
```

qeFetchSetOptions

qeFetchSetOptions sets options that determine which functions you can use to retrieve records.

Syntax

```
int16 res_code qeFetchSetOptions (int16   hstmt, int32
options)
```

Description

qeFetchSetOptions sets options that determine which qeFetch functions can be used to retrieve records. If qeFetchSetOptions is not called, only qeFetchNext can be used to retrieve records. If you wish to use qeFetchPrev, qeFetchRandom, or qeFetchNumRecs, you must call this function to enable their use.

qeFetchSetOptions must be called immediately after calling qeExecSQL or qeSQLExecute and before calling qeBindCol or any other qeFetch function.

You can call qeFetchSetOptions only once for a given *hstmt*.

Calling qeFetchSetOptions on data dictionary queries returns an error.

Parameters

hstmt is the handle to the statement returned by qeExecSQL.

options is the set of options to be enabled. The separate options have a value associated with them. To set more than one option, add the values together. The individual option values are as follows:

Constant	Value	Description
qeFETCH_FORWARD	0x0000	The default; only forward fetching allowed.
qeFETCH_RANDOM	0x0001	Allows the use of qeFetchPrev, qeFetchRandom, and qeFetchNumRecs.
qeFORCE_LOG	0x0002	Forces the use of temporary log files for database systems that do not require them.

res_code is the result code returned by qeFetchSetOptions, which returns the same set of result codes as qeErr. See Appendix D, “Result and Error Message Codes,” on page 537 for a list of these result codes.

Example

To set the fetch options to enable the use of qeFetchPrev, qeFetchRandom, and qeFetchNumRecs, and to retrieve the last record selected:

```
hdbc = qeConnect("DSN=QEDBF")   ;
...
hstmt = qeExecSQL(hdbc, "SELECT * FROM emp")    ;
res_code = qeFetchSetOptions(hstmt,1)   ;
num_recs = qeFetchNumRecs(hstmt)   ;
res_code = qeFetchRandom(hstmt, num_recs)    ;
/* Code to use the values in the record *    /
res_code = qeEndSQL (hstmt)   ;
res_code = qeDisconnect (hdbc)   ;
```

Notes

Most of the database systems DTK supports provide only a fetch next function, neither previous nor random fetches are permitted. Also, the database systems do not provide a function that returns the number of records selected. If you call qeFetchSetOptions to enable these functions, DTK creates a temporary log file in your TEMP directory (specified by the 'SET TEMP=' line in your DOS AUTOEXEC.BAT or OS/2 CONFIG.SYS file). Every record read from the database is saved in the temporary file so that DTK can support the qeFetchPrev, qeFetchRandom, and qeFetchNumRecs functions. The temporary log file is deleted when qeEndSQL is called.

If you call qeFetchSetOptions to enable the functions, you must have sufficient disk space available to hold copies of the records selected from the database.

If you call qeFetchNumRecs, DTK retrieves every record chosen by your Select statement and copies it to the temporary log files. DTK determines the number of records by counting the number of records retrieved. This operation can be slow for queries that return a large number of records.

Since there are a limited number of files that an application can have open at any time (20 is the DOS/Windows default), you may exceed the limit if your application has other files open or if you have several Select statements active at the same time. You can call qeFetchLogClose to close the temporary log file used by a statement. DTK automatically re-opens the files when you call a qeFetch function.

To increase the number of files that DTK can have open at one time, DTK sets the limit to 200 by calling the Windows SetHandleCount function or the OS/2 DosSetMaxFH function. If your application may exceed the default number of file opens, it is recommended that your application also call these system functions.

DTK's Btrieve, dBASE, Paradox, text, and Excel file database drivers do not require temporary log files to support the qeFetchPrev, qeFetchRandom, and qeFetchNumRecs functions. If you enable the functions by calling qeFetchSetOptions, DTK does not create temporary log files for these database systems. If you want to force the use of temporary files for these database systems, set the *options* parameter to 3 (1 to enable the functions + 2 to force the log file).

For all other database systems, you only need to set the *options* parameter to 1, since DTK must create the temporary log files for these systems.

ODBC Compatibility

DTK uses the ODBC API to communicate with database drivers. This section lists the ODBC functions that DTK uses. You should be aware of these compatibility issues when using ODBC database drivers other than those supplied by INTERSOLV.

Required Functions

DTK will not run if the database driver does not implement the following ODBC functions:

Core Compliance

The following are required Core functions:

SQLAllocConnect	SQLExecute
SQLAllocEnv	SQLFetch
SQLAllocStmt	SQLFreeStmt
SQLBindCol	SQLGetCursorName
SQLColAttributes	SQLNumResultCols
SQLDescribeCol	SQLPrepare
SQLDisconnect	SQLRowCount
SQLError	SQLSetParam
SQLExecDirect	

Level 1 Compliance

The following are required Level 1 functions:

SQLColumns	SQLGetTypeInfo
SQLDriverConnect	SQLParamData
SQLGetData	SQLPutData
SQLGetFunctions	SQLSetConnectOption
SQLGetInfo	SQLSetStmtOption

Optional Functions

These functions are used by DTK, but can be absent in a driver. If a driver does not support these functions, pieces of DTK functionality will not work. This information is listed by function.

Core Compliance

There is one optional Core function, as follows:

SQLTransact	Failure to implement this function will cause qeBeginTran, qeCommit and qeRollback to be unsupported.

Level 1 Compliance

The following level 1 functions are optional:

SQLSpecialColumns	DTK will use this function if it is available, but no functions are disabled if it is not.
SQLTables	Failure to implement this function will cause qeTables to fail, and will make the Query Builder unable to populate the table name list box.

Level 2 Compliance

The following level 2 functions are optional:

SQLDataSources	The driver does not have to implement this function, it is provided by ODBC.
SQLExtendedFetch	Failure to implement these functions will result in DTK being unable to take advantage of the native database's ability to fetch records at random.
SQLSetScrollOptions	Failure to implement these functions will result in DTK being unable to take advantage of the native database’s ability to fetch records at random.
SQLMoreResults	Failure to implement this function will cause qeMoreResults to fail.

SQLNativeSql	Failure to implement this function will cause qeNativeSQL to fail.
SQLProcedureColumns	Failure to implement this function will cause qeProcedureColumns to fail.

Go To

F The QELIB.INI File

The QELIB.INI file contains a [QELIB] section containing global tracing and revision level information. It can also contain a section corresponding to each DTK application that runs on your system. The values in these application-specific sections take precedence over the global settings in the [QELIB] section. The application-specific sections also let you specify default connection string values for the corresponding application.

The following sections describe the global and application-specific sections that this file contains.

[QELIB]

In this section, you can specify the following default values for all DTK applications:

TraceOptions = flags

The default tracing options for all DTK applications. This takes the same set of flags as the parameter to the qeSetTraceOptions function:

0x0001	Trace all non-qeVal calls (qeTRACE_NON_VAL_CALLS).
0x0002	Trace strings sent via qeTraceUser (qeTRACE_USER).
0x0004	Trace qeVal calls and bound data at fetch time (qeTRACE_VAL_CALLS).
0x0008	Write all info (except ODBC calls) to a trace window (qeTRACE_WINDOW).

0x0010	Trace ODBC calls (qeTRACE_ODBC).
0x0020	Allows faster tracing by writing trace strings to disk in blocks instead of one at a time (qeTRACE_NO_FLUSH). Choosing this method can cause some loss of trace information if your program terminates abnormally—use it only when your application is reasonably stable.

If you don't specify a different one here, the default when qeTraceOn is called will be 0x0003 (qeTRACE_NON_VAL_CALLS + qeTRACE_USER).

TraceFile = filename

The default file name for DTK trace files. This file name is equivalent to the one passed as a parameter to the qeTraceOn function. Any file name passed to qeTraceOn overrides this setting, but this file name will be used when tracing is enabled by this section of the QELIB.INI file.

Revision = {1|2}

The default revision level support provided by DTK. If you do not specify this setting, DTK defaults to Revision level 2 support. See Appendix E, "Compatibility Issues," on page 553 for information on how this setting affects DTK functionality.

[program]

The QELIB.INI file can contain a section for each DTK application that runs on your system. This section's name is the same as that of the application's executable (.EXE) file, without the .EXE extension. It can specify the following values:

TraceOptions = flags

The default tracing options for this application. This setting takes precedence over any TraceOptions setting in the [QELIB] section, and takes the same set of flags as that setting.

TraceFile = filename

For this application, the default file name for DTK trace files. Any file name passed to the qeTraceOn function in this application overrides this setting, but this file name will be used when tracing is enabled by this section of the QELIB.INI file.

Revision = {1|2}

The Revision level support provided by DTK for this application. See Appendix E, “Compatibility Issues,” on page 553 for information on how this setting affects DTK functionality.

ConnectString = connection_string

Specifies a string that is added to the connection string passed to the qeConnect function. Any connection option passed to qeConnect that contradicts a value in this string takes precedence over the value specified here.

Go To

Index

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W