SYBASE iAnywhere

UltraLite® C and C++ Programming

February 2009

Version 11.0.1

Copyright and trademarks

Contents

About this book

Subject

This book describes UltraLite C and C++ programming interfaces. With UltraLite, you can develop and deploy database applications to handheld, mobile, or embedded devices.

Audience

This book is intended for C and C++ application developers, who want to take advantage of the performance, resource efficiency, robustness, and security of an UltraLite relational database for data storage and synchronization.

About the SQL Anywhere documentation

The complete SQL Anywhere documentation is available in four formats that contain identical information.

- **HTML Help** The online Help contains the complete SQL Anywhere documentation, including the books and the context-sensitive help for SQL Anywhere tools.

  If you are using a Microsoft Windows operating system, the online Help is provided in HTML Help (CHM) format. To access the documentation, choose **Start** » **Programs** » **SQL Anywhere 11** » **Documentation** » **Online Books**.

  The administration tools use the same online documentation for their Help features.

- **Eclipse** On Unix platforms, the complete online Help is provided in Eclipse format. To access the documentation, run *sadoc* from the *bin32* or *bin64* directory of your SQL Anywhere 11 installation.

- **DocCommentXchange** DocCommentXchange is a community for accessing and discussing SQL Anywhere documentation.

  Use DocCommentXchange to:

  - View documentation
  - Check for clarifications users have made to sections of documentation
  - Provide suggestions and corrections to improve documentation for all users in future releases

  Visit http://dcx.sybase.com.

- **PDF** The complete set of SQL Anywhere books is provided as a set of Portable Document Format (PDF) files. You must have a PDF reader to view information. To download Adobe Reader, visit http://get.adobe.com/reader/.

  To access the PDF documentation on Microsoft Windows operating systems, choose **Start** » **Programs** » **SQL Anywhere 11** » **Documentation** » **Online Books - PDF Format**.

  To access the PDF documentation on Unix operating systems, use a web browser to open *install-dir/documentation/en/pdf/index.html*.

About the books in the documentation set

The SQL Anywhere documentation consists of the following books:

- **SQL Anywhere 11 - Introduction** This book introduces SQL Anywhere 11, a comprehensive package that provides data management and data exchange, enabling the rapid development of database-powered applications for server, desktop, mobile, and remote office environments.
- **SQL Anywhere 11 - Changes and Upgrading** This book describes new features in SQL Anywhere 11 and in previous versions of the software.
- **SQL Anywhere Server - Database Administration** This book describes how to run, manage, and configure SQL Anywhere databases. It describes database connections, the database server, database

files, backup procedures, security, high availability, replication with the Replication Server, and administration utilities and options.

- **SQL Anywhere Server - Programming** This book describes how to build and deploy database applications using the C, C++, Java, PHP, Perl, Python, and .NET programming languages such as Visual Basic and Visual C#. A variety of programming interfaces such as ADO.NET and ODBC are described.
- **SQL Anywhere Server - SQL Reference** This book provides reference information for system procedures, and the catalog (system tables and views). It also provides an explanation of the SQL Anywhere implementation of the SQL language (search conditions, syntax, data types, and functions).
- **SQL Anywhere Server - SQL Usage** This book describes how to design and create databases; how to import, export, and modify data; how to retrieve data; and how to build stored procedures and triggers.
- **MobiLink - Getting Started** This book introduces MobiLink, a session-based relational-database synchronization system. MobiLink technology allows two-way replication and is well suited to mobile computing environments.
- **MobiLink - Client Administration** This book describes how to set up, configure, and synchronize MobiLink clients. MobiLink clients can be SQL Anywhere or UltraLite databases. This book also describes the Dbmlsync API, which allows you to integrate synchronization seamlessly into your C++ or .NET client applications.
- **MobiLink - Server Administration** This book describes how to set up and administer MobiLink applications.
- **MobiLink - Server-Initiated Synchronization** This book describes MobiLink server-initiated synchronization, a feature that allows the MobiLink server to initiate synchronization or perform actions on remote devices.
- **QAnywhere** This book describes QAnywhere, which is a messaging platform for mobile, wireless, desktop, and laptop clients.
- **SQL Remote** This book describes the SQL Remote data replication system for mobile computing, which enables sharing of data between a SQL Anywhere consolidated database and many SQL Anywhere remote databases using an indirect link such as email or file transfer.
- **UltraLite - Database Management and Reference** This book introduces the UltraLite database system for small devices.
- **UltraLite - C and C++ Programming** This book describes UltraLite C and C++ programming interfaces. With UltraLite, you can develop and deploy database applications to handheld, mobile, or embedded devices.
- **UltraLite - M-Business Anywhere Programming** This book describes UltraLite for M-Business Anywhere. With UltraLite for M-Business Anywhere you can develop and deploy web-based database applications to handheld, mobile, or embedded devices, running Palm OS, Windows Mobile, or Windows.
- **UltraLite - .NET Programming** This book describes UltraLite.NET. With UltraLite.NET you can develop and deploy database applications to computers, or handheld, mobile, or embedded devices.
- **UltraLiteJ** This book describes UltraLiteJ. With UltraLiteJ, you can develop and deploy database applications in environments that support Java. UltraLiteJ supports BlackBerry smartphones and Java SE environments. UltraLiteJ is based on the iAnywhere UltraLite database product.

- **Error Messages** This book provides a complete listing of SQL Anywhere error messages together with diagnostic information.

Documentation conventions

This section lists the conventions used in this documentation.

Operating systems

SQL Anywhere runs on a variety of platforms. In most cases, the software behaves the same on all platforms, but there are variations or limitations. These are commonly based on the underlying operating system (Windows, Unix), and seldom on the particular variant (AIX, Windows Mobile) or version.

To simplify references to operating systems, the documentation groups the supported operating systems as follows:

- **Windows** The Microsoft Windows family includes Windows Vista and Windows XP, used primarily on server, desktop, and laptop computers, and Windows Mobile used on mobile devices.

  Unless otherwise specified, when the documentation refers to Windows, it refers to all Windows-based platforms, including Windows Mobile.
- **Unix** Unless otherwise specified, when the documentation refers to Unix, it refers to all Unix-based platforms, including Linux and Mac OS X.

Directory and file names

In most cases, references to directory and file names are similar on all supported platforms, with simple transformations between the various forms. In these cases, Windows conventions are used. Where the details are more complex, the documentation shows all relevant forms.

These are the conventions used to simplify the documentation of directory and file names:

- **Uppercase and lowercase directory names** On Windows and Unix, directory and file names may contain uppercase and lowercase letters. When directories and files are created, the file system preserves letter case.

  On Windows, references to directories and files are *not* case sensitive. Mixed case directory and file names are common, but it is common to refer to them using all lowercase letters. The SQL Anywhere installation contains directories such as *Bin32* and *Documentation*.

  On Unix, references to directories and files *are* case sensitive. Mixed case directory and file names are not common. Most use all lowercase letters. The SQL Anywhere installation contains directories such as *bin32* and *documentation*.

  The documentation uses the Windows forms of directory names. In most cases, you can convert a mixed case directory name to lowercase for the equivalent directory name on Unix.
- **Slashes separating directory and file names** The documentation uses backslashes as the directory separator. For example, the PDF form of the documentation is found in *install-dir\Documentation\en\PDF* (Windows form).

On Unix, replace the backslash with the forward slash. The PDF documentation is found in *install-dir/documentation/en/pdf*.

- **Executable files** The documentation shows executable file names using Windows conventions, with a suffix such as *.exe* or *.bat*. On Unix, executable file names have no suffix.

  For example, on Windows, the network database server is *dbsrv11.exe*. On Unix, it is *dbsrv11*.

- ***install-dir*** During the installation process, you choose where to install SQL Anywhere. The environment variable SQLANY11 is created and refers to this location. The documentation refers to this location as *install-dir*.

  For example, the documentation may refer to the file *install-dir*\readme.txt. On Windows, this is equivalent to *%SQLANY11%\readme.txt*. On Unix, this is equivalent to *$SQLANY11/readme.txt* or *${SQLANY11}/readme.txt*.

  For more information about the default location of *install-dir*, see "SQLANY11 environment variable" [*SQL Anywhere Server - Database Administration*].

- ***samples-dir*** During the installation process, you choose where to install the samples included with SQL Anywhere. The environment variable SQLANYSAMP11 is created and refers to this location. The documentation refers to this location as *samples-dir*.

  To open a Windows Explorer window in *samples-dir*, from the **Start** menu, choose **Programs** » **SQL Anywhere 11** » **Sample Applications And Projects**.

  For more information about the default location of *samples-dir*, see "SQLANYSAMP11 environment variable" [*SQL Anywhere Server - Database Administration*].

Command prompts and command shell syntax

Most operating systems provide one or more methods of entering commands and parameters using a command shell or command prompt. Windows command prompts include Command Prompt (DOS prompt) and 4NT. Unix command shells include Korn shell and bash. Each shell has features that extend its capabilities beyond simple commands. These features are driven by special characters. The special characters and features vary from one shell to another. Incorrect use of these special characters often results in syntax errors or unexpected behavior.

The documentation provides command line examples in a generic form. If these examples contain characters that the shell considers special, the command may require modification for the specific shell. The modifications are beyond the scope of this documentation, but generally, use quotes around the parameters containing those characters or use an escape character before the special characters.

These are some examples of command line syntax that may vary between platforms:

- **Parentheses and curly braces** Some command line options require a parameter that accepts detailed value specifications in a list. The list is usually enclosed with parentheses or curly braces. The documentation uses parentheses. For example:

  ```
  -x tcpip(host=127.0.0.1)
  ```

  Where parentheses cause syntax problems, substitute curly braces:

  ```
  -x tcpip{host=127.0.0.1}
  ```

If both forms result in syntax problems, the entire parameter should be enclosed in quotes as required by the shell:

```
-x "tcpip(host=127.0.0.1)"
```

- **Quotes** If you must specify quotes in a parameter value, the quotes may conflict with the traditional use of quotes to enclose the parameter. For example, to specify an encryption key whose value contains double-quotes, you might have to enclose the key in quotes and then escape the embedded quote:

  ```
  -ek "my \"secret\" key"
  ```

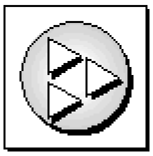
  In many shells, the value of the key would be `my "secret" key`.

- **Environment variables** The documentation refers to setting environment variables. In Windows shells, environment variables are specified using the syntax *%ENVVAR%*. In Unix shells, environment variables are specified using the syntax *$ENVVAR* or *${ENVVAR}*.

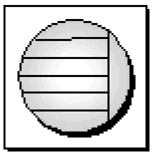
Graphic icons

The following icons are used in this documentation.

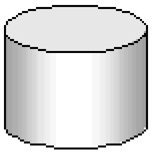
- A client application.



- A database server, such as Sybase SQL Anywhere.



- A database. In some high-level diagrams, the icon may be used to represent both the database and the database server that manages it.



- Replication or synchronization middleware. These assist in sharing data among databases. Examples are the MobiLink server and the SQL Remote Message Agent.

- A programming interface.

Interface

Contacting the documentation team

We would like to receive your opinions, suggestions, and feedback on this Help.

To submit your comments and suggestions, send an email to the SQL Anywhere documentation team at iasdoc@sybase.com. Although we do not reply to emails, your feedback helps us to improve our documentation, so your input is welcome.

DocCommentXchange

You can also leave comments directly on help topics using DocCommentXchange. DocCommentXchange (DCX) is a community for accessing and discussing SQL Anywhere documentation. Use DocCommentXchange to:

- View documentation
- Check for clarifications users have made to sections of documentation
- Provide suggestions and corrections to improve documentation for all users in future releases

Visit http://dcx.sybase.com.

Finding out more and requesting technical support

Additional information and resources are available at the Sybase iAnywhere Developer Community at http://www.sybase.com/developer/library/sql-anywhere-techcorner.

If you have questions or need help, you can post messages to the Sybase iAnywhere newsgroups listed below.

When you write to one of these newsgroups, always provide details about your problem, including the build number of your version of SQL Anywhere. You can find this information by running the following command: **dbeng11 -v**.

The newsgroups are located on the *forums.sybase.com* news server.

The newsgroups include the following:

- sybase.public.sqlanywhere.general
- sybase.public.sqlanywhere.linux
- sybase.public.sqlanywhere.mobilink
- sybase.public.sqlanywhere.product_futures_discussion
- sybase.public.sqlanywhere.replication
- sybase.public.sqlanywhere.ultralite
- ianywhere.public.sqlanywhere.qanywhere

For web development issues, see http://groups.google.com/group/sql-anywhere-web-development.

Newsgroup disclaimer

iAnywhere Solutions has no obligation to provide solutions, information, or ideas on its newsgroups, nor is iAnywhere Solutions obliged to provide anything other than a systems operator to monitor the service and ensure its operation and availability.

iAnywhere Technical Advisors, and other staff, assist on the newsgroup service when they have time. They offer their help on a volunteer basis and may not be available regularly to provide solutions and information. Their ability to help is based on their workload.

UltraLite for C/C++ developers

Contents

The C and C++ interface provides the following benefits for UltraLite developers targeting small devices:

- A small, high-performance database store.
- The power, efficiency, and flexibility of the C or C++ language.
- The ability to deploy an application on Windows Mobile, Palm OS, and Windows desktop platforms.

For more information about the features of UltraLite databases, see "Creating and configuring UltraLite databases" [*UltraLite - Database Management and Reference*].

UltraLite developers that use C++ have two options available:

- The UltraLite C++ API.
- The ODBC programming interface (component interface).

UltraLite developers that use C must use Embedded SQL or the ODBC programming interface.

Developing embedded SQL applications

When developing embedded SQL applications, you mix SQL statements with standard C or C++ source code. To develop embedded SQL applications you should be familiar with the C or C++ programming language.

The development process for embedded SQL applications is as follows:

1. Create your UltraLite database.
2. Write your source code in an embedded SQL source file, which typically has extension *.sqc*.

   When you need data access in your source code, use the SQL statement you want to execute, prefixed by the EXEC SQL keywords. For example:

   ```
   EXEC SQL SELECT price, prod_name
      INTO :cost, :pname
      FROM ULProduct
      WHERE prod_id= :pid;
   if((SQLCODE==SQLE_NOTFOUND)||(SQLCODE<0)) {
     return(-1);
   }
   ```
3. Preprocess the *.sqc* files.

   SQL Anywhere includes a SQL preprocessor (sqlpp), which reads the *.sqc* files and generates *.cpp* files. These files hold function calls to the UltraLite runtime library.
4. Compile your *.cpp* files.
5. Link the *.cpp* files.

   You must link the files with the UltraLite runtime library.

For more information about embedded SQL development, see "Building embedded SQL applications" on page 63.

System requirements and supported platforms

Development platforms

To develop applications using UltraLite C++, you require the following:

- A Microsoft Windows desktop as a development platform.
- A supported C/C++ compiler.

Target platforms

UltraLite C/C++ supports the following target platforms:

- Windows Mobile 3.0 or later
- Palm OS 4.0 or later

For more information about supported target platforms, see http://www.sybase.com/detail?id=1002288.

UltraLite C++ Component architecture

The UltraLite C++ component interface is defined in the *uliface.h* header file. The following list describes some of the commonly used objects:

- **DatabaseManager** Create one DatabaseManager object for each application.
- **Connection** Represents a connection to an UltraLite database. You can create one or more Connection objects.
- **Table** Provides access to the data in the database.
- **PreparedStatement, ResultSet, and ResultSetSchema** Create Dynamic SQL statements, make queries and execute INSERT, UPDATE, and DELETE statements, and attain programmatic control over database result sets.
- **SyncParms** Synchronize your UltraLite database with a MobiLink server.

For more information about accessing the API reference, see "UltraLite C++ API reference" on page 129.

Understanding the SQL Communications Area

All UltraLite C/C++ interfaces utilize the same UltraLite run time engine. The APIs each provide access to the same underlying functionality.

All UltraLite C/C++ interfaces share the same basic data structure for marshaling data between the UltraLite runtime and your application. This data structure is the SQL Communications Area or SQLCA. Each SQLCA has a current connection, and separate threads cannot share a common SQLCA.

Your application code must carry out the following tasks before connecting to a database:

- Initialize a SQLCA. This prepares your application for communication with the UltraLite runtime.
- Register your error callback function.
- Start a database. This operation can be carried out as part of opening the connection.

The following functions are equivalent ways of carrying out these tasks.

Task	Interface	Function
Initialize SQLCA	Embedded SQL	db_init
	C++	ULSqlca::Initialize
Initialize SQLCA and start database	Embedded SQL	db_init db_start_database
	C++	The database is started as part of the connection function in UltraLite_DatabaseManager

Creating databases

An UltraLite database can be created using any of the following methods:

- The **Create Database Wizard** in Sybase Central.
- A command line utility like ulcreate or ulinit.
- Calling the ULCreateDataBase function.

Using Sybase Central, a database can be interactively created with appropriate definitions for the desired tables and other schema-related items.

The ulcreate utility creates an empty database that does not have any tables defined. Applications that create a database by calling ULCreateDatabase need to also execute SQL CREATE statements to create tables and index definitions.

Name the database explicitly

Different interfaces may use different default file names for the database. If you are mixing the interfaces, it is best to always explicitly name the database when creating or connecting. You can do this using the DBN= connection parameter. See "UltraLite DBN connection parameter" [*UltraLite - Database Management and Reference*].

Application Development

This section provides development notes for UltraLite C/C++ programmers.

Developing applications using the UltraLite C++ API

Contents

Using the UltraLite namespace

The UltraLite C++ interface provides a set of classes with names that are prefixed by UltraLite_ (for example, UltraLite_Connection and UltraLite_DatabaseManager). Most of the functions for each of these classes implement a function from an underlying interface with the string _iface appended to it. For example, the UltraLite_Connection class implements functions from UltraLite_Connection_iface.

When you explicitly use the UltraLite namespace, you can use a shorter name to refer to each class. Instead of declaring a connection as an UltraLite_Connection object, you can declare it as a Connection object if you are using the UltraLite namespace:

```
using namespace UltraLite;
ULSqlca sqlca;
sqlca.Initialize();
DatabaseManager * dbMgr = ULInitDatabaseManager(sqlca);
Connection *  conn = UL_NULL;
```

As a result of this architecture, code samples in this chapter use types such as DatabaseManager, Connection, and TableSchema, but links for more information may direct you to UltraLite_DatabaseManager_iface, UltraLite_Connection_iface, and UltraLite_TableSchema_iface, respectively.

Connecting to a database

UltraLite applications must connect to the database before performing operations on its data. This section describes how to connect to an UltraLite database.

You can find sample code in the *samples-dir\UltraLite\CustDB\* directory.

Properties of the Connection object

- **Commit behavior** In the UltraLite C++ API, there is no AutoCommit mode. Each transaction must be followed by a Conn->Commit() statement. See "Managing transactions" on page 23.
- **User authentication** You can change the user ID and password for the application (from the default values of **DBA** and **sql**, respectively) by using methods to grant and revoke connection permissions. Each database can have a maximum of four user IDs. See "Authenticating users" on page 26.
- **Synchronization** You can synchronize an UltraLite database with a consolidated database by using methods of the Connection object. See "Synchronizing data" on page 28.
- **Tables** UltraLite database tables are accessed using methods of the Connection object. See "Accessing data with the table API" on page 17.
- **Prepared statements** Methods are provided to handle the execution of SQL statements. See "Accessing data using SQL" on page 13 and "UltraLite_PreparedStatement class" on page 202.

Connecting to an UltraLite database

To connect to an UltraLite database

1. Use the UltraLite namespace.

   Using the UltraLite namespace allows you to use simple names for classes in the C++ interface.

   ```
   using namespace UltraLite;
   ```

2. Create and initialize a DatabaseManager object and an UltraLite SQL communications area (ULSqlca). The ULSqlca is a structure that handles communication between the application and the database.

   The DatabaseManager object is at the root of the object hierarchy. You create only one DatabaseManager object per application. It is often best to declare the DatabaseManager object as global to the application.

   ```
   ULSqlca sqlca;
   sqlca.Initialize();
   DatabaseManager * dbMgr = ULInitDatabaseManager(sqlca);
   ```

   If the application does not require SQL support and directly links the UltraLite runtime, the application can call ULInitDatabaseManagerNoSQL to initialize the ULSqlca. This variant reduces the size of the application.

   See "UltraLite_DatabaseManager_iface class" on page 189.

3. Open a connection to an existing database, or create a new database if the specified database file does not exist. See "OpenConnection function" on page 190.

   UltraLite applications can be deployed with an initial empty database or the application can create the UltraLite database if it does not already exist. Deploying an initial database is the simplest solution;

otherwise, the application must call the ULCreateDatabase function to create the database and must create all the tables the application requires. See "ULCreateDatabase function" on page 106.

```
Connection * conn = dbMgr->OpenConnection( sqlca,
UL_TEXT("dbf=mydb.udb") );
if( sqlca.GetSQLCode() ==
  SQLE_ULTRALITE_DATABASE_NOT_FOUND ) {
    printf( "Open failed with sql code: %d.\n" , sqlca.GetSQLCode() );
  }
}
```

Multi-threaded applications

Each connection and all objects created from it should be used by a single thread. If an application requires multiple threads accessing the UltraLite database, each thread requires a separate connection.

Accessing data using SQL

UltraLite applications can access table data by executing SQL statements or using the Table API. This section describes data access using SQL statements.

For more information about using the Table API, see "Accessing data with the table API" on page 17.

This section explains how to perform the following tasks using SQL:

- Inserting, deleting, and updating rows.
- Retrieving rows to a result set.
- Scrolling through the rows of a result set.

This section does not describe the SQL language. For more information about the SQL language, see "UltraLite SQL statements" [*UltraLite - Database Management and Reference*].

Data manipulation: Insert, Delete, and Update

With UltraLite, you can perform SQL data manipulation by using the ExecuteStatement method (a member of the PreparedStatement class).

See "UltraLite_PreparedStatement class" on page 202.

Referencing parameters in prepared statements
UltraLite indicates query parameters using the ? character. For any INSERT, UPDATE, or DELETE statement, each ? is referenced according to its ordinal position in the prepared statement. For example, the first ? is referred to as parameter 1, and the second as parameter 2.

To insert a row

1. Declare a PreparedStatement.

   ```
   PreparedStatement * prepStmt;
   ```

   See "PrepareStatement function" on page 170.

2. Assign a SQL statement to the PreparedStatement object.

   ```
   prepStmt = conn->PrepareStatement( UL_TEXT("INSERT INTO MyTable(MyColumn)
   values (?)") );
   ```

3. Assign input parameter values for the statement.

   The following code shows a string parameter.

   ```
   prepStmt->SetParameter( 1, UL_TEXT("newValue") );
   ```

4. Execute the prepared statement.

   The return value indicates the number of rows affected by the statement.

```
ul_s_long rowsInserted;
rowsInserted = prepStmt->ExecuteStatement();
```

5. Commit the change.

   ```
   conn->Commit();
   ```

To delete a row

1. Declare a PreparedStatement.

   ```
   PreparedStatement * prepStmt;
   ```

2. Assign a SQL statement to the PreparedStatement object.

   ```
   ULValue sqltext( );
   prepStmt = conn->PrepareStatement( UL_TEXT("DELETE FROM MyTable WHERE
   MyColumn = ?") );
   ```

3. Assign input parameter values for the statement.

   ```
   prepStmt->SetParameter( 1, UL_TEXT("deleteValue") );
   ```

4. Execute the statement.

   ```
   ul_s_long rowsDeleted;
   rowsDeleted = prepStmt->ExecuteStatement();
   ```

5. Commit the change.

   ```
   conn->Commit();
   ```

To update a row

1. Declare a PreparedStatement.

   ```
   PreparedStatement * prepStmt;
   ```

2. Assign a statement to the PreparedStatement object.

   ```
   prepStmt = conn->PrepareStatement(
       UL_TEXT("UPDATE MyTable SET MyColumn1 = ? WHERE MyColumn1 = ?") );
   ```

3. Assign input parameter values for the statement.

   ```
   prepStmt->SetParameter( 1, UL_TEXT("newValue") );
   prepStmt->SetParameter( 2, UL_TEXT("oldValue") );
   ```

4. Execute the statement.

   ```
   ul_s_long rowsUpdated;
   rowsUpdated = prepStmt->ExecuteStatement();
   ```

5. Commit the change.

   ```
   conn->Commit();
   ```

Data retrieval: SELECT

The SELECT statement allows you to retrieve information from the database. When you execute a SELECT statement, the PreparedStatement.ExecuteQuery method returns a ResultSet object.

See "UltraLite_PreparedStatement_iface class" on page 203.

To execute a SELECT statement

1. Create a prepared statement object.

```
PreparedStatement * prepStmt =
    conn->PrepareStatement( UL_TEXT("SELECT MyColumn FROM MyTable") );
```

2. Execute the statement.

   In the following code, the result of the SELECT query contains a string, which is output to the command prompt.

```
#define MAX_NAME_LEN     100
ULValue val;
ResultSet * rs = prepStmt->ExecuteQuery();
while( rs->Next() ){
   char mycol[ MAX_NAME_LEN ];
   val = rs->Get( 1 );
   val.GetString( mycol, MAX_NAME_LEN );
   printf( "mycol= %s\n", mycol );
}
```

Navigating SQL result sets

You can navigate through a result set using methods associated with the ResultSet object.

The result set object provides you with the following methods to navigate a result set:

- **AfterLast** Position immediately after the last row.
- **BeforeFirst** Position immediately before the first row.
- **First** Move to the first row.
- **Last** Move to the last row.
- **Next** Move to the next row.
- **Previous** Move to the previous row.
- **Relative(offset)** Move a specified number of rows relative to the current row, as specified by the signed offset value. Positive offset values move forward in the result set, relative to the current position of the cursor in the result set. Negative offset values move backward in the result set. An offset value of zero does not move the current location, but allows you to repopulate the row buffer.

See "UltraLite_ResultSet_iface class" on page 208.

Result set schema description

The ResultSet->GetSchema method allows you to retrieve schema information about a result set, such as: column names, total number of columns, column scales, column sizes, and column SQL types.

Example

The following example demonstrates how to use the ResultSet.GetSchema method to display schema information in a command prompt.

```
ResultSetSchema * rss = rs->GetSchema();
ULValue val;
char name[ MAX_NAME_LEN ];

for( int i = 1;
     i <= rss->GetColumnCount();
     i++ ){
   val = rss->GetColumnName( i );
   val.GetString( name, MAX_NAME_LEN );
   printf( "id= %d, name= %s \n", i, name );
}
```

See "GetSchema function" on page 166.

Accessing data with the table API

UltraLite applications can access table data by executing SQL statements or using the Table API. This section describes data access using the Table API.

For more information about accessing data by executing SQL statements, see "Accessing data using SQL" on page 13.

This section explains how to perform the following tasks using the Table API:

- Scrolling through the rows of a table.
- Accessing the values of the current row.
- Using find and lookup methods to locate rows in a table.
- Inserting, deleting, and updating rows.

Navigating the rows of a table

The UltraLite C++ API provides you with several methods to navigate a table to perform a wide range of navigation tasks.

The table object provides you with the following methods to navigate a table:

- **AfterLast** Position immediately after the last row.
- **BeforeFirst** Position immediately before the first row.
- **First** Move to the first row.
- **Last** Move to the last row.
- **Next** Move to the next row.
- **Previous** Move to the previous row.
- **Relative(offset)** Move a specified number of rows relative to the current row, as specified by the signed offset value. Positive offset values move forward in the result set, relative to the current position of the cursor in the result set. Negative offset values move backward in the result set. An offset value of zero does not move the current location, but allows you to repopulate the row buffer.

See "UltraLite_Table_iface class" on page 224.

Example

The following code opens the table named MyTable and displays the value of the column named MyColumn for each row.

```
Table * tbl = conn->openTable( "MyTable" );
ul_column_num colID =
   tbl->GetSchema()->GetColumnID( "MyColumn" );

while ( tbl->Next() ){
   char buffer[ MAX_NAME_LEN ];
```

```
        ULValue colValue = tbl->Get(colID);
        colValue.GetString(buffer, MAX_NAME_LEN);
        printf( "%s\n", buffer );
    }
```

You expose the rows of the table to the application when you open the table object. By default, the rows are ordered by primary key value, but you can specify an index when opening a table to access the rows in a particular order.

Example

The following code fragment moves to the first row of the MyTable table as ordered by the ix_col index.

```
ULValue table_name( UL_TEXT("MyTable") )
ULValue index_name( UL_TEXT("ix_col") )
Table * tbl =
   conn->OpenTableWithIndex( table_name, index_name );
```

See "UltraLite_Table_iface class" on page 224.

UltraLite modes

The UltraLite mode determines how values in the buffer are used. You can set the UltraLite mode to one of the following:

- **Insert mode** Data in the buffer is added to the table as a new row when the insert method is called.
- **Update mode** Data in the buffer replaces the current row when the update method is called.
- **Find mode** Locates a row whose value exactly matches the data in the buffer when one of the find methods is called.
- **Lookup mode** Locates a row whose value matches or is greater than the data in the buffer when one of the lookup methods is called.

The mode is set by calling the corresponding method to set the mode. For example, InsertBegin, BeginUpdate, FindBegin, and so on.

Accessing the current row

A Table object is always located at one of the following positions:

- Before the first row of the table.
- On a row of the table.
- After the last row of the table.

If the Table object is positioned on a row, you can use one of a set of methods appropriate for the data type to retrieve or modify the value of the columns in that row.

Retrieving column values

The Table object provides a set of methods for retrieving column values. These methods take the column ID as the argument.

The following code fragment retrieves the value of the lname column, which is a character string.

```
ULValue val;
char lname[ MAX_NAME_LEN ];
val = tbl->Get( UL_TEXT("lname") );
val.GetString( lname, MAX_NAME_LEN );
```

The following code retrieves the value of the cust_id column, which is an integer.

```
int id = tbl->Get( UL_TEXT("cust_id") );
```

Modifying column values

In addition to the methods for retrieving values, there are methods for setting values. These methods take the column ID and the value as arguments.

For example, the following code sets the value of the lname column to Kaminski.

```
ULValue lname_col( UL_TEXT("fname") );
ULValue v_lname( UL_TEXT("Kaminski") );
tbl->Set( lname_col, v_lname );
```

By setting column values, you do not directly alter the data in the database. You can assign values to the properties, even if you are before the first row or after the last row of the table. Do not attempt to access data when the current row is undefined. For example, attempting to fetch the column value in the following example is incorrect:

```
// This code is incorrect
tbl.BeforeFirst();
id = tbl.Get( cust_id );
```

Casting values

The method you choose should match the data type you want to assign. UltraLite automatically casts database data types where they are compatible, so that you can use the GetString method to fetch an integer value into a string variable, and so on. See "Converting data types explicitly" [*UltraLite - Database Management and Reference*].

Searching rows

UltraLite has different modes of operation for working with data. You can use two of these modes, find and lookup, for searching. The Table object has methods corresponding to these modes for locating particular rows in a table.

Note
The columns you search with Find and Lookup methods must be in the index that is used to open the table.

- **Find methods** Move to the first row that exactly matches specified search values, under the sort order specified when the Table object was opened. If the search values cannot be found, the application is positioned before the first or after the last row.
- **Lookup methods** Move to the first row that matches or is greater than a specified search value, under the sort order specified when the Table object was opened.

To search for a row

1. Enter find or lookup mode.

   Call a method on the table object to set the mode. For example, the following code enters find mode.

   ```
   tbl.FindBegin();
   ```

2. Set the search values.

   Set the search values in the current row. Setting these values only affects the buffer holding the current row, not the database. For example, the following code fragment sets the value in the buffer to Kaminski.

   ```
   ULValue lname_col = t->GetSchema()->GetColumnID( UL_TEXT("lname") );
   ULValue v_lname( UL_TEXT("Kaminski") );
   tbl.Set( lname_col, v_lname );
   ```

3. Search for the row.

   Call the appropriate method to carry out the search. For example, the following code looks for the first row that exactly matches the specified value in the current index.

   For multi-column indexes, a value for the first column is always used, but you can omit the other columns.

   ```
   tCustomer.FindFirst();
   ```

4. Search for the next instance of the row.

   Call the appropriate method to carry out the search. For a find operation, FindNext locates the next instance of the parameters in the index. For a lookup, MoveNext locates the next instance.

See “UltraLite_Table_iface class” on page 224.

Updating rows

The following procedure updates a row.

To update a row

1. Move to the row you want to update.

   You can move to a row by scrolling through the table or by searching the table using find and lookup methods.

2. Enter update mode.

   For example, the following instruction enters update mode on table tbl.

```
tbl.BeginUpdate();
```

3. Set the new values for the row to be updated. For example, the following instruction sets the id column in the buffer to 3.

   ```
   tbl.Set( UL_TEXT("id"), 3 );
   ```

4. Execute the Update.

   ```
   tbl.Update();
   ```

After the update operation, the current row is the row that has been updated.

The UltraLite C++ API does not commit changes to the database until you commit them using `conn->Commit()`. See "Managing transactions" on page 23.

Caution
Do not update the primary key of a row: delete the row and add a new row instead.

Inserting rows

The steps to insert a row are very similar to those for updating rows, except that there is no need to locate a row in the table before carrying out the insert operation. The order of row insertion into the table has no significance since data is always inserted in the database according to the index.

Example

The following code fragment inserts a new row.

```
tbl.InsertBegin();
tbl.Set( UL_TEXT("id"), 3 );
tbl.Set( UL_TEXT("lname"), "Carlo" );
tbl.Insert();
tbl.Commit();
```

If you do not set a value for one of the columns, and that column has a default, the default value is used. If the column has no default, one of the following entries is used:

- For nullable columns, NULL.
- For numeric columns that disallow NULL, zero.
- For character columns that disallow NULL, an empty string.
- To explicitly set a value to NULL, use the SetNull method.

Deleting rows

The steps to delete a row are simpler than inserting or updating rows.

The following procedure deletes a row.

To delete a row

1. Move to the row you want to delete.
2. Execute the Table.Delete method.

   ```
   tbl.Delete();
   ```

Managing transactions

The UltraLite C++ API does not support AutoCommit mode. Transactions must be explicitly committed or rolled back.

To commit a transaction

- Execute a Conn->Commit statement.

  See "Commit function" on page 159.

To roll back a transaction

- Execute a Conn->Rollback statement.

  See "Rollback function" on page 172.

For more information about transaction management in UltraLite, see "UltraLite transaction processing" [*UltraLite - Database Management and Reference*].

Accessing schema information

The objects in the API represent tables, columns, indexes, and synchronization publications. Each object has a GetSchema method that provides access to information about the structure of that object.

You cannot modify the schema through the API. You can only retrieve information about the schema.

You can access the following schema objects and information:

- **DatabaseSchema** Exposes the number and names of the tables in the database, and the global properties such as the format of dates and times.

  To obtain a DatabaseSchema object, use Conn->GetSchema.

  See "GetSchema function" on page 166.

- **TableSchema** Exposes the number and names of the columns and indexes for this table.

  To obtain a TableSchema object, use tbl->GetSchema.

  See "GetSchema function" on page 166.

- **IndexSchema** Returns information about the column in the index. As an index has no data directly associated with it there is no separate Index class, just the IndexSchema class.

  To obtain a IndexSchema object, call the table_schema->GetIndexSchema or table_schema->GetPrimaryKey methods.

  See "UltraLite_Table_iface class" on page 224.

Handling errors

You should check for errors after each database operation by using methods of the ULSqlca object. For example, LastCodeOK checks if the operation was successful, while GetSQLCode returns the numerical value of the SQLCode. For more information about the meaning of these values, see "SQL Anywhere error messages sorted by Sybase error code" [*Error Messages*].

In addition to explicit error handling, UltraLite supports an error callback function. If you register a callback function, UltraLite calls the function whenever an UltraLite error occurs. The callback function does not control application flow, but does enable you to be notified of all errors. Use of a callback function is particularly helpful during application development and debugging. For more information about using the callback function, see "Tutorial: Build an application using the C++ API" on page 319.

For a sample callback function, see "Callback function for ULRegisterErrorCallback" on page 100 and "ULRegisterErrorCallback function" on page 120.

For a list of error codes thrown by the UltraLite C++ API, see "SQL Anywhere error messages sorted by Sybase error code" [*Error Messages*].

Authenticating users

UltraLite databases can define up to four user IDs. UltraLite databases are created with a default user ID and password of **DBA** and **sql**, respectively. All connections to an UltraLite database must supply a user ID and password. Password changes and user ID additions and deletions can be performed once a connection is established.

You cannot directly change a user ID. You can add a user ID and delete an existing user ID.

To add a user or change a password for an existing user

1. Connect to the database as an existing user.
2. Grant the user connection authority with the desired password using the conn->GrantConnectTo method.

   This procedure is the same whether you are adding a new user or changing the password of an existing user.

   See "GrantConnectTo function" on page 167.

To delete an existing user

1. Connect to the database as an existing user.
2. Revoke the user's connection authority using the conn->RevokeConnectFrom method.

   See "RevokeConnectFrom function" on page 171.

Encrypting data

You can choose to either encrypt or obfuscate an UltraLite database using the UltraLite C++ API. Encryption provides very secure representation of the data in the database whereas obfuscation provides a simplistic level of security that is intended to prevent casual observation of the contents of the database.

For background information, see "Choosing database creation parameters for UltraLite" [*UltraLite - Database Management and Reference*],

Encryption

To create a database with encryption, specify an encryption key by specifying the **key=** connection parameter in the connection string. When you call the CreateDatabase method, the database is created and encrypted with the specified key.

After the database is encrypted, all connections to the database must specify the correct encryption key. Otherwise, the connection fails.

See "UltraLite DBKEY connection parameter" [*UltraLite - Database Management and Reference*].

Obfuscation

To obfuscate the database, specify `obfuscate=1` as a database creation parameter.

See "Securing UltraLite databases" [*UltraLite - Database Management and Reference*].

Synchronizing data

UltraLite applications can synchronize data with a central database. Synchronization requires the MobiLink synchronization software included with SQL Anywhere.

The UltraLite C++ API supports TCP/IP, TLS, HTTP, and HTTPS synchronization. Synchronization is initiated by the UltraLite application. In all cases, you use methods and properties of the connection object to control synchronization.

For more information about synchronization, see “UltraLite clients” [*UltraLite - Database Management and Reference*].

For more information about the ul_synch_info structure used for synchronization, see “ul_synch_info_a struct” on page 133 or “ul_synch_info_w2 struct” on page 136 depending whether you are using ASCII or wide characters.

For more information about synchronization parameters, see “Synchronization parameters for UltraLite” [*UltraLite - Database Management and Reference*].

Compiling and linking your application

A set of runtime libraries is available for some platforms when using the UltraLite C++ API. These include, for Windows Mobile and Windows, a database engine that permits multi-process access to the same database.

The runtime libraries are provided in the *install-dir\UltraLite\Palm*, *install-dir\UltraLite\ce*,*install-dir \UltraLite\win32* and *install-dirx64* directories.

Runtime libraries for Palm OS

The following libraries are supplied for applications on the Palm OS. The libraries are located in *install-dir \UltraLite\Palm\68k\lib\cw*.

- **ulrt.lib** A static library.
- **ulbase.lib** A library containing extra functions that can not be provided in a separate dynamic link library (DLL). C/C++ applications should link against this library to ensure access to UltraLite features.

Runtime libraries for Windows Mobile

The Windows Mobile libraries are in the *install-dir\UltraLite\ce\arm.50\Lib* directory.

Dynamic libraries are provided for Windows Mobile:

- **ulbase.lib** A library containing extra functions that can not be provided in a separate dynamic link library (DLL). C/C++ applications should link against this library to ensure access to UltraLite features.
- **ulrt.lib** When linking against this library, be sure to specify the following compilation option:

  ```
  /DUNICODE
  ```

- **ulrtc.lib** A Unicode character set static library for use with the UltraLite engine for multi-process access to an UltraLite database.

  When linking against this library, be sure to specify the following compilation option:

  ```
  /DUNICODE
  ```

Runtime libraries for Windows desktops

The *install-dir\UltraLite\win32\386\Lib\vs8* and *install-dir\UltraLite\x64\Lib\vs8* directories contain libraries for supported Windows desktop operating systems. The following libraries are included:

- **ulbase.lib** A library containing functions that can not be provided in a separate dynamic link library (DLL). C/C++ applications should link against this library to ensure access to UltraLite features.
- **ulrt11.dll** An ANSI character set dynamic link library. To use this library, link your application against the import library, *ulimp.lib*.

  When linking against this library, be sure to specify the following compilation option:

  ```
  /DUL_USE_DLL
  ```

Developing applications using Embedded SQL

Contents

This section describes how to write database access code for embedded SQL UltraLite applications.

For an overview of the UltraLite C/C++ development process, see "UltraLite for C/C++ developers" on page 1.

For embedded SQL reference information, see "Embedded SQL API reference" on page 259.

For more information about the SQL preprocessor, see "SQL Preprocessor for UltraLite utility (sqlpp)" [*UltraLite - Database Management and Reference*].

Example of embedded SQL

Embedded SQL is an environment that is a combination of C/C++ program code and pseudo-code. The pseudo-code that can be interspersed with traditional C/C++ code is a subset of SQL statements. A preprocessor converts the embedded SQL statements into function calls that are part of the actual code that is compiled to create the application.

Following is a very simple example of an embedded SQL program. It illustrates updating an UltraLite database record by changing the surname of employee 195.

```
#include <stdio.h>
EXEC SQL INCLUDE SQLCA;
main( )
{
   db_init( &sqlca );
   EXEC SQL WHENEVER SQLERROR GOTO error;
   EXEC SQL CONNECT "DBA" IDENTIFIED BY "sql";
   EXEC SQL UPDATE employee
      SET emp_lname = 'Johnson'
      WHERE emp_id = 195;
   EXEC SQL COMMIT;
   EXEC SQL DISCONNECT;
   db_fini( &sqlca );
   return( 0 );
   error:
      printf( "update unsuccessful: sqlcode = %ld\n",
         sqlca.sqlcode );
      return( -1 );
}
```

Although this example is too simplistic to be useful, it illustrates the following aspects common to all embedded SQL applications:

- Each SQL statement is prefixed with the keywords EXEC SQL.
- Each SQL statement ends with a semicolon.
- Some embedded SQL statements are not part of standard SQL. The INCLUDE SQLCA statement is one example.
- In addition to SQL statements, embedded SQL also provides library functions to perform some specific tasks. The functions db_init and db_fini are two examples of library function calls.

Initialization

The above sample code illustrates initialization statements that must be included before working with the data in an UltraLite database:

1. Define the SQL Communications Area (SQLCA), using the following command:

   ```
   EXEC SQL INCLUDE SQLCA;
   ```

   This definition must be the first embedded SQL statement, so a natural place for it is the end of the include list.

   If you have multiple *.sqc* files in your application, each file must have this line.

2. The first database action must be a call to an embedded SQL library function named db_init. This function initializes the UltraLite runtime library. Only embedded SQL definition statements can be executed before this call.

   See "db_init function" on page 262.

3. You must use the SQL CONNECT statement to connect to the UltraLite database.

Preparing to exit

The above sample code demonstrates the sequence of calls required when preparing to exit:

1. Commit or rollback any outstanding changes.
2. Disconnect from the database.
3. End your SQL work with a call to a library function named db_fini.

When you exit, any uncommitted database changes are automatically rolled back.

Error handling

There is virtually no interaction between the SQL and C code in this example. The C code only controls the flow of the program. The WHENEVER statement is used for error checking. The error action, GOTO in this example, is executed whenever any SQL statement causes an error.

Structure of embedded SQL programs

All embedded SQL statements start with the words EXEC SQL and end with a semicolon. Normal C-language comments are allowed in the middle of embedded SQL statements.

Every C program using embedded SQL must contain the following statement before any other embedded SQL statements in the source file.

```
EXEC SQL INCLUDE SQLCA;
```

The first embedded SQL executable statement in the program must be a SQL CONNECT statement. The CONNECT statement supplies connection parameters that are used to establish a connection to the UltraLite database.

Some embedded SQL commands do not generate any executable C code, or do not involve communication with the database. Only these commands are allowed before the CONNECT statement. Most notable are the INCLUDE statement and the WHENEVER statement for specifying error processing.

Initializing the SQL Communications Area

The SQL Communications Area (SQLCA) is an area of memory that is used for communicating statistics and errors from the application to the database and back to the application. The SQLCA is used as a handle for the application-to-database communication link. It is passed explicitly to all database library functions that communicate with the database. It is implicitly passed in all embedded SQL statements.

UltraLite defines a SQLCA global variable for you in the generated code. The preprocessor generates an external reference for the global SQLCA variable. The external reference is named sqlca and is of type SQLCA. The actual global variable is declared in the imports library.

The SQLCA type is defined in the header file *install-dir\SDK\Include\sqlca.h*.

After declaring the SQLCA (`EXEC SQL INCLUDE SQLCA;`), but before your application can carry out any operations on a database, you must initialize the communications area by calling db_init and passing it the SQLCA:

```
db_init( &sqlca );
```

SQLCA provides error codes

You reference the SQLCA to test for a particular error code. The sqlcode field contains an error code when a database request causes an error. Macros are defined for referencing the sqlcode field and some other fields in the sqlca.

SQLCA fields

The SQLCA contains the following fields:

- **sqlcaid** An 8-byte character field that contains the string SQLCA as an identification of the SQLCA structure. This field helps in debugging when you are looking at memory contents.
- **sqlcabc** A long integer that contains the length in bytes of the SQLCA structure.
- **sqlcode** A long integer that contains an error code when the database detects an error on a request. Definitions for the error codes are in the header file *install-dir\SDK\Include\sqlerr.h*. The error code is 0 (zero) for a successful operation, a positive value for a warning, and a negative value for an error.

  You can access this field directly using the SQLCODE macro.

  For a list of error codes, see "SQL Anywhere error messages" [*Error Messages*].
- **sqlerrml** The length of the information in the sqlerrmc field.

  UltraLite applications do not use this field.
- **sqlerrmc** May contain one or more character strings to be inserted into an error message. Some error messages contain a placeholder string (*%1*) which is replaced with the text in this field.

  UltraLite applications do not use this field.
- **sqlerrp** Reserved.

- **sqlerrd**   A utility array of long integers.
- **sqlwarn**   Reserved.

  UltraLite applications do not use this field.
- **sqlstate**   The SQLSTATE status value.

  UltraLite applications do not use this field.

Connecting to a database

To connect to an UltraLite database from an embedded SQL application, include the EXEC SQL CONNECT statement in your code after initializing the SQLCA.

The CONNECT statement has the following form:

EXEC SQL CONNECT USING
'uid=*user-name***;pwd=***password***;dbf=***database-filename***'**;

The connection string (enclosed in single quotes) may include additional database connection parameters.

For more information about database connection parameters, see "UltraLite connection parameters" [*UltraLite - Database Management and Reference*].

For more information about the CONNECT statement, see "CONNECT statement [ESQL] [Interactive SQL]" [*SQL Anywhere Server - SQL Reference*].

Managing multiple connections

If you want more than one database connection in your application, you can either use multiple SQLCAs or you can use a single SQLCA to manage the connections.

To use multiple SQLCAs

Managing multiple SQLCAs

1. Each SQLCA used in your program must be initialized with a call to **db_init** and cleaned up at the end with a call to **db_fini**.

   See "db_init function" on page 262.

2. The embedded SQL statement SET SQLCA is used to tell the SQL preprocessor to use a specific SQLCA for database requests. Usually, a statement such as the following is used at the top of your program or in a header file to set the SQLCA reference to point at task specific data:

   ```
   EXEC SQL SET SQLCA 'task_data->sqlca';
   ```

   This statement does not generate any code and does not affect performance. It changes the state within the preprocessor so that any reference to the SQLCA will use the given string.

   For more information about creating SQLCAs, see "SET SQLCA statement [ESQL]" [*SQL Anywhere Server - SQL Reference*].

To use a single SQLCA

As an alternative to using multiple SQLCAs, you can use a single SQLCA to manage more than one connection to a database.

Each SQLCA has a single active or current connection, but that connection can be changed. Before executing a command, use the SET CONNECTION statement to specify the connection on which the command should be executed.

See “SET CONNECTION statement [Interactive SQL] [ESQL]” [*SQL Anywhere Server - SQL Reference*].

Using host variables

Embedded SQL applications use host variables to communicate values to and from the database. Host variables are C variables that are identified to the SQL preprocessor in a declaration section.

Declaring host variables

Define host variables by placing them within a declaration section. Host variables are declared by surrounding the normal C variable declarations with BEGIN DECLARE SECTION and END DECLARE SECTION statements.

Whenever you use a host variable in a SQL statement, you must prefix the variable name with a colon (:) so the SQL preprocessor knows you are referring to a (declared) host variable and distinguish it from other identifiers allowed in the statement.

You can use host variables in place of value constants in any SQL statement. When the database server executes the command, the value of the host variable is read from or written to each host variable. Host variables cannot be used in place of table or column names.

The SQL preprocessor does not scan C language code except inside a declaration section. Initializers for variables are allowed inside a declaration section, while **typedef** types and structures are not permitted.

The following sample code illustrates the use of host variables with an INSERT command. The variables are filled in by the program and then inserted into the database:

```
/* Declare fields for personal data. */
EXEC SQL BEGIN DECLARE SECTION;
   long employee_number = 0;
   char employee_name[50];
   char employee_initials[8];
   char employee_phone[15];
EXEC SQL END DECLARE SECTION;
/* Fill variables with appropriate values. */
/* Insert a row in the database. */
EXEC SQL INSERT INTO Employee
   VALUES (:employee_number, :employee_name,
      :employee_initials, :employee_phone );
```

Data types in embedded SQL

To transfer information between a program and the database server, every data item must have a data type. You can create a host variable with any one of the supported types.

Only a limited number of C data types are supported as host variables. Also, certain host variable types do not have a corresponding C type.

Macros defined in the *sqlca.h* header file can be used to declare a host variable of type VARCHAR, FIXCHAR, BINARY, DECIMAL, or SQLDATETIME. These macros are used as follows:

```
EXEC SQL BEGIN DECLARE SECTION;
   DECL_VARCHAR( 10 ) v_varchar;
   DECL_FIXCHAR( 10 ) v_fixchar;
   DECL_BINARY( 4000 ) v_binary;
   DECL_DECIMAL( 10, 2 ) v_packed_decimal;
   DECL_DATETIME v_datetime;
EXEC SQL END DECLARE SECTION;
```

The preprocessor recognizes these macros within a declaration section and treats the variable as the appropriate type.

The following data types are supported by the embedded SQL programming interface:

- **16-bit signed integer**

  ```
  short int I;
  unsigned short int I;
  ```

- **32-bit signed integer**

  ```
  long int l;
  unsigned long int l;
  ```

- **4-byte floating point number**

  ```
  float f;
  ```

- **8-byte floating point number**

  ```
  double d;
  ```

- **Packed decimal number**

  ```
  DECL_DECIMAL(p,s)
  typedef struct TYPE_DECIMAL {
     char array[1];
  } TYPE_DECIMAL;
  ```

- **Null terminated, blank-padded character string**

  ```
  char a[n]; /* n > 1 */
  char *a; /* n = 2049 */
  ```

  Because the C-language array must also hold the NULL terminator, a char a[n] data type maps to a CHAR(n - 1) SQL data type, which can hold –1 characters.

  Pointers to char, WCHAR, and TCHAR
  The SQL preprocessor assumes that a **pointer to char** points to a character array of size 2049 bytes and that this array can safely hold 2048 characters, plus the NULL terminator. In other words, a char* data type maps to a CHAR(2048) SQL type. If that is not the case, your application may corrupt memory.

  If you are using a 16-bit compiler, requiring 2049 bytes can make the program stack overflow. Instead, use a declared array, even as a parameter to a function, to let the SQL preprocessor know the size of the array. WCHAR and TCHAR behave similarly to char.

- **NULL terminated UNICODE or wide character string** Each character occupies two bytes of space and so may contain UNICODE characters.

  ```
  WCHAR a[n]; /* n > 1 */
  ```

- **NULL terminated system-dependent character string** A TCHAR is equivalent to a WCHAR for systems that use UNICODE (for example, Windows Mobile) for their character set; otherwise, a TCHAR is equivalent to a char. The TCHAR data type is designed to support character strings in either kind of system automatically.

```
TCHAR a[n]; /* n > 1 */
```

- **Fixed-length blank padded character string**

```
char a; /* n  = 1 */
DECL_FIXCHAR(n) a; /* n >= 1 */
```

- **Variable-length character string with a two-byte length field** When supplying information to the database server, you must set the length field. When fetching information from the database server, the server sets the length field (not padded).

```
DECL_VARCHAR(n) a; /* n >= 1 */
typedef struct VARCHAR {
   unsigned short int len;
   TCHAR array[1];
} VARCHAR;
```

- **Variable-length binary data with a two-byte length field** When supplying information to the database server, you must set the length field. When fetching information from the database server, the server sets the length field.

```
DECL_BINARY(n) a; /* n >= 1 */
typedef struct BINARY {
   unsigned short int len;
   unsigned char array[1];
} BINARY;
```

- **SQLDATETIME structure with fields for each part of a timestamp**

```
DECL_DATETIME a;
typedef struct SQLDATETIME {
   unsigned short year; /* for example: 1999 */
   unsigned char month; /* 0-11 */
   unsigned char day_of_week; /* 0-6, 0 = Sunday */
   unsigned short day_of_year; /* 0-365 */
   unsigned char day; /* 1-31 */
   unsigned char hour; /* 0-23 */
   unsigned char minute; /* 0-59 */
   unsigned char second; /* 0-59 */
   unsigned long microsecond; /* 0-999999 */
} SQLDATETIME;
```

The SQLDATETIME structure is used to retrieve fields of the DATE, TIME, and TIMESTAMP type (or anything that can be converted to one of these). Often, applications have their own formats and date manipulation code. Fetching data in this structure makes it easier for the programmer to manipulate this data. Note that DATE, TIME, and TIMESTAMP fields can also be fetched and updated with any character type.

If you use a SQLDATETIME structure to enter a date, time, or timestamp into the database, the day_of_year and day_of_week members are ignored.

For more information about the date_format, time_format, timestamp_format, and date_order database options, see "Database options" [*SQL Anywhere Server - Database Administration*].

- **DT_LONGVARCHAR** Long varying length character data. The macro defines a structure, as follows:

```
#define DECL_LONGVARCHAR( size ) \
  struct { a_sql_uint32    array_len;    \
               a_sql_uint32    stored_len;   \
               a_sql_uint32    untrunc_len;  \
               char            array[size+1];\
             }
```

The DECL_LONGVARCHAR struct may be used with more than 32KB of data. Data may be fetched all at once, or in pieces using the GET DATA statement. Data may be supplied to the server all at once, or in pieces by appending to a database variable using the SET statement. The data is not null terminated.

```
typedef struct BINARY {
  unsigned short int len;
  char array[1];
} BINARY;
```

- **DT_LONGBINARY** Long binary data. The macro defines a structure, as follows:

```
#define DECL_LONGBINARY( size )  \
  struct { a_sql_uint32    array_len;    \
           a_sql_uint32    stored_len;   \
           a_sql_uint32    untrunc_len;  \
           char            array[size];  \
         }
```

The DECL_LONGBINARY struct may be used with more than 32KB of data. Data may be fetched all at once, or in pieces using the GET DATA statement. Data may be supplied to the server all at once, or in pieces by appending to a database variable using the SET statement.

The structures are defined in the *install-dir\SDK\Include\sqlca.h* file. The VARCHAR, BINARY, and TYPE_DECIMAL types contain a one-character array and are not useful for declaring host variables. However, they are useful for allocating variables dynamically or typecasting other variables.

DATE and TIME database types

There are no corresponding embedded SQL interface data types for the various DATE and TIME database types. These database types are fetched and updated either using the SQLDATETIME structure or using character strings.

There are no embedded SQL interface data types for LONG VARCHAR and LONG BINARY database types.

Host variable usage

Host variables can be used in the following circumstances:

- In a SELECT, INSERT, UPDATE, or DELETE statement in any place where a number or string constant is allowed.
- In the INTO clause of a SELECT or FETCH statement.
- In CONNECT, DISCONNECT, and SET CONNECT statements, a host variable can be used in place of a user ID, password, connection name, or database name.

Host variables can *never* be used in place of a table name or a column name.

The scope of host variables

A host-variable declaration section can appear anywhere that C variables can normally be declared, including the parameter declaration section of a C function. The C variables have their normal scope (available within the block in which they are defined). However, since the SQL preprocessor does not scan C code, it does not respect C blocks.

Preprocessor assumes all host variables are global

As far as the SQL preprocessor is concerned, host variables are globally known in the source module following their declaration. Two host variables cannot have the same name. The only exception to this rule is that two host variables can have the same name if they have identical types (including any necessary lengths).

The best practice is to give each host variable a unique name.

Examples

Because the SQL preprocessor can not parse C code, it assumes all host variables, no matter where they are declared, are known globally following their declaration.

```
// Example demonstrating poor coding
EXEC SQL BEGIN DECLARE SECTION;
   long emp_id;
EXEC SQL END DECLARE SECTION;
long getManagerID( void )
{
   EXEC SQL BEGIN DECLARE SECTION;
      long manager_id = 0;
   EXEC SQL END DECLARE SECTION;
   EXEC SQL SELECT manager_id
            INTO :manager_id
            FROM employee
            WHERE emp_number = :emp_id;
   return( manager_number );
}
void setManagerID( long manager_id )
{
   EXEC SQL UPDATE employee
            SET manager_number = :manager_id
            WHERE emp_number = :emp_id;
}
```

Although the above code works, it is confusing because the SQL preprocessor relies on the declaration inside *getManagerID* when processing the statement within *setManagerID*. You should rewrite this code as follows:

```
// Rewritten example
#if 0
   // Declarations for the SQL preprocessor
   EXEC SQL BEGIN DECLARE SECTION;
      long emp_id;
      long manager_id;
   EXEC SQL END DECLARE SECTION;
#endif
long getManagerID( long emp_id )
{
   long manager_id = 0;
   EXEC SQL SELECT manager_id
```

```
            INTO :manager_id
            FROM employee
            WHERE emp_number = :emp_id;
   return( manager_number );
}
void setManagerID( long emp_id, long manager_id )
{
   EXEC SQL UPDATE employee
            SET manager_number = :manager_id
            WHERE emp_number = :emp_id;
}
```

The SQL preprocessor sees the declaration of the host variables contained within the #if directive because it ignores these directives. On the other hand, it ignores the declarations within the procedures because they are not inside a DECLARE SECTION. Conversely, the C compiler ignores the declarations within the #if directive and uses those within the procedures.

These declarations work only because variables having the same name are declared to have exactly the same type.

Using expressions as host variables

Host variables must be simple names because the SQL preprocessor does not recognize pointer or reference expressions. For example, the following statement *does not work* because the SQL preprocessor does not understand the dot operator. The same syntax has a different meaning in SQL.

```
// Incorrect statement:
EXEC SQL SELECT LAST sales_id INTO :mystruct.mymember;
```

Although the above syntax is not allowed, you can still use an expression with the following technique:

- Wrap the SQL declaration section in an #if 0 preprocessor directive. The SQL preprocessor will read the declarations and use them for the rest of the module because it ignores preprocessor directives.
- Define a macro with the same name as the host variable. Since the SQL declaration section is not seen by the C compiler because of the #if directive, no conflict will arise. Ensure that the macro evaluates to the same type host variable.

The following code demonstrates this technique to hide the *host_value* expression from the SQL preprocessor.

```
#include <sqlerr.h>
#include <stdio.h>
EXEC SQL INCLUDE SQLCA;
typedef struct my_struct {
   long     host_field;
} my_struct;
#if 0
   // Because it ignores #if preprocessing directives,
   // SQLPP reads the following declaration.
   EXEC SQL BEGIN DECLARE SECTION;
      long    host_value;
   EXEC SQL END DECLARE SECTION;
#endif
// Make C/C++ recognize the 'host_value' identifier
// as a macro that expands to a struct field.
#define host_value my_s.host_field
```

Since the SQLPP processor ignores directives for conditional compilation, *host_value* is treated as a *long* host variable and will emit that name when it is subsequently used as a host variable. The C/C++ compiler processes the emitted file and will substitute *my_s.host_field* for all such uses of that name.

With the above declarations in place, you can proceed to access *host_field* as follows.

```
void main( void )
{
   my_struct      my_s;
   db_init( &sqlca );
   EXEC SQL CONNECT "DBA" IDENTIFIED BY "SQL";
   EXEC SQL DECLARE my_table_cursor CURSOR FOR
      SELECT int_col FROM my_table order by int_col;
   EXEC SQL OPEN my_table_cursor;
   for( ; ; ) {
      // :host_value references my_s.host_field
      EXEC SQL FETCH NEXT AllRows INTO :host_value;
      if( SQLCODE == SQLE_NOTFOUND ) {
         break;
      }
      printf( "%ld\n", my_s.host_field );
   }
   EXEC SQL CLOSE my_table_cursor;
   EXEC SQL DISCONNECT;
   db_fini( &sqlca );
}
```

You can use the same technique to use other lvalues as host variables:

- pointer indirections

  ```
  *ptr
  p_struct->ptr
  (*pp_struct)->ptr
  ```

- array references

  ```
  my_array[ I ]
  ```

- arbitrarily complex lvalues

Using host variables in C++

A similar situation arises when using host variables within C++ classes. It is frequently convenient to declare your class in a separate header file. This header file might contain, for example, the following declaration of *my_class*.

```
typedef short a_bool;
#define  TRUE  ((a_bool)(1==1))
#define  FALSE ((a_bool)(0==1))
public class {
   long  host_member;
   my_class();    // Constructor
   ~my_class();      // Destructor
   a_bool FetchNextRow( void );
      // Fetch the next row into host_member
} my_class;
```

In this example, each method is implemented in an embedded SQL source file. Only simple variables can be used as host variables. The technique introduced in the preceding section can be used to access a data member of a class.

```
EXEC SQL INCLUDE SQLCA;
#include "my_class.hpp"
#if 0
   // Because it ignores #if preprocessing directives,
   // SQLPP reads the following declaration.
   EXEC SQL BEGIN DECLARE SECTION;
      long  this_host_member;
   EXEC SQL END DECLARE SECTION;
#endif
// Macro used by the C++ compiler only.
#define this_host_member this->host_member
my_class::my_class()
{
   EXEC SQL DECLARE my_table_cursor CURSOR FOR
      SELECT int_col FROM my_table order by int_col;
   EXEC SQL OPEN my_table_cursor;
}
my_class::~my_class()
{
   EXEC SQL CLOSE my_table_cursor;
}
a_bool my_class::FetchNextRow( void )
{
   // :this_host_member references this->host_member
   EXEC SQL FETCH NEXT AllRows INTO :this_host_member;
   return( SQLCODE != SQLE_NOTFOUND );
}
void main( void )
{
   db_init( &sqlca );
   EXEC SQL CONNECT "DBA" IDENTIFIED BY "SQL";
   {
      my_class mc; // Created after connecting.
      while( mc.FetchNextRow() ) {
         printf( "%ld\n", mc.host_member );
      }
   }
   EXEC SQL DISCONNECT;
   db_fini( &sqlca );
}
```

The above example declares `this_host_member` for the SQL preprocessor, but the macro causes C++ to convert it to `this->host_member`. The preprocessor would otherwise not know the type of this variable. Many C/C++ compilers do not tolerate duplicate declarations. The `#if` directive hides the second declaration from the compiler, but leaves it visible to the SQL preprocessor.

While multiple declarations can be useful, you must ensure that each declaration assigns the same variable name to the same type. The preprocessor assumes that each host variable is globally known following its declaration because it can not fully parse the C language.

Using indicator variables

An indicator variable is a C variable that holds supplementary information about a particular host variable. You can use a host variable when fetching or putting data. Use indicator variables to handle NULL values.

An indicator variable is a host variable of type short int. To detect or specify a NULL value, place the indicator variable immediately following a regular host variable in a SQL statement.

Example

For example, in the following INSERT statement, **:ind_phone** is an indicator variable.

```
EXEC SQL INSERT INTO Employee
   VALUES (:employee_number, :employee_name,
   :employee_initials, :employee_phone:ind_phone );
```

Indicator variable values

The following table provides a summary of indicator variable usage:

Indicator value	Supplying value to database	Receiving value from database
0	Host variable value	Fetched a non-NULL value.
-1	NULL value	Fetched a NULL value

Using indicator variables to handle NULL

Do not confuse the SQL concept of NULL with the C-language constant of the same name. In the SQL language, NULL represents either an unknown attribute or inapplicable information. The C-language constant represents a pointer value that does not point to a memory location.

When NULL is used in the SQL Anywhere documentation, it refers to the SQL database meaning given above. The C language constant is referred to as the null pointer (lowercase).

NULL is not the same as any value of the column's defined type. To pass NULL values to the database or receive NULL results back, you require something beyond regular host variables. Indicator variables serve this purpose.

Using indicator variables when inserting NULL

An INSERT statement can include an indicator variable as follows:

```
EXEC SQL BEGIN DECLARE SECTION;
short int employee_number;
char employee_name[50];
char employee_initials[6];
char employee_phone[15];
short int ind_phone;
EXEC SQL END DECLARE SECTION;
/* set values of empnum, empname,
   initials, and homephone */
if( /* phone number is known */ ) {
      ind_phone =  0;
   } else {
      ind_phone = -1; /* NULL */
   }
   EXEC SQL INSERT INTO Employee
      VALUES (:employee_number, :employee_name,
      :employee_initials, :employee_phone:ind_phone );
```

If the indicator variable has a value of -1, a NULL is written. If it has a value of 0, the actual value of employee_phone is written.

Using indicator variables when fetching NULL

Indicator variables are also used when receiving data from the database. They are used to indicate that a NULL value was fetched (indicator is negative). If a NULL value is fetched from the database and an indicator variable is not supplied, the SQLE_NO_INDICATOR error is generated.

For more information about errors and warnings returned in the SQLCA structure, see "Initializing the SQL Communications Area" on page 34.

Fetching data

Fetching data in embedded SQL is done using the SELECT statement. There are two cases:

1. The SELECT statement returns no rows or returns exactly one row.
2. The SELECT statement returns multiple rows.

Fetching one row

A single row query retrieves at most one row from the database. A single row query SELECT statement may have an INTO clause following the select list and before the FROM clause. The INTO clause contains a list of host variables to receive the value for each select list item. There must be the same number of host variables as there are select list items. The host variables may be accompanied by indicator variables to indicate NULL results.

When the SELECT statement is executed, the database server retrieves the results and places them in the host variables.

- If the query returns more than one row, the database server returns the SQLE_TOO_MANY_RECORDS error.
- If the query returns no rows, the SQLE_NOTFOUND warning is returned.

For more information about errors and warnings returned in the SQLCA structure, see "Initializing the SQL Communications Area" on page 34.

Example

For example, the following code fragment returns 1 if a row from the employee table is successfully fetched, 0 if the row doesn't exist, and -1 if an error occurs.

```
EXEC SQL BEGIN DECLARE SECTION;
   long int    emp_id;
   char        name[41];
   char        sex;
   char        birthdate[15];
   short int   ind_birthdate;
EXEC SQL END DECLARE SECTION;
int find_employee( long employee )
{
   emp_id = employee;
   EXEC SQL SELECT emp_fname || ' ' || emp_lname,
         sex, birth_date
      INTO :name, :sex, birthdate:ind_birthdate
      FROM "DBA".employee
      WHERE emp_id = :emp_id;
   if( SQLCODE == SQLE_NOTFOUND ) {
      return( 0 ); /* employee not found */
   } else if( SQLCODE < 0 ) {
      return( -1 ); /* error */
   } else {
      return( 1 ); /* found */
      }
   }
```

Fetching multiple rows

You use a cursor to retrieve rows from a query that has multiple rows in the result set. A cursor is a handle or an identifier for the SQL query result set and a position within that result set.

For an introduction to cursors, see “Working with cursors” [*SQL Anywhere Server - Programming*].

To manage a cursor in embedded SQL

1. Declare a cursor for a particular SELECT statement, using the DECLARE statement.
2. Open the cursor using the OPEN statement.
3. Retrieve rows from the cursor one at a time using the FETCH statement.
   - Fetch rows until the SQLE_NOTFOUND warning is returned. Error and warning codes are returned in the variable SQLCODE, defined in the SQL communications area structure.
4. Close the cursor, using the CLOSE statement.

Cursors in UltraLite applications are always opened using the WITH HOLD option. They are never closed automatically. You must explicitly close each cursor using the CLOSE statement.

The following is a simple example of cursor usage:

```
void print_employees( void )
{
   int status;
   EXEC SQL BEGIN DECLARE SECTION;
   char name[50];
   char sex;
   char birthdate[15];
   short int ind_birthdate;
   EXEC SQL END DECLARE SECTION;
   /* 1. Declare the cursor. */
   EXEC SQL DECLARE C1 CURSOR FOR
      SELECT emp_fname || ' ' || emp_lname,
                sex, birth_date
      FROM "DBA".employee
      ORDER BY emp_fname, emp_lname;
   /* 2. Open the cursor. */
   EXEC SQL OPEN C1;
   /* 3. Fetch each row from the cursor. */
   for( ;; ) {
      EXEC SQL FETCH C1 INTO :name, :sex,
            :birthdate:ind_birthdate;
      if( SQLCODE == SQLE_NOTFOUND ) {
         break; /* no more rows */
      } else if( SQLCODE < 0 ) {
         break; /* the FETCH caused an error */
      }
      if( ind_birthdate < 0 ) {
         strcpy( birthdate, "UNKNOWN" );
      }
      printf( "Name: %s Sex: %c Birthdate:
```

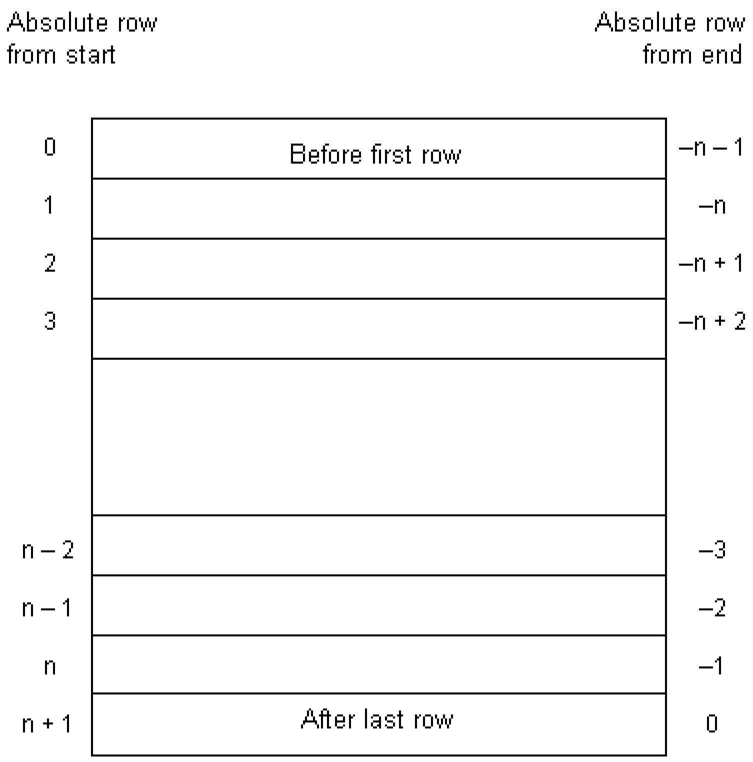
```
                    %s\n",name, sex, birthdate );
      }
      /* 4. Close the cursor. */
      EXEC SQL CLOSE C1;
   }
```

For more information about the FETCH statement, see "FETCH statement [ESQL] [SP]" [*SQL Anywhere Server - SQL Reference*].

Cursor positioning

A cursor is positioned in one of three places:

- On a row
- Before the first row
- After the last row



Order of rows in a cursor

You control the order of rows in a cursor by including an ORDER BY clause in the SELECT statements that defines that cursor. If you omit this clause, the order of the rows is unpredictable.

If you don't explicitly define an order, the only guarantee is that fetching repeatedly will return each row in the result set once and only once before SQLE_NOTFOUND is returned.

Repositioning a cursor

When you open a cursor, it is positioned before the first row. The FETCH statement automatically advances the cursor position. An attempt to FETCH beyond the last row results in a SQLE_NOTFOUND error, which can be used as a convenient signal to complete sequential processing of the rows.

You can also reposition the cursor to an absolute position relative to the start or end of the query results, or you can move the cursor relative to the current position. There are special *positioned* versions of the UPDATE and DELETE statements that can be used to update or delete the row at the current position of the cursor. If the cursor is positioned before the first row or after the last row, a SQLE_NOTFOUND error is returned.

To avoid unpredictable results when using explicit positioning, you can include an ORDER BY clause in the SELECT statement that defines the cursor.

You can use the PUT statement to insert a row into a cursor.

Cursor positioning after updates

After updating any information that is being accessed by an open cursor, it is best to fetch and display the rows again. If the cursor is being used to display a single row, FETCH RELATIVE 0 will re-fetch the current row. When the current row has been deleted, the next row will be fetched from the cursor (or SQLE_NOTFOUND is returned if there are no more rows).

When a temporary table is used for the cursor, inserted rows in the underlying tables do not appear at all until that cursor is closed and reopened. It is difficult for most programmers to detect whether a temporary table is involved in a SELECT statement without examining the code generated by the SQL preprocessor or by becoming knowledgeable about the conditions under which temporary tables are used. Temporary tables can usually be avoided by having an index on the columns used in the ORDER BY clause.

For more information about temporary tables, see “Use work tables in query processing (use All-rows optimization goal)” [*SQL Anywhere Server - SQL Usage*].

Inserts, updates, and deletes to non-temporary tables may affect the cursor positioning. Because UltraLite materializes cursor rows one at a time (when temporary tables are not used), the data from a freshly inserted row (or the absence of data from a freshly deleted row) may affect subsequent FETCH operations. In the simple case where (parts of) rows are being selected from a single table, an inserted or updated row will appear in the result set for the cursor when it satisfies the selection criteria of the SELECT statement. Similarly, a freshly deleted row that previously contributed to the result set will no longer be within it.

Authenticating users

UltraLite databases are created with a default user ID of DBA and default password of sql; you must first connect as this initial user. New users must be added from an existing connection.

A user ID cannot be changed; instead, you add the new user ID and then delete the existing user ID. A maximum of four user IDs are permitted for each UltraLite database.

On Palm OS, if you want to authenticate users whenever they return to an application from some other application, you must include the prompt for user and password information in your PilotMain routine.

User authentication example

A complete sample can be found in the *samples-dir\UltraLite\esqlauth* directory. The code below is taken from *samples-dir\UltraLite\esqlauth\sample.sqc*.

```
//embedded SQL
app() {
   ...
/* Declare fields */
   EXEC SQL BEGIN DECLARE SECTION;
      char uid[31];
      char pwd[31];
   EXEC SQL END DECLARE SECTION;
   db_init( &sqlca );
   ...
   EXEC SQL CONNECT "DBA" IDENTIFIED BY "sql";
   if( SQLCODE == SQLE_NOERROR ) {
      printf("Enter new user ID and password\n" );
      scanf( "%s %s", uid, pwd );
      ULGrantConnectTo( &sqlca,
         UL_TEXT( uid ), UL_TEXT( pwd ) );
      if( SQLCODE == SQLE_NOERROR ) {
         // new user added: remove DBA
         ULRevokeConnectFrom( &sqlca, UL_TEXT("DBA") );
      }
      EXEC SQL DISCONNECT;
   }
   // Prompt for password
    printf("Enter user ID and password\n" );
    scanf( "%s %s", uid, pwd );
    EXEC SQL CONNECT :uid IDENTIFIED BY :pwd;
```

The code carries out the following tasks:

1. Initiate database functionality by calling db_init.
2. Attempt to connect using the default user ID and password.
3. If the connection attempt is successful, add a new user.
4. If the new user is successfully added, delete the DBA user from the UltraLite database.
5. Disconnect. An updated user ID and password is now added to the database.
6. Connect using the updated user ID and password.

See:

- “ULGrantConnectTo function” on page 281
- “ULRevokeConnectFrom function” on page 286

Encrypting data

You can encrypt or obfuscate your UltraLite database using the UltraLite embedded SQL.

See "Encrypting data" on page 54.

Encryption

When an UltraLite database is created (using Sybase Central for example), an optional encryption key may be specified. The encryption key is used to encrypt the database. Once the database is encrypted, all subsequent connection attempts must supply the encryption key. The supplied key is checked against the original encryption key and the connection fails unless the key matches.

Choose an encryption key value that cannot be easily guessed. The key can be of arbitrary length, but generally a longer key is better, because a shorter key is easier to guess. Including a combination of numbers, letters, and special characters decreases the chances of someone guessing the key.

Do not include semicolons in your key. Do not put the key itself in quotes, or the quotes will be considered part of the key.

To connect to an encrypted UltraLite database

1. Specify the encryption key in the connection string used in the EXEC SQL CONNECT statement.

   The encryption key is specified with the key= connection string parameter.

   You must supply this key each time you want to connect to the database. Lost or forgotten keys result in completely inaccessible databases.

2. Handle attempts to open an encrypted database with the wrong key.

   If an attempt is made to open an encrypted database and the wrong key is supplied, db_init returns ul_false and SQLCODE -840 is set.

Changing the encryption key

You can change the encryption key for a database. The application must already be connected to the database using the existing key before the change can be made.

To change the encryption key on an UltraLite database

- Call the ULChangeEncryptionKey function, supplying the new key as an argument.

  The application must already be connected to the database using the old key before this function is called.

  See "ULChangeEncryptionKey function" on page 265.

Obfuscation

To obfuscate an UltraLite database

- An alternative to using database encryption is to specify that the database is to be obfuscated. Obfuscation is a simple masking of the data in the database that is intended to prevent browsing the data in the database with a low level file examination utility. Obfuscation is a database creation option and must be specified when the database is created.

  See "Choosing database creation parameters for UltraLite" [*UltraLite - Database Management and Reference*].

Adding synchronization to your application

Synchronization is a key feature of many UltraLite applications. This section describes how to add synchronization to your application.

The synchronization logic that keeps UltraLite applications up to date with the consolidated database is not held in the application itself. Synchronization scripts stored in the consolidated database, together with the MobiLink server and the UltraLite runtime library, control how changes are processed when they are uploaded and determines which changes are to be downloaded.

Overview

The specifics of each synchronization are controlled by a set of synchronization parameters. These parameters are gathered into a structure, which is then supplied as an argument in a function call to synchronize. The outline of the method is the same in each development model.

To add synchronization to your application

1. Initialize the structure that holds the synchronization parameters.

   See "Initializing the synchronization parameters" on page 56.

2. Assign the parameter values for your application.

   See "Network protocol options for UltraLite synchronization streams" [*UltraLite - Database Management and Reference*].

3. Call the synchronization function, supplying the structure or object as argument.

   See "Invoking synchronization" on page 57.

You must ensure that there are no uncommitted changes when you synchronize.

Synchronization parameters

The ul_synch_info structure is documented in the C/C++ component chapter; however, members of the structures are common to embedded SQL development as well. See "ul_synch_info_a struct" on page 133 or "ul_synch_info_w2 struct" on page 136 depending whether you are using ASCII or wide characters.

For a general description of synchronization parameters, see "Synchronization parameters for UltraLite" [*UltraLite - Database Management and Reference*].

Initializing the synchronization parameters

The synchronization parameters are stored in a structure.

The members of the structure undefined on initialization. You must set your parameters to their initial values with a call to a special function. The synchronization parameters are defined in a structure declared in the UltraLite header file *install-dir\SDK\Include\ulglobal.h*.

To initialize the synchronization parameters (embedded SQL)

- Call the ULInitSynchInfo function. For example:

  ```
  ul_synch_info synch_info;
  ULInitSynchInfo( &synch_info );
  ```

Setting synchronization parameters

The following code initiates TCP/IP synchronization. The MobiLink user name is `Betty Best`, with password `TwentyFour`, the script version is `default`, and the MobiLink server is running on the host computer `test.internal`, on port `2439`:

```
ul_synch_info synch_info;
ULInitSynchInfo( &synch_info );
synch_info.user_name = UL_TEXT("Betty Best");
synch_info.password = UL_TEXT("TwentyFour");
synch_info.version = UL_TEXT("default");
synch_info.stream = ULSocketStream();
synch_info.stream_parms =
   UL_TEXT("host=test.internal;port=2439");
ULSynchronize( &sqlca, &synch_info );
```

The following code for an application on the Palm Computing Platform is called when the user exits the application. It allows HotSync synchronization to take place, with a MobiLink user name of `50`, an empty password, a script version of `custdb`. The HotSync conduit communicates over TCP/IP with a MobiLink server running on the same computer as the conduit (`localhost`), on the default port (2439):

```
ul_synch_info synch_info;
ULInitSynchInfo( &synch_info );
synch_info.name = UL_TEXT("Betty Best");
synch_info.version = UL_TEXT("default");
synch_info.stream = ULConduitStream();
synch_info.stream_parms =
   UL_TEXT("stream=tcpip;host=localhost");
ULSetSynchInfo( &sqlca, &synch_info );
```

Invoking synchronization

The details of how to invoke synchronization depends on your target platform and on the synchronization stream.

The synchronization process can only work if the device running the UltraLite application is able to communicate with the Mobilink server. For some platforms, this means that the device needs to be physically connected by placing it in its cradle or by attaching it to a server computer using the appropriate cable. You need to add error handling code to your application in case the synchronization cannot be carried out.

To invoke synchronization (TCP/IP, TLS, HTTP, or HTTPS streams)

- Call ULInitSynchInfo to initialize the synchronization parameters, and call ULSynchronize to synchronize.

To invoke synchronization (HotSync)

- Call ULInitSynchInfo to initialize the synchronization parameters, and call ULSetSynchInfo to manage synchronization before exiting the application.

  See "ULSetSynchInfo function" on page 292.

The synchronization call requires a structure that holds a set of parameters describing the specifics of the synchronization. The particular parameters used depend on the stream.

Commit all changes before synchronizing

An UltraLite database cannot have uncommitted changes when it is synchronized. If you attempt to synchronize an UltraLite database when any connection has an uncommitted transaction, the synchronization fails, an exception is thrown and the SQLE_UNCOMMITTED_TRANSACTIONS error is set. This error code also appears in the MobiLink server log.

For more information about download-only synchronizations, see "Download Only synchronization parameter" [*UltraLite - Database Management and Reference*].

Adding initial data to your application

Many UltraLite applications need data to start working. You can download data into your application by synchronizing. You may want to add logic to your application to ensure that, the first time it is run, it downloads all necessary data before any other actions are carried out.

Performance tip
It is easier to locate errors if you develop an application in stages. When developing a prototype, temporarily use INSERT statements in your application to provide data for testing and demonstration purposes. Once your prototype is working correctly, replace the temporary INSERT statements with the code to perform the synchronization.

For more synchronization development tips, see "MobiLink development tips" [*MobiLink - Server Administration*].

Handling synchronization communications errors

The following code illustrates how to handle communications errors from embedded SQL applications:

```
if( psqlca->sqlcode == SQLE_COMMUNICATIONS_ERROR ) {
     printf( "  Stream error information:\n"
                "    stream_error_code = %ld\t(ss_error_code)\n"
                "    error_string      = \"%s\"\n"
                "    system_error_code = %ld\n",
        (long)info.stream_error.stream_error_code,
        info.stream_error.error_string,
        (long)info.stream_error.system_error_code );
```

SQLE_COMMUNICATIONS_ERROR is the general error code for communications errors. More information about the specific error is supplied to your application in the members of the stream_error synchronization parameter.

To keep UltraLite small, the runtime reports numbers rather than messages.

Monitoring and canceling synchronization

This section describes how to monitor and cancel synchronization from UltraLite applications.

Monitoring synchronization

- Specify the name of your callback function in the observer member of the synchronization structure (ul_synch_info).
- Call the synchronization function or method to start synchronization.
- UltraLite calls your callback function whenever the synchronization state changes. The following section describes the synchronization state.

The following code shows how this sequence of tasks can be implemented in an embedded SQL application:

```
ULInitSynchInfo( &info );
info.user_name = m_EmpIDStr;
...
//The info parameter of ULSynchronize() contains
// a pointer to the observer function
info.observer = ObserverFunc;
ULSynchronize( &sqlca, &info );
```

Handling synchronization status information

The callback function that monitors synchronization takes a ul_synch_status structure as parameter. For additional information, see "ul_synch_status struct" on page 141.

The ul_synch_status structure has the following members:

```
struct ul_synch_status {
    struct {
       ul_u_long   bytes;
       ul_u_long   inserts;
       ul_u_long   updates;
       ul_u_long   deletes;
     }               sent;
    struct {
       ul_u_long   bytes;
       ul_u_long   inserts;
       ul_u_long   updates;
```

```
        ul_u_long    deletes;
      }                received;
    p_ul_synch_info  info;
    ul_synch_state   state;
    ul_u_short       db_tableCount;
    ul_u_short       table_id;
    char             table_name[];
    ul_wchar         table_name_w2[];
    ul_u_short       sync_table_count;
    ul_u_short       sync_table_index;
    ul_sync_state    state;
    ul_bool          stop;
    ul_u_short       flags;
    ul_void *        user_data;
    SQLCA *          sqlca;
}
```

- **sent.inserts** The number of inserted rows that have been uploaded so far.
- **sent.updates** The number of updated rows that have been uploaded so far.
- **sent.deletes** The number of deleted rows that have been uploaded so far.
- **sent.bytes** The number of bytes that have been uploaded so far.
- **received.inserts** The number of inserted rows that have been downloaded so far.
- **received.updates** The number of updated rows that have been downloaded so far.
- **received.deletes** The number of deleted rows that have been downloaded so far.
- **received.bytes** The number of bytes that have been downloaded so far.
- **info** A pointer to the ul_synch_info structure. See "ul_synch_info_a struct" on page 133.
- **db_tableCount** Returns the number of tables in the database.
- **table_id** The current table number (relative to 1) that is being uploaded or downloaded. This number may skip values when not all tables are being synchronized and is not necessarily increasing.
- **table_name[]** Name of the current table.
- **table_name_w2[]** Name of the current table (wide character version). This field is only populated in the Windows (desktop and Mobile) environment.
- **sync_table_count** Returns the number of tables being synchronized.
- **sync_table_index** The number of the table that is being uploaded or downloaded, starting at 1 and ending at the **sync_table_count** value. This number may skip values when not all tables are being synchronized.
- **state** One of the following states:
  - **UL_SYNCH_STATE_STARTING** No synchronization actions have yet been taken.
  - **UL_SYNCH_STATE_CONNECTING** The synchronization stream has been built, but not yet opened.
  - **UL_SYNCH_STATE_SENDING_HEADER** The synchronization stream has been opened, and the header is about to be sent.

- **UL_SYNCH_STATE_SENDING_TABLE** A table is being sent.
- **UL_SYNCH_STATE_SENDING_DATA** Schema information or data is being sent.
- **UL_SYNCH_STATE_FINISHING_UPLOAD** The upload stage is completed and a commit is being carried out.
- **UL_SYNCH_STATE_RECEIVING_UPLOAD_ACK** An acknowledgement that the upload is complete is being received.
- **UL_SYNCH_STATE_RECEIVING_TABLE** A table is being received.
- **UL_SYNCH_STATE_RECEIVING_DATA** Schema information or data is being received.
- **UL_SYNCH_STATE_COMMITTING_DOWNLOAD** The download stage is completed and a commit is being carried out.
- **UL_SYNCH_STATE_SENDING_DOWNLOAD_ACK** An acknowledgement that download is complete is being sent.
- **UL_SYNCH_STATE_DISCONNECTING** The synchronization stream is about to be closed.
- **UL_SYNCH_STATE_DONE** Synchronization has completed successfully.
- **UL_SYNCH_STATE_ERROR** Synchronization has completed, but with an error.
- **UL_SYNCH_STATE_ROLLING_BACK_DOWNLOAD** An error occurred during download and the download is being rolled back.

  For more information about the synchronization process, see "The synchronization process" [*MobiLink - Getting Started*].

- **stop** Set this member to true to interrupt the synchronization. The SQL exception SQLE_INTERRUPTED is set, and the synchronization stops as if a communications error had occurred. The observer is *always* called with either the DONE or ERROR state so that it can do proper cleanup.
- **flags** Returns the current synchronization flags indicating additional information related to the current state.
- **user_data** Returns the user data object that is passed as an argument to the ULRegisterSynchronizationCallback function.
- **sqlca** Pointer to the connection's active SQLCA.

Example

The following code illustrates a very simple observer function:

```
extern void __stdcall ObserverFunc(
    p_ul_synch_status status )
{
    switch( status->state ) {
        case UL_SYNCH_STATE_STARTING:
        printf( "Starting\n");
        break;
        case UL_SYNCH_STATE_CONNECTING:
            printf( "Connecting\n"  );
            break;
        case UL_SYNCH_STATE_SENDING_HEADER:
                printf( "Sending Header\n" );
                break;
            case UL_SYNCH_STATE_SENDING_TABLE:
                printf( "Sending Table %d of %d\n",
                        status->tableIndex + 1,
                        status->tableCount );
                break;
            case UL_SYNCH_RECEIVING_UPLOAD_ACK:
                printf( "Receiving Upload Ack\n" );
                break;
            case UL_SYNCH_STATE_RECEIVING_TABLE:
                printf( "Receiving Table %d of %d\n",
                        status->tableIndex + 1,
                        status->tableCount );
                break;
            case UL_SYNCH_STATE_SENDING_DOWNLOAD_ACK:
                printf( "Sending Download Ack\n" );
                break;
            case UL_SYNCH_STATE_DISCONNECTING:
                printf( "Disconnecting\n" );
                break;
            case UL_SYNCH_STATE_DONE:
                printf( "Done\n" );
                break;
            break;
    ...
```

This observer produces the following output when synchronizing two tables:

```
Starting
Connecting
Sending Header
Sending Table 1 of 2
Sending Table 2 of 2
Receiving Upload Ack
Receiving Table 1 of 2
Receiving Table 2 of 2
Sending Download Ack
Disconnecting
Done
```

CustDB example

An example of an observer function is included in the CustDB sample application. The implementation in CustDB provides a window that displays synchronization progress and allows the user to cancel synchronization. The user-interface component makes the observer function platform specific.

The CustDB sample code is in the *samples-dir\UltraLite\CustDB* directory. The observer function is contained in platform-specific subdirectories of the *CustDB* directory.

Building embedded SQL applications

This section describes a general build procedure for UltraLite embedded SQL applications.

This section assumes a familiarity with the overall embedded SQL development model.

General build procedure

Sample code

You can find a makefile that uses this process in the *samples-dir\UltraLite\ESQLSecurity* directory.

Separately licensed component required
ECC encryption and FIPS-certified encryption require a separate license. All strong encryption technologies are subject to export regulations.

See "Separately licensed components" [*SQL Anywhere 11 - Introduction*].

Procedure

To build an UltraLite embedded SQL application

1. Run the SQL preprocessor on *each* embedded SQL source file.

   The SQL preprocessor is the sqlpp command line utility. It preprocesses the embedded SQL source files, producing C++ source files to be compiled into your application.

   For more information about the SQL preprocessor, see "SQL Preprocessor for UltraLite utility (sqlpp)" [*UltraLite - Database Management and Reference*].

   Caution
   sqlpp overwrites the output file without regard to its contents. Ensure that the output file name does not match the name of any of your source files. By default, *sqlpp* constructs the output file name by changing the suffix of your source file to *.cpp*. When in doubt, specify the output file name explicitly, following the name of the source file.

2. Compile *each* C++ source file for the target platform of your choice. Include:
   - each C++ file generated by the SQL preprocessor
   - any additional C or C++ source files required by your application
3. Link *all* these object files, together with the UltraLite runtime library.

Configuring development tools for embedded SQL development

Many development tools use a dependency model, sometimes expressed as a makefile, in which the timestamp on each source file is compared with that on the target file (object file, in most cases) to decide whether the target file needs to be regenerated.

With UltraLite development, a change to any SQL statement in a development project means that the generated code needs to be regenerated. Changes are not reflected in the timestamp on any individual source file because the SQL statements are stored in the reference database.

This section describes how to incorporate UltraLite application development, specifically the SQL preprocessor, into a dependency-based build environment. The specific instructions provided are for Visual C++, and you may need to modify them for your own development tool.

The UltraLite plug-in for CodeWarrior automatically provides Palm Computing Platform developers with the techniques described here. For more information about this plug-in, see "Developing UltraLite applications for the Palm OS" on page 67.

SQL preprocessing

The first set of instructions describes how to add instructions to run the SQL preprocessor to your development tool.

To add embedded SQL preprocessing into a dependency-based development tool

1. Add the *.sqc* files to your development project.

   The development project is defined in your development tool.

2. Add a custom build rule for each *.sqc* file.

   - The custom build rule should run the SQL preprocessor. In Visual C++, the build rule should have the following command (entered on a single line):

     ```
     "%SQLANY11%\Bin32\sqlpp.exe" -q -u $(InputPath) $(InputName).cpp
     ```

     where SQLANY11 is an environment variable that points to your SQL Anywhere installation directory.

     For a full description of the SQL preprocessor command line, see "SQL Preprocessor for UltraLite utility (sqlpp)" [*UltraLite - Database Management and Reference*].

   - Set the output for the command to **$(InputName).cpp**.

3. Compile the *.sqc* files, and add the generated *.cpp* files to your development project.

   You need to add the generated files to your project even though they are not source files, so that you can set up dependencies and build options.

4. For each generated *.cpp* file, set the preprocessor definitions.

   - Under General or Preprocessor, add UL_USE_DLL to the Preprocessor definitions.

- Under Preprocessor, add *$(SQLANY11)\SDK\Include* and any other include folders you require to your include path, as a comma-separated list.

Developing UltraLite applications for the Palm OS

Contents

You can use the CodeWarrior plug-in to simplify the creation of UltraLite applications using embedded SQL or UltraLite C++.

The plug-in is supplied in the *install-dir\UltraLite\Palm\68k\cwplugin* directory. Read the file *readme.txt* in that directory.

CodeWarrior includes a version of the Palm SDK. Depending on the particular devices you are targeting, you may want to upgrade your Palm SDK to a more recent version than that included with the development tool.

For a list of supported platforms, see "Supported platforms" [*SQL Anywhere 11 - Introduction*].

For a list of supported target operating systems, see "Supported platforms" [*SQL Anywhere 11 - Introduction*].

Installing the UltraLite plug-in for CodeWarrior

The files for the UltraLite plug-in for CodeWarrior are placed on your disk during UltraLite installation, but you cannot use the plug-in without an additional installation step.

Installing the UltraLite plug-in for CodeWarrior

1. From a command prompt, change to the *install-dir\UltraLite\Palm\68k\cwplugin* directory.
2. Run *install.bat* to copy the appropriate files into your CodeWarrior installation directory. The *install.bat* file requires two arguments:
   - CodeWarrior directory
   - CodeWarrior version.

   For example, the following command on one line, installs the plug-in for CodeWarrior 9 in the default CodeWarrior installation directory.

   ```
   install "c:\Program Files\Metrowerks\CodeWarrior for Palm OS Platform 9.0"
   r9
   ```

   You need double quotes around the directory if the path has any embedded spaces.

Uninstalling the CodeWarrior plug-in

You can use *uninstall.bat* to uninstall the UltraLite plug-in from CodeWarrior. The *uninstall.bat* file requires the same arguments as described above for *install.bat*.

Creating UltraLite projects in CodeWarrior

To create an UltraLite project in CodeWarrior

1. Start CodeWarrior.
2. Create a new project:
   a. From the **CodeWarrior** menu, choose **File** » **New**.
   b. Click the **Projects** tab.
   c. Choose **Palm OS Application Stationery**.
   d. Choose a name and location for the project and click **OK**.
3. Choose an UltraLite stationery.

   The UltraLite plug-in adds the following choices to the stationery list:

   - Palm OS UltraLite C++ App
   - Palm OS UltraLite ESQL App

   Choose the development model you want to use and click **OK** to create the project.

   The stationery is standard C stationery for embedded SQL, and standard C++ stationery for C++.
4. If you are using embedded SQL, configure the settings in the **UltraLite Preprocessor Panel** for your project. If you are using C++ these settings are ignored.
   a. On your project window (*.mcp*), click the **Settings** icon on the toolbar.
   b. In the tree in the left pane, choose **Target** » **UltraLite Preprocessor**. Enter the settings for your project.

Preprocessing

When you build an embedded SQL project, the UltraLite plug-in calls *sqlpp* to preprocess *.sqc* files into *.c/.cpp* files and also converts ESQL statements to UltraLite function calls.

When building UltraLite Embedded SQL or C++ applications in the CodeWarrior environment, the plug-in does not add the access paths to *install-dir\SDK\Include* and *install-dir\UltraLite\Palm\68k\lib\cw\* and to the UltraLite libraries *ulrt.lib* and *ulbase.lib*. The plug-in only adds the generated files from running the SQL preprocessor to the CodeWarrior project for UltraLite Embedded SQL applications.

Converting an existing CodeWarrior project to an UltraLite application

If you install the UltraLite plug-in for CodeWarrior, you will be asked to convert older projects when they are first opened. In this conversion, CodeWarrior sets the default SQL preprocessor settings and saves them in the project file. This causes no disruption to projects that do not use the SQL preprocessor. If you want to further convert a project to invoke the SQL preprocessor automatically, you need to do the following:

1. Add a file mapping entry for *.sqc* files to the File Mappings panel of the Target settings.

   The file type is **TEXT** and the Compiler is **UltraLite Preprocessor**. All flags for these files must be unchecked.

2. For embedded SQL applications, remove all generated files from the Files view. These files are automatically generated and re-added when the *.sqc* files are built.
3. Remove any file mappings for *.ulg* files.
4. Verify the references to required library files.

Using the UltraLite plug-in for CodeWarrior

For embedded SQL, the UltraLite plug-in for CodeWarrior integrates the UltraLite preprocessing steps into the CodeWarrior compilation model. It ensures that the SQL preprocessor runs when required.

Using prefix files

A prefix file is a header file that all source files in a CodeWarrior project must include. You should use *install-dir\SDK\Include\ulpalmos.h* as your prefix file.

If you have your own prefix file, it must include *ulpalmos.h*. The *ulpalmos.h* file defines macros required by UltraLite Palm applications and also sets CodeWarrior compiler options required by UltraLite.

Encrypted synchronization
If you are using either the TLS or HTTPS protocols to implement encrypted synchronization, you must add *ulrsa.lib*, *ulecc.lib*, or *ulfips.lib* and *gselst.lib* to your CodeWarrior UltraLite projects.

Building the CustDB sample application in CodeWarrior

CustDB is a simple sales-status application.

For a diagram of the sample database schema, see "About the CustDB sample database" [*SQL Anywhere 11 - Introduction*].

Files for the application are located in the *samples-dir\UltraLite\CustDB* directory. Generic files are located in the *CustDB* directory. Files specific to CodeWarrior for the Palm OS are in the following directories:

- *samples-dir\UltraLite\CustDB\cwcommon*
- *samples-dir\UltraLite\CustDB\cw*

The instructions in this section describe how to build the CustDB application using CodeWarrior 9.

To build the CustDB sample application using CodeWarrior

1. Start the CodeWarrior IDE.
2. Open the CustDB project file:
   - Choose **File** » **Open**.
   - Open the project file *samples-dir\UltraLite\CustDB\cw\custdb.mcp*
3. To build the target application (*custdb.prc*), choose **Project** » **Make**.

Building Expanded Mode applications

CodeWarrior supports a code generation mode called expanded mode, which improves memory use for global data. If you are using CodeWarrior version 9 you can use expanded mode with an A5-based jump table. To do so, you must use the expanded mode version of the UltraLite runtime library and the UltraLite base library. The expanded mode versions of these libraries are located as follows:

- *install-dir\UltraLite\Palm\68k\lib\cw9_a4a5jt\ulrt.lib*
- *install-dir\UltraLite\Palm\68k\lib\cw9_a4a5jt\ulbase.lib*

Expanded mode may be helpful for large applications that would otherwise exceed the 64 KB global data limit. A limitation of expanded mode is that encrypted synchronization can be used only via HotSync, as the synchronization security libraries for UltraLite do not use expanded mode.

Maintaining state in UltraLite Palm applications (deprecated)

You can save the state of tables and cursors when an application is closed by suspending the connection instead of closing it.

The current state is only stored for tables that are open when the connection object remains open.

Whenever your UltraLite application is closed or the user switches to another application, UltraLite saves the state of any open cursors and tables.

1. When the user returns to the application, call the appropriate open methods:
   - For embedded SQL, call the following functions:
     - db_init
     - EXEC SQL CONNECT
   - For C++, call the following functions:
     - ULSqlca.Initialize
     - ULInitDatabaseManager
     - OpenConnection
2. Confirm the connection was restored properly by checking that the SQLCODE is SQLE_CONNECTION_RESTORED.
3. For cursor objects, including instances of generated result set classes, you can do one of the following:
   - Ensure the object is closed when the user switches away from the application, and call Open when you next need the object. If you choose this option, the current position is not restored.
   - Do not close the object when the user switches away, and call Reopen when you next need to access the object. The current position is then maintained, but the application takes more memory in the Palm when the user is using other applications.
4. For table objects, including instances of generated table classes, you cannot save a position. You must close table objects before a user moves away from the application, and Open them when the user needs them again. Do not use Reopen on table objects.

Closing a connection rolls back any uncommitted transactions. By not closing connection objects, any outstanding transactions are saved (not committed), so that when the application restarts, those transactions appear and can be committed or rolled back. Uncommitted changes are not synchronized.

Restoring state in UltraLite Palm applications (deprecated)

When an application restarts on the Palm OS, UltraLite restores the state of any cursors or tables that were not explicitly closed when the application was most recently shut down.

Saving, retrieving, and clearing encryption keys on Palm OS

On Palm OS, applications are automatically shut down by the system whenever a user switches to a different application. Therefore, if you encrypt an UltraLite database on Palm OS, the user is prompted to re-enter the key *each* time they switch back to the application.

To avoid re-entering the encryption key

1. Save the encryption key in dynamic memory as a Palm feature.

   Features are indexed by creator and a feature number. Applications can then pass in their creator ID or NULL, along with the feature number or NULL, to save and retrieve the encryption key.

2. Program the application to retrieve the key upon re-launch.

   Note
   Because the encryption key is cleared on any reset of the device, the retrieval of the key will fail at this time. The user is prompted to re-enter the key in this case.

   The following sample code (embedded SQL) illustrates how to save and retrieve the encryption key:

   ```
   startupRoutine() {
      ul_char buffer[MAX_PWD];

      if( !ULRetrieveEncryptionKey(
             buffer, MAX_PWD, NULL, NULL  ) ){
         // prompt user for key
         userPrompt( buffer, MAX_PWD );

         if( !ULSaveEncryptionKey( buffer, NULL, NULL ) ) {
            // inform user save failed
         }
      }
   }
   ```

3. Use a menu item to clear the encryption key and thereby secure the device.

   The following code sample shows the method to accommodate this goal:

   ```
   case MenuItemClear
       ULClearEncryptionKey( NULL, NULL );
       break;
   ```

See also

- For UltraLite.NET: "ULConnectionParms class" [*UltraLite - .NET Programming*]
- For UltraLite C++: "UltraLite_Connection_iface class" on page 156
- "UltraLite DBKEY connection parameter" [*UltraLite - Database Management and Reference*]
- For UltraLite for M-Business Anywhere: "Database encryption and obfuscation" [*UltraLite - M-Business Anywhere Programming*]
- "UltraLite obfuscate creation parameter" [*UltraLite - Database Management and Reference*]

Registering the Palm creator ID

UltraLite applications for the Palm OS, like all Palm OS applications, require a creator ID. You assign this creator ID to your application at creation time and, if you are using HotSync synchronization, you register the creator ID with HotSync manager for use by the MobiLink synchronization.

For information about assigning creator IDs to applications, see your development tool documentation. For more information about registering creator IDs with HotSync manager, see "HotSync on Palm OS" [*UltraLite - Database Management and Reference*].

The creator ID is a string from one to four characters long. The first character should be an uppercase letter, as Palm OS reserves the use of an initial lowercase letter for PALM OS system use.

Adding HotSync synchronization to Palm applications

HotSync synchronization takes place when an UltraLite application is closed. It is initiated by the HotSync.

If you use HotSync, then you synchronize by calling ULSetSynchInfo before closing the application. Do not use ULSynchronize or ULConnection.Synchronize for HotSync synchronization.

To enable HotSync synchronization from your application you must add code for the following steps:

1. Prepare a ul_synch_info structure.
2. Call the ULSetSynchInfo function, supplying the ul_synch_info structure as an argument.

   This function is called when the user switches away from the UltraLite application. You must ensure that all outstanding operations are committed before calling db_fini. The ul_synch_info.stream parameter is ignored, and so does not need to be set.

   For example:

   ```
   //C++ API
   ul_synch_info info;
   ULInitSynchInfo( &info );
   info.stream_parms =
     UL_TEXT( "stream=tcpip;host=localhost" );
   info.user_name = UL_TEXT( "50" );
   info.version = UL_TEXT( "custdb" );

   ULSetSynchInfo( &sqlca, &info );

   if( !db.Close( ) ) {
     return( false );
   }
   ```
3. Call db_fini.

See "Maintaining state in UltraLite Palm applications (deprecated)" on page 74 and "Synchronization parameters for UltraLite" [*UltraLite - Database Management and Reference*].

An UltraLite HotSync conduit is required for HotSync synchronization of UltraLite applications. If there are uncommitted transactions when you close your Palm application, and if you synchronize, the conduit reports that synchronization fails because of uncommitted changes in the database.

Specifying stream parameters

The synchronization stream parameters in the ul_synch_info structure control communication with the MobiLink server. For HotSync synchronization, the UltraLite application does not communicate directly with a MobiLink server; it is the HotSync conduit instead.

You can supply synchronization stream parameters to govern the behavior of the MobiLink conduit in one of the following ways:

- Supply the required information in the stream_parms member of ul_synch_info passed to ULSetSynchInfo.

For a list of available values, see "Network protocol options for UltraLite synchronization streams" [*UltraLite - Database Management and Reference*].

- Supply a null value for the stream_parms member. The MobiLink conduit then searches in the *ClientParms* registry entry on the computer where it is running for information on how to connect to the MobiLink server.

  The stream and stream parameters in the registry entry are specified in the same format as in the ul_synch_info structure stream_parms field.

  See "UltraLite synchronization parameters and network protocol options" [*UltraLite - Database Management and Reference*].

See also

- "HotSync on Palm OS" [*UltraLite - Database Management and Reference*]

Adding TCP/IP, HTTP, or HTTPS synchronization to Palm applications

This section describes how to add TCP/IP, HTTP, or HTTPS synchronization to your Palm application.

For a general description of how to add synchronization to UltraLite applications, see "Adding synchronization to your UltraLite application" [*UltraLite - Database Management and Reference*].

Transport layer security on the Palm OS

You can use transport-layer security with Palm applications built with CodeWarrior.

For more information about transport-layer security, see "Encrypting MobiLink client/server communications" [*SQL Anywhere Server - Database Administration*].

Palm devices can synchronize using TCP/IP, HTTP, or HTTPS communication by setting the stream member of the ul_synch_info structure to the appropriate stream, and calling ULSynchronize or ULConnection.Synchronize to carry out the synchronization.

When using TCP/IP, HTTP, or HTTPS synchronization, db_init or db_fini save and restore the state of the application on exiting and activating the application, but do not participate in synchronization.

Before closing the application, set the synchronization information using ULSetSynchInfo, providing the ul_synch_info structure as an argument.

When using TCP/IP, HTTP, or HTTPS synchronization from a Palm device, you must specify an explicit host name or IP address in the stream_parms member of the ul_synch_info structure. Specifying NULL defaults to `localhost`, which represents the device, not the host.

For more information about the ul_synch_info structure, see "Network protocol options for UltraLite synchronization streams" [*UltraLite - Database Management and Reference*].

Deploying Palm applications

This section describes the following aspects of deploying Palm applications:

- Deploying the application.

  See "Deploying the application" on page 80.
- Deploying the UltraLite synchronization conduit for HotSync.

  See "HotSync on Palm OS" [*UltraLite - Database Management and Reference*].
- Deploying an initial copy of the UltraLite database.

  See "Deploying UltraLite databases" on page 80.

Install your UltraLite application on your Palm device as you would any other Palm OS application.

Deploying the application

To install an application on a Palm device

1. Open the Install Tool, included with your Palm Desktop Organizer Software.
2. Choose **Add** and specify the location of your compiled application (*.prc* file).
3. Close the Install Tool.
4. Use the HotSync utility to copy the application to your Palm device.

Deploying the MobiLink synchronization conduit

For applications using HotSync synchronization, each end user must have the MobiLink synchronization conduit installed on their desktop.

For more information about installing the MobiLink synchronization conduit, see "HotSync on Palm OS" [*UltraLite - Database Management and Reference*].

Deploying UltraLite databases

If you deploy your application without a database, the application must contain relatively complex code to create the database. The recommended approach is to create an initial database on a Windows desktop and copy the database file to the Palm device. Sybase Central (or the utility ulcreate) can be used to create an initial database. The user must then obtain an initial copy of data on the first synchronization. You can use the uldbutil utility to back up the UltraLite database to the PC. To deploy many UltraLite databases with an initial database including data, you can perform an initial synchronization and then back up the UltraLite database. The database can be deployed on other devices so they do not need to perform an initial synchronization.

See "UltraLite Data Management utility for Palm OS (ULDBUtil)" [*UltraLite - Database Management and Reference*].

If you are using HotSync synchronization, each of your end users must also install the synchronization conduit onto their desktop computer.

For more information about installing the synchronization conduit, see "Deploy the UltraLite HotSync conduit" [*UltraLite - Database Management and Reference*].

If you deploy a database using HotSync, HotSync sets a backup bit on the database. When this backup bit is set, the entire database is backed up to the desktop computer on each synchronization. This behavior is generally not appropriate for UltraLite databases. When an UltraLite application is launched, the Palm data store is checked to see if its backup bit is set to true. If it is set, it is cleared. If it is not set, there is no change.

If you want the backup bit to remain set to true, you can set the store parameter palm_allow_backup in the database connection string.

Developing UltraLite applications for Windows Mobile

Contents

Microsoft eMbedded Visual C++ can be used to develop applications for the Windows Mobile environment. This development environment is available from Microsoft as part of eMbedded Visual Tools.

You can download eMbedded Visual C++ from the Microsoft Developer Network at http://msdn.microsoft.com/.

Applications targeting Windows Mobile should use the default setting for wchar_t and link against the UltraLite runtime libraries in *install-dir\ultralite\ce\arm.50\lib\*.

For a list of supported host platforms and development tools for Windows Mobile development, and for a list of supported target Windows Mobile platforms, see "Supported platforms" [*SQL Anywhere 11 - Introduction*].

You can test your applications under an emulator on most Windows Mobile target platforms.

Choosing how to link the runtime library

Windows Mobile supports dynamic link libraries. At link time, you have the option of linking your UltraLite application to the runtime DLL using an imports library, or statically linking your application using the UltraLite runtime library.

Performance tip
If you have a single UltraLite application on your target device, a statically linked library uses less memory. If you have multiple UltraLite applications on your target device, using the DLL may be more economical in memory use.

If you are repeatedly downloading UltraLite applications to a device, over a slow link, then you may want to use the DLL to minimize the size of the downloaded executable, after the initial download.

To build and deploy an application using the UltraLite runtime DLL

1. Preprocess your code, then compile the output with UL_USE_DLL.
2. Link your application using the UltraLite imports library.
3. Copy both your application executable and the UltraLite runtime DLL to your target device.

Building the CustDB sample application

CustDB is a simple sales-status application. It is located in the *samples-dir\UltraLite\* directory of your SQL Anywhere installation. Generic files are located in the *CustDB* subdirectory. Files specific to Windows Mobile are located in the *EVC* subdirectory of *CustDB*.

The CustDB application is provided as an eMbedded Visual C++ 3.0 project.

For a diagram of the sample database schema, see "About the CustDB sample database" [*SQL Anywhere 11 - Introduction*].

To build the CustDB sample application

1. Start eMbedded Visual C++.
2. Open the project file that corresponds to your version of eMbedded Visual C++:
   - *samples-dir\UltraLite\CustDB\EVC\EVCCustDB.vcp* for eVC 3.0.
   - *samples-dir\UltraLite\CustDB\EVC40\EVCCustDB.vcp* for eVC 4.0.
3. Choose **Build** » **Set Active Platform** to set the target platform.
   - Set a platform of your choice.
4. Choose **Build** » **Set Active Configuration** to select the configuration.
   - Set an active configuration of your choice.
5. If you are building CustDB for the Pocket PC x86em emulator platform only:
   - Choose **Project** » **Settings**.
   - On the **Link** tab, in the **Object/library Modules field**, change the UltraLite runtime library entry to the *emulator30* directory rather than the *emulator* directory.
6. Build the application:
   - Press **F7** or choose **Build** » **Build EVCCustDB.exe** to build CustDB.

     When eMbedded Visual C++ has finished building the application, it automatically attempts to upload it to the remote device.
7. Start the MobiLink server:
   - To start the MobiLink server, from the **Start** menu choose **Programs** » **SQL Anywhere 11** » **MobiLink** » **Synchronization Server Sample**.
8. Run the CustDB application:

   Before running the CustDB application, the custdb database must be copied to the root folder of the device. Copy the database file named *samples-dir\UltraLite\CustDB\custdb.udb* to the root of the device.

   Press Ctrl+F5 or choose **Build** » **Execute CustDB.exe**.

Folder locations and environment variables
The sample project uses environment variables wherever possible. It may be necessary to adjust the project for the application to build properly. If you experience problems, try searching for missing files in the Microsoft Visual C++ folder(s) and adding the appropriate directory settings.

For embedded SQL, the build process uses the SQL preprocessor, *sqlpp*, to preprocess the file *CustDB.sqc* into the file *CustDB.cpp*. This one-step process is useful in smaller UltraLite applications where all the embedded SQL can be confined to one source module. In larger UltraLite applications, you need to use multiple *sqlpp* invocations.

See "Building embedded SQL applications" on page 63.

Storing persistent data

The UltraLite database is stored in the Windows Mobile file system. The default file is *\UltraLiteDB \ul_<project>.udb*, with *project* being truncated to eight characters. You can override this choice using the **file_name** connection parameter which specifies the full path name of the file-based persistent store.

The UltraLite runtime carries out no substitutions on the **file_name** parameter. If a directory has to be created for the file name to be valid, the application must ensure that any directories are created before calling **db_init**.

As an example, you could make use of a flash memory storage card by scanning for storage cards and prefixing a name by the appropriate directory name for the storage card. For example,

```
file_name = "\\Storage Card\\My Documents\\flash.udb"
```

Deploying Windows Mobile applications

When compiling UltraLite applications for Windows Mobile, you can link the UltraLite runtime library either statically or dynamically. If you link it dynamically, you must copy the UltraLite runtime library for your platform to the target device.

To build and deploy an application using the UltraLite runtime DLL

1. Preprocess your code, then compile the output with UL_USE_DLL.
2. Link your application using the UltraLite imports library.
3. Copy both your application executable and the UltraLite runtime DLL to your target device.

   The UltraLite runtime DLL is in chip-specific directories under the *\ultralite\ce* subdirectory of your SQL Anywhere installation directory.

To deploy the UltraLite runtime DLL for the Windows Mobile emulator, place the DLL in the appropriate subdirectory of your Windows Mobile tools directory. The following directory is the default setting for the Pocket PC emulator:

```
C:\Program Files\Windows CE Tools\wce300\MS Pocket PC\
emulation\palm300\windows
```

Deploying applications that use ActiveSync

Applications that use ActiveSync synchronization must be registered with ActiveSync and copied to the device. See "Register applications with the ActiveSync Manager" [*UltraLite - Database Management and Reference*].

The MobiLink provider for ActiveSync must also be installed. See "Deploy the ActiveSync provider for UltraLite" [*UltraLite - Database Management and Reference*].

Assigning class names for applications

When registering applications for use with ActiveSync you must supply a window class name. Assigning class names is carried out at development time and your application development tool documentation is the primary source of information on the topic.

Microsoft Foundation Classes (MFC) dialog boxes are given a generic class name of **Dialog**, which is shared by all dialogs in the system. This section describes how to assign a distinct class name for your application if you are using MFC and eMbedded Visual C++.

To assign a window class name for MFC applications using eMbedded Visual C++

1. Create and register a custom window class for dialog boxes, based on the default class.

   Add the following code to your application's startup code. The code must be executed before any dialogs get created:

   ```
   WNDCLASS wc;
   if( ! GetClassInfo( NULL, L"Dialog", &wc ) ) {
    AfxMessageBox( L"Error getting class info" );
   }
   wc.lpszClassName = L"MY_APP_CLASS";
   if( ! AfxRegisterClass( &wc ) ) {
    AfxMessageBox( L"Error registering class" );
   }
   ```

   where *MY_APP_CLASS* is the unique class name for your application.

2. Determine which dialog is the main dialog for your application.

   If your project was created with the MFC Application Wizard, this is likely to be a dialog named **CMyAppDlg**.

3. Find and record the resource ID for the main dialog.

   The resource ID is a constant of the same general form as IDD_MYAPP_DIALOG.

4. Ensure that the main dialog remains open any time your application is running.

   Add the following line to your application's **InitInstance** function. The line ensures that if the main dialog **dlg** is closed, the application also closes.

   ```
   m_pMainWnd = &dlg;
   ```

   For more information, see the Microsoft documentation for **CWinThread::m_pMainWnd**.

   If the dialog does not remain open for the duration of your application, you must change the window class of other dialogs as well.

5. Save your changes.

   If eMbedded Visual C++ is open, save your changes and close your project and workspace.

6. Modify the resource file for your project.

   - Open your resource file (which has an extension of *.rc*) in a text editor such as Notepad.

   Locate the resource ID of your main dialog.

- Change the main dialog's definition to use the new window class as in the following example. The *only* change that you should make is the addition of the **CLASS** line:

  ```
  IDD_MYAPP_DIALOG DIALOG DISCARDABLE  0, 0, 139, 103
  STYLE WS_POPUP | WS_VISIBLE | WS_CAPTION
  EXSTYLE WS_EX_APPWINDOW | WS_EX_CAPTIONOKBTN
  CAPTION "MyApp"
  FONT 8, "System"
  CLASS "MY_APP_CLASS"
  BEGIN
         LTEXT   "TODO: Place dialog controls here.",IDC_STATIC,
  13,33,112,17
  END
  ```

  where *MY_APP_CLASS* is the name of the window class you used earlier.

- Save the *.rc* file.

7. Reopen eMbedded Visual C++ and load your project.
8. Add code to catch the synchronization message.

   See "Adding ActiveSync synchronization (MFC)" on page 93.

Synchronization on Windows Mobile

UltraLite applications on Windows Mobile can synchronize through the following stream types:

- **ActiveSync** See "Adding ActiveSync synchronization to your application" on page 92.
- **TCP/IP** See "TCP/IP, HTTP, or HTTPS synchronization from Windows Mobile" on page 94.
- **HTTP** See "TCP/IP, HTTP, or HTTPS synchronization from Windows Mobile" on page 94.

The *user_name* and *stream_parms* parameters must be surrounded by the **UL_TEXT()** macro for Windows Mobile when initializing, since the compilation environment is Unicode wide characters.

For more information about synchronization parameters, see "Synchronization parameters for UltraLite" [*UltraLite - Database Management and Reference*].

Adding ActiveSync synchronization to your application

ActiveSync is software from Microsoft that handles data synchronization between a desktop computer running Windows and a connected Windows Mobile handheld device. UltraLite supports ActiveSync versions 3.5 and later.

This section describes how to add ActiveSync provider to your application, and how to register your application for use with ActiveSync on your end users' computers.

If you use ActiveSync, synchronization can be initiated only by ActiveSync itself. ActiveSync automatically initiates a synchronization when the device is placed in the cradle or when the Synchronization command is selected from the ActiveSync window. The MobiLink provider starts the application, if it is not already running, and sends a message to the application.

For more information about setting up ActiveSync synchronization, see "Deploying applications that use ActiveSync" on page 89.

The ActiveSync provider uses the **wParam** parameter. A **wParam** value of 1 indicates that the MobiLink provider for ActiveSync launched the application. The application must then shut itself down after it has finished synchronizing. If the application was already running when called by the MobiLink provider for ActiveSync, **wParam** is 0. The application can ignore the **wParam** parameter if it wants to keep running.

To determine which platforms the provider is supported on, see SQL Anywhere Components by Platform.

Adding synchronization depends on whether you are addressing the Windows API directly or whether you are using the Microsoft Foundation Classes. Both development models are described here.

Adding ActiveSync synchronization (Windows API)

If you are programming directly to the Windows API, you must handle the message from the MobiLink provider in your application's **WindowProc** function, using the **ULIsSynchronizeMessage** function to determine if it has received the message.

Here is an example of how to handle the message:

```
LRESULT CALLBACK WindowProc( HWND hwnd,
         UINT uMsg,
         WPARAM wParam,
         LPARAM lParam )
{
  if( ULIsSynchronizeMessage( uMsg ) ) {
    DoSync();
    if( wParam == 1 ) DestroyWindow( hWnd );
    return 0;
  }
  switch( uMsg ) {
  // code to handle other windows messages
  default:
    return DefWindowProc( hwnd, uMsg, wParam, lParam );
  }
  return 0;
}
```

where **DoSync** is the function that actually calls ULSynchronize.

See "ULIsSynchronizeMessage function" on page 283.

Adding ActiveSync synchronization (MFC)

If you are using Microsoft Foundation Classes to develop your application, you can catch the synchronization message in the main dialog class or in your application class. Both methods are described here.

Your application must create and register a custom window class name for notification. See "Assigning class names for applications" on page 90.

To add ActiveSync synchronization in the main dialog class

1. Add a registered message and declare a message handler.

   Find the message map in the source file for your main dialog (the name is of the same form as *CMyAppDlg.cpp*). Add a registered message using the **static** and declare a message handler using ON_REGISTERED_MESSAGE as in the following example:

   ```
   static UINT WM_ULTRALITE_SYNC_MESSAGE =
    ::RegisterWindowMessage( UL_AS_SYNCHRONIZE );
   BEGIN_MESSAGE_MAP(CMyAppDlg, CDialog)
     //{{AFX_MSG_MAP(CMyAppDlg)
     //}}AFX_MSG_MAP
    ON_REGISTERED_MESSAGE( WM_ULTRALITE_SYNC_MESSAGE,
     OnDoUltraLiteSync )
   END_MESSAGE_MAP()
   ```

2. Implement the message handler.

   Add a method to the main dialog class with the following signature. This method is automatically executed any time the MobiLink provider for ActiveSync requests that your application synchronize. The method should call **ULSynchronize**.

   ```
   LRESULT CMyAppDlg::OnDoUltraLiteSync(
      WPARAM wParam,
      LPARAM lParam
   );
   ```

The return value of this function should be 0.

For more information about handling the synchronization message, see "ULIsSynchronizeMessage function" on page 283.

To add ActiveSync synchronization in the Application class

1. Open the **Class Wizard** for the application class.
2. In the **Messages** list, highlight **PreTranslateMessage** and then click **Add Function**.
3. Click **Edit Code**. The PreTranslateMessage function appears. Change it to read as follows:

```
BOOL CMyApp::PreTranslateMessage(MSG* pMsg)
{
 if( ULIsSynchronizeMessage(pMsg->message) ) {
  DoSync();
  // close application if launched by provider
  if( pMsg->wParam == 1 ) {
   ASSERT( AfxGetMainWnd() != NULL );
   AfxGetMainWnd()->SendMessage( WM_CLOSE );
  }
  return TRUE; // message has been processed
 }
 return CWinApp::PreTranslateMessage(pMsg);
}
```

where **DoSync** is the function that actually calls ULSynchronize.

For more information about handling the synchronization message, see "ULIsSynchronizeMessage function" on page 283.

TCP/IP, HTTP, or HTTPS synchronization from Windows Mobile

For TCP/IP, HTTP, or HTTPS synchronization, the application controls when synchronization occurs. Your application should provide a menu item or user interface control so that the user can request synchronization.

The sample eMbedded Visual C++ project

A sample eMbedded Visual C++ project is provided in the *samples-dir\UltraLite\CEStarter* directory. The workspace file is *samples-dir\UltraLite\CEStarter\ul_wceapplication.vcw*.

When preparing to use eMbedded Visual C++ for UltraLite applications, you should make the following changes to the project settings. The CEStarter application has these changes made.

- Compiler settings:
  - Add *$(SQLANY11)\SDK\Include* to the include path.
  - Define appropriate compiler directives. For example, the UNDER_CE macro should be defined for eMbedded Visual C++ projects.
- Linker settings:
  - Add "*$(SQLANY11)\ultralite\ce\processor\lib\ulrt.lib*"

    where *processor* is the target processor for your application.
  - Add *winsock.lib*.
- The *.sqc* file (embedded SQL only):
  - Add *ul_database.sqc* and *ul_database.cpp* to the project
  - Add the following custom build step for the *.sqc* file:

    ```
    "$(SQLANY11)\Bin32\sqlpp" -q $(InputPath) ul_database.cpp
    ```
  - Set the output file to *ul_database.cpp*.
  - Disable the use of precompiled headers for *ul_database.cpp*.

API Reference

This section provides API reference material for UltraLite C/C++ programmers.

UltraLite C/C++ common API reference

Contents

This section lists functions and macros that you can use with either the embedded SQL or C++ interface. Most of the functions in this section require a SQL Communications Area described in “Understanding the SQL Communications Area” on page 5.

Callback function for ULRegisterErrorCallback

Handles errors that the UltraLite runtime signals to your application.

For a description of error handling using this technique, see "ULRegisterErrorCallback function" on page 120.

Syntax

ul_error_action **UL_GENNED_FN_MOD** *error-callback-function* **(**
SQLCA * *sqlca*,
ul_void * *user_data*,
ul_char * *buffer*
);

Parameters

- **error-callback-function** The name of your function. You must supply the name to ULRegisterErrorCallback.
- **sqlca** A pointer to the SQL communications area (SQLCA).

  The SQLCA contains the SQL code in `sqlca->sqlcode`. Any error parameters have already been retrieved from the SQLCA and stored in *buffer*.

  This sqlca pointer does not necessarily point to the SQLCA in your application, and cannot be used to call back to UltraLite. It is used only to communicate the SQL code to the callback.

  In the C++ component, use the Sqlca.GetCA method.
- **user_data** The user data supplied to ULRegisterErrorCallback. UltraLite does not change this data in any way. Because the callback function may be signaled anywhere in your application, the user_data argument is an alternative to creating a global variable.
- **buffer** The buffer supplied when the callback function was registered. UltraLite fills the buffer with a string, which holds any substitution parameters for the error message. To keep UltraLite as small as possible, UltraLite does not supply error messages themselves. The substitution parameters depend on the specific error. For more information about error parameters for SQL errors, see "SQL Anywhere error messages" [*Error Messages*].

Return value

Returns one of the following actions:

- **UL_ERROR_ACTION_CANCEL** Cancel the operation that raised the error.
- **UL_ERROR_ACTION_CONTINUE** Continue execution, ignoring the operation that raised the error.
- **UL_ERROR_ACTION_DEFAULT** Behave as if there is no error callback.
- **UL_ERROR_ACTION_TRY_AGAIN** Retry the operation that raised the error.

See also

- "ULRegisterErrorCallback function" on page 120
- "SQL Anywhere error messages sorted by Sybase error code" [*Error Messages*]

Callback function for ULRegisterSQLPassthroughCallback

This callback provides script execution progress (status) during SQL passthrough script execution.

Syntax

void ul_sql_passthrough_observer_fn**(** *ul_sql_passthrough_status* * *status* **);**

Parameters

- **ul_sql_passthrough_status** Provides the current status of script execution, including state, number of scripts to be executed, the current script being executed and any user data provided in the register call.

Example

The progress observer callback is defined as follows:

```
typedef void(UL_CALLBACK_FN * ul_sql_passthrough_observer_fn)
    (ul_sql_passthrough_status * status);
```

The ul_sql_passthrough_status structure is defined as follows:

```
typedef struct {
    ul_sql_passthrough_state       state; // current state
    ul_u_long   script_count;          // total number of scripts to execute
    ul_u_long   cur_script;            // current script being executed (1-
based)
    ul_bool     stop;                  // set to true to stop script
execution
                                       // can only be set in the starting
state
    ul_void *   user_data;             // user data provided in register call
    SQLCA *     sqlca;
} ul_sql_passthrough_status;
```

The progress observer callback is registered via the following method:

```
UL_FN_SPEC ul_ret_void UL_FN_MOD ULRegisterSQLPassthroughCallback(
SQLCA *                         sqlca,
ul_sql_passthrough_observer_fn  callback,
ul_void *                       user_data );
```

Here is a sample callback and code to invoke ULRegisterSQLPassthroughCallback:

```
static void UL_GENNED_FN_MOD passthroughCallback(ul_sql_passthrough_status *
status) {
    switch( status->state ) {
        case UL_SQL_PASSTHROUGH_STATE_STARTING:
            printf( "SQL Passthrough script execution starting\n" );
            break;
        case UL_SQL_PASSTHROUGH_STATE_RUNNING_SCRIPT:
           printf( "Executing script %d of %d\n", status->cur_script, status-
>script_count );
            break;
        case UL_SQL_PASSTHROUGH_STATE_DONE:
            printf( "Finished executing SQL Passthrough scripts\n" );
            break;
        default:
            printf( "SQL Passthrough script execution has failed\n" );
            break;
        }
    }

    int main() {
        ULSqlca sqlca;

        sqlca.Initialize();
        ULRegisterSQLPassthroughCallback( sqlca.GetCA(), passthroughCallback,
    NULL );
        DatabaseManager * dm = ULInitDatabaseManager( sqlca );
        ...
    }
```

Return value

Returns one of the following actions *** Are these valid? ***):

- **UL_ERROR_ACTION_CANCEL** Cancel the operation that raised the error.
- **UL_ERROR_ACTION_CONTINUE** Continue execution, ignoring the operation that raised the error.
- **UL_ERROR_ACTION_DEFAULT** Behave as if there is no error callback.
- **UL_ERROR_ACTION_TRY_AGAIN** Retry the operation that raised the error.

See also

- "ULRegisterErrorCallback function" on page 120
- "SQL Anywhere error messages sorted by Sybase error code" [*Error Messages*]

MLFileTransfer function

Downloads a file from a MobiLink server with the MobiLink interface.

Syntax

ul_bool **MLFileTransfer (** ml_file_transfer_info * *info* **);**

Parameters

info A structure containing the file transfer information.

ML File Transfer parameters

The ML File Transfer parameters are members of a structure that is passed as a parameter to the MLFileTransfer function. The ml_file_transfer_info structure is defined in a header file named *mlfiletransfer.h*. The individual fields of the structure are specified as follows:

filename Required. The file name to be transferred from the server running MobiLink. MobiLink searches the *username* subdirectory first, before defaulting to the root directory. See “-ftr option” [*MobiLink - Server Administration*].

If the file cannot be found, an error is set in the error field. The file name must not include any drive or path information, or MobiLink cannot find it.

dest_path The local path to store the downloaded file. If this parameter is empty (the default), the downloaded file is stored in the current directory.

- On Windows Mobile, if dest_path is empty, the file is stored in the root (\) directory of the device.
- On the desktop, if the dest_path is empty, the file is stored in the user's current directory.
- On Palm OS, when downloading to the device external storage, prefix the dest_path with **vfs:**. The path should then be specified with the platform file naming convention. See “Palm OS” [*UltraLite - Database Management and Reference*].

  If dest_path field is empty, MLFileTransfer assumes it is downloading a Palm record database (*.pdb*).

dest_filename The local name for the downloaded file. If this parameter is empty, the value in file name is used.

stream Required. The protocol can be one of: TCPIP, TLS, HTTP, or HTTPS. See “Stream Type synchronization parameter” [*UltraLite - Database Management and Reference*].

stream_parms The protocol options for a given stream. See “Network protocol options for UltraLite synchronization streams” [*UltraLite - Database Management and Reference*].

username Required. MobiLink user name.

password The password for the MobiLink user name.

version Required. The MobiLink script version.

observer A callback can be provided to observe file download progress through the 'observer' field. For more details, see description of Callback Function that follows.

user_data The application-specific information made available to the synchronization observer. See "User Data synchronization parameter" [*UltraLite - Database Management and Reference*].

force_download Set to true to download the file even if the timestamp indicates it is already present. Set to false to downloaded only if the server version and the local version are different. In this case, the server version of the file overwrites the client version. Any previous file of the same name on the client is discarded before the file is downloaded. MLFileTransfer compares the server and client version of the file by computing a cryptographic hash value for each file; the hash values are identical only if the files are identical in content.

enable_resume If set to true, MLFileTransfer resumes a previous download that was interrupted because of a communications error or because it was canceled by the user. If the file on the server is newer than the partial local file, the partial file is discarded and the new version is downloaded from the beginning. The force_download parameter overrides this parameter.

num_auth_parms The number of authentication parameters being passed to authentication parameters in MobiLink events. See "Number of Authentication Parameters parameter" [*UltraLite - Database Management and Reference*].

auth_parms Supplies parameters to authentication parameters in MobiLink events. See "Authentication Parameters synchronization parameter" [*UltraLite - Database Management and Reference*].

downloaded_file Is set to one of the following:

- 1 if the file was successfully downloaded.
- 0 if an error occurs. An error occurs if the file is already up-to-date when MLFileTransfer is invoked. In this case, the function returns true rather than false. For the Palm OS, when downloading a record database (*.pdb*) file, MLFileTransfer always downloads the file, whether the file is up-to-date or not.

auth_status Reports the status of MobiLink user authentication. The MobiLink server provides this information to the client. See "Authentication Status synchronization parameter" [*UltraLite - Database Management and Reference*].

auth_value Reports results of a custom MobiLink user authentication script. The MobiLink server provides this information to the client. See "Authentication Value synchronization parameter" [*UltraLite - Database Management and Reference*].

file_auth_code Contains the return code of the optional authenticate_file_transfer script on the server.

error Contains information about any error that occurs.

Returns

- **ul_true** The file was successfully downloaded.
- **ul_false** The file was not downloaded successfully. You can supply Error information in the error field of the ml_file_transfer_info structure. Incomplete file transfers are resumable.

Remarks

You must set the source location of the file to be transferred. This location must be specified as a MobiLink user's directory on the MobiLink server (or in the default directory on that server). You can also set the intended target location and file name of the file.

For example, you can program your application to download a new or replacement database from the MobiLink server. You can customize the file for specific users, since the first location that is searched is a specific user's subdirectory. You can also maintain a default file in the root folder on the server, since that location is used if the specified file is not found in the user's folder.

Callback Function

The callback to observe file transfer progress through the observer parameter has the following prototype:

```
typedef void(*ml_file_transfer_observer_fn)( ml_file_transfer_status *
status );
```

The ml_file_transfer_status object passed to the callback is defined as follows:

```
typedef struct ml_file_transfer_status {
    asa_uint64                  file_size;
    asa_uint64                  bytes_received;
    asa_uint64                  resumed_at_size;
    ml_file_transfer_info_a *   info;
    asa_uint16                  flags;
    asa_uint8                   stop;
} ml_file_transfer_status;
```

file_size The total size in bytes of the file being downloaded.

bytes_received Indicates how much of the file has been downloaded so far, including previous synchronizations, if the download is resumed.

resumed_at_size Used with download resumption and indicates at what point the current download resumed.

info Points to the info object passed to MLFileTransfer. You can access the user_data parameter through this pointer.

flags Provides additional information. The value MLFT_STATUS_FLAG_IS_BLOCKING is set when MLFileTransfer is blocking on a network call and the download status has not changed since the last time the observer function was called.

stop May be set to true to cancel the current download. You can resume the download in a subsequent call to MLFileTransfer, but only if you have set the enable_resume parameter.

ULCreateDatabase function

Creates an UltraLite database.

Syntax

ul_bool **ULCreateDatabase (** SQLCA * *sqlca*,
ul_char * *connection-parms*,
void const * *collation*,
ul_char * *creation-parms*,
void * *reserved*
);

Parameters

sqlca A pointer to the initialized SQLCA.

connection-parms A semicolon separated string of connection parameters, which are set as keyword value pairs. The connection string must include the name of the database. These parameters are the same set of parameters that may be specified when you connect to a database. For a complete list, see "UltraLite connection parameters" [*UltraLite - Database Management and Reference*].

collation
The desired collation sequence for the database. You can get a collation sequence by calling the appropriate function. For example:

```
void const * collation = ULGetCollation_1250LATIN2();
```

The function name is constructed by prefixing the name of the desired collation with **ULGetCollation_**. For a list of all available collation functions see *install-dir\SDK\Include\ulgetcoll.h*. You must include this file in programs that call any of the **ULGetCollation_** functions.

creation-parms
A semicolon separated string of database creation parameters, which are set as keyword value pairs. For example:

```
page_size=2048;obfuscate=yes
```

For a complete list, see "Choosing database creation parameters for UltraLite" [*UltraLite - Database Management and Reference*].

reserved This parameter is reserved for future use.

Returns

- **ul_true** Indicates that database was successfully created.
- **ul_false** A detailed error message is defined by the SQLCODE field in the SQLCA. Typically this is caused by an invalid file name or access denial.

Remarks

The database is created with information provided in two sets of parameters:

- The connection-parms are standard connection parameters that are applicable whenever the database is accessed (for example, file name, user id, password, optional encryption key, and so on).
- The creation-parms are parameters that are only relevant when creating a database (for example, obfuscation, page-size, time and date format, and so one)

Applications can call this function after initializing the SQLCA.

Example

The following code illustrates using ULCreateDatabase to create an UltraLite database as the file *C:\myfile.udb*.

```
if( ULCreateDatabase(&sqlca
    ,UL_TEXT("DBF=C:\myfile.udb;uid=DBA;pwd=sql")
    ,ULGetCollation_1250LATIN2()
    ,UL_TEXT("obfuscate=1;page_size=8192")
    ,NULL)
{
    // success
};
```

ULEnableEccSyncEncryption function

Enables ECC encryption for SSL or TLS streams. This is required when set a stream parameter to TLS or HTTPS. In this case, you must also set the synchronization parameter tls_type as ECC.

Syntax

void **ULEnableEccSyncEncryption(** SQLCA * *sqlca* **);**

Parameters

sqlca A pointer to the initialized SQLCA.

In the C++ API use the Sqlca.GetCA method.

Remarks

You can use this function in C++ API applications and embedded SQL applications. You must call this function before the Synchronize function. If you attempt to synchronize without a preceding call to enable the synchronization type, the error SQLE_METHOD_CANNOT_BE_CALLED occurs.

Separately licensed component required
ECC encryption and FIPS-certified encryption require a separate license. All strong encryption technologies are subject to export regulations.

See "Separately licensed components" [*SQL Anywhere 11 - Introduction*].

See also

- "ULEnableZlibSyncCompression function" on page 117
- "ULEnableRsaFipsSyncEncryption function" on page 112

ULEnableFIPSStrongEncryption function

Enables FIPS-based strong encryption for the database. Calling this function causes the appropriate encryption routines to be included in the application and increases the size of the application code.

Syntax

void **ULEnableFIPSStrongEncryption(** SQLCA * *sqlca* **);**

Parameters

sqlca A pointer to the initialized SQLCA.

In the C++ API use the Sqlca.GetCA method.

Remarks

You can use this function in C++ API applications and embedded SQL applications. You must call this function before the Synchronize function. If you attempt to synchronize without a preceding call to enable the synchronization type, the error SQLE_METHOD_CANNOT_BE_CALLED occurs.

Separately licensed component required
ECC encryption and FIPS-certified encryption require a separate license. All strong encryption technologies are subject to export regulations.

See "Separately licensed components" [*SQL Anywhere 11 - Introduction*].

See also

- "ULEnableStrongEncryption function" on page 114

ULEnableHttpSynchronization function

Enables HTTP synchronization.

Syntax

void **ULEnableHttpSynchronization(** SQLCA * *sqlca* **);**

Parameters

sqlca A pointer to the initialized SQLCA.

In the C++ API use the Sqlca.GetCA method.

Remarks

You can use this function in C++ API applications and embedded SQL applications. You must call this function before the Synchronize function. If you attempt to synchronize without a preceding call to enable the synchronization type, the error SQLE_METHOD_CANNOT_BE_CALLED occurs.

ULEnableHttpsSynchronization function

Enables the SSL synchronization stream for HTTP.

Syntax

void **ULEnableHttpsSynchronization(** SQLCA * *sqlca* **);**

Parameters

sqlca A pointer to the initialized SQLCA.

In the C++ API use the Sqlca.GetCA method.

Remarks

You can use this function in C++ API applications and embedded SQL applications. You must call this function before the Synchronize function. If you attempt to synchronize without a preceding call to enable the synchronization type, the error SQLE_METHOD_CANNOT_BE_CALLED occurs.

Example

```
ULEnableHttpsSynchronization( sqlca );
ULEnableRsaSyncEncryption( sqlca );
synch_info.stream = "https";
synch_info.stream_parms = "tls_type=rsa";    // rsa is default, so setting
this parameter is optional
conn->Synchronize( sqlca );
```

ULEnableRsaFipsSyncEncryption function

Enables RSA FIPS encryption for SSL or TLS streams. This is required when setting a stream parameter to TLS or HTTPS. In this case, you must also set the synchronization parameter tls_type as RSA.

Syntax

void **ULEnableRsaFipsSyncEncryption(** SQLCA * *sqlca* **);**

Parameters

sqlca A pointer to the initialized SQLCA.

In the C++ API use the Sqlca.GetCA method.

Remarks

You can use this function in C++ API applications and embedded SQL applications. You must call this function before the Synchronize function. If you attempt to synchronize without a preceding call to enable the synchronization type, the error SQLE_METHOD_CANNOT_BE_CALLED occurs.

See also

- "ULEnableRsaSyncEncryption function" on page 113
- "ULEnableEccSyncEncryption function" on page 108

ULEnableRsaSyncEncryption function

Enables RSA encryption for SSL or TLS streams. This is required when setting a stream parameter to TLS or HTTPS. In this case, you must also set the synchronization parameter tls_type as RSA.

Syntax

void **ULEnableRsaSyncEncryption(** SQLCA * *sqlca* **);**

Parameters

sqlca A pointer to the initialized SQLCA.

In the C++ API use the Sqlca.GetCA method.

Remarks

You can use this function in C++ API applications and embedded SQL applications. You must call this function before the Synchronize function. If you attempt to synchronize without a preceding call to enable the synchronization type, the error SQLE_METHOD_CANNOT_BE_CALLED occurs.

See also

- "ULEnableEccSyncEncryption function" on page 108
- "ULEnableRsaFipsSyncEncryption function" on page 112

ULEnableStrongEncryption function

Enables strong encryption.

Syntax

void **ULEnableStrongEncryption(** SQLCA * *sqlca* **);**

Parameters

sqlca A pointer to the initialized SQLCA.

In the C++ API use the Sqlca.GetCA method.

Remarks

You can use this function in C++ API applications and embedded SQL applications. You must call this function before db_init or ULInitDatabaseManager.

Note
Calling this function causes the encryption routines to be included in the application and increases the size of the application code.

ULEnableTcpipSynchronization function

Enables TCP/IP synchronization.

Syntax

void **ULEnableTcpipSynchronization(** SQLCA * *sqlca* **);**

Parameters

sqlca A pointer to the initialized SQLCA.

In the C++ API use the Sqlca.GetCA method.

Remarks

You can use this function in C++ API applications and embedded SQL applications. You must call this function before the Synchronize function. If you attempt to synchronize without a preceding call to enable the synchronization type, the error SQLE_METHOD_CANNOT_BE_CALLED occurs.

ULEnableTlsSynchronization function

Enables TLS synchronization.

Syntax

void **ULEnableTlsSynchronization(** SQLCA * *sqlca* **);**

Parameters

sqlca A pointer to the initialized SQLCA.

In the C++ API use the Sqlca.GetCA method.

Remarks

You can use this function in C++ API applications and embedded SQL applications. You must call this function before the Synchronize function. If you attempt to synchronize without a preceding call to enable the synchronization type, the error SQLE_METHOD_CANNOT_BE_CALLED occurs.

ULEnableZlibSyncCompression function

Enables ZLIB compression for a synchronization stream.

Syntax

void **ULEnableZlibSyncCompression(** SQLCA * *sqlca* **);**

Parameters

sqlca A pointer to the initialized SQLCA.

In the C++ API use the Sqlca.GetCA method.

Remarks

You can use this function in C++ API applications and embedded SQL applications. You must call this function before calling the Synchronize function. If you attempt to synchronize without a preceding call to enable the synchronization type, the error SQLE_METHOD_CANNOT_BE_CALLED occurs.

ULInitDatabaseManager

Initializes the UltraLite database manager.

Syntax

pointer **ULInitDatabaseManager(** SQLCA * *sqlca* **);**

Parameters

sqlca A pointer to the initialized SQLCA.

In the C++ API use the Sqlca.GetCA method.

Returns

- A pointer to the database manager.
- NULL if the function fails.

Remarks

The function fails if you have not previously initialized the Database Manager and did not issue a Shutdown.

ULInitDatabaseManagerNoSQL

Initialize the UltraLite database manager and exclude support for processing SQL statements (this can significantly reduce the application run time size).

Syntax

pointer **ULInitDatabaseManagerNoSQL(** SQLCA * *sqlca* **);**

Parameters

sqlca   A pointer to the initialized SQLCA.

In the C++ API use the Sqlca.GetCA method.

Returns

- A pointer to the database manager.
- NULL if the function fails.

Remarks

The function fails if you have not previously initialized the Database Manager and did not issue a Shutdown.

The application must access data through the Table API and cannot use SQL statements. You cannot use the call if the database schema contains publication predicates; use ULInitDatabaseManager instead.

ULRegisterErrorCallback function

Registers a callback function that handles errors.

Syntax

void **ULRegisterErrorCallback (**
   SQLCA * *sqlca*,
   ul_error_callback_fn *callback*,
   ul_void *   *user_data*,
   ul_char *   *buffer*,
   size_t   *len*
);

Parameters

- **sqlca**   A pointer to the SQL Communications Area.

  In the C++ API use the Sqlca.GetCA method.

- **callback**   The name of your callback function. For more information about the prototype of the function, see "Callback function for ULRegisterErrorCallback" on page 100.

  A callback value of UL_NULL disables any previously registered callback function.

- **user_data**   An alternative to global variables to make any context information globally accessible. This is required because you can call the callback function from any location in your application. UltraLite does not modify the supplied data; it simply passes it to your callback function when it is invoked.

  You can declare any data type and cast it into the correct type in your callback function. For example, you could include a line of the following form in your callback function:

  ```
  MyContextType * context = (MyContextType *)user_data;
  ```

- **buffer**   A character array holding the substitution parameters for the error message, including a null terminator. To keep UltraLite as small as possible, UltraLite does not supply error messages. The substitution parameters depend on the specific error. For a complete list, see "SQL Anywhere error messages" [*Error Messages*].

  The buffer must exist as long as UltraLite is active. Supply UL_NULL if you do not want to receive parameter information.

- **len**   The length of the buffer (preceding parameter), in ul_char characters. A value of 100 is large enough to hold most error parameters. If the buffer is too small, the parameters are truncated.

Remarks

Once you call this function, the user-supplied callback function is called whenever UltraLite signals an error. You should therefore call ULRegisterErrorCallback immediately after initializing the SQLCA.

Error handling with this callback technique is particularly helpful during development, as it ensures that your application is notified of any and all errors that occur. However, the callback function does not control execution flow, so the application should check the SQLCODE field in the SQLCA after all calls to UltraLite functions.

Example

The following code registers a callback function for an UltraLite C++ Component application:

```
int main() {
    ul_char buffer[100];
    DatabaseManager * dm;
    Connection * conn;
    Sqlca.Initialize();
    ULRegisterErrorCallback(
        Sqlca.GetCA(),
        MyErrorCallBack,
        UL_NULL,
        buffer,
        sizeof (buffer) );
    dm = ULInitDatabaseManager( Sqlca );
    ...
}
```

The following is a sample callback function:

```
ul_error_action UL_GENNED_FN_MOD MyErrorCallBack(
    SQLCA *      Sqlca,
    ul_void *    user_data,
    ul_char *    message_param )
{
    ul_error_action rc = 0;
    (void) user_data;

    switch( Sqlca->sqlcode ) {
        // The following error is used for flow control - don't report it
here
        case SQLE_NOTFOUND:
            break;
        case SQLE_ULTRALITE_DATABASE_NOT_FOUND:
          _tprintf( _TEXT( "Error %ld: Database file %s not found\n" ), Sqlca-
>sqlcode, message_param );
            break;
        default:
            _tprintf( _TEXT( "Error %ld: %s\n" ), Sqlca->sqlcode,
message_param );
        break;
     }
    return rc;
}
```

See also

- “SQL Anywhere error messages sorted by Sybase error code” [*Error Messages*]
- “Callback function for ULRegisterErrorCallback” on page 100

ULRegisterSQLPassthroughCallback

Registers a callback function that provides ongoing status.

Syntax

void **ULRegisterSQLPassthroughCallback (**
SQLCA * *sqlca*,
ul_sql_passthrough_observer_fn*callback*,
ul_void * *user_data*
);

Parameters

- **sqlca** A pointer to the SQL Communications Area.

  In the C++ API use the Sqlca.GetCA method.

- **callback** The name of your callback function.

  A callback value of UL_NULL disables any previously registered callback function.

- **user_data** An alternative to global variables to make any context information globally accessible. This is required because you can call the callback function from any location in your application. UltraLite does not modify the supplied data; it simply passes it to your callback function when it is invoked.

  You can declare any data type and cast it into the correct type in your callback function. For example, you could include a line of the following form in your callback function:

  ```
  MyContextType * context = (MyContextType *) user_data;
  ```

Example

Here is a sample callback and code to invoke ULRegisterSQLPassthroughCallback:

```
static void UL_GENNED_FN_MOD passthroughCallback( ul_sql_passthrough_status *
status ) {
    switch( status->state ) {
        case UL_SQL_PASSTHROUGH_STATE_STARTING:
            printf( "SQL Passthrough script execution starting\n" );
            break;
        case UL_SQL_PASSTHROUGH_STATE_RUNNING_SCRIPT:
           printf( "Executing script %d of %d\n", status->cur_script, status-
>script_count );
            break;
        case UL_SQL_PASSTHROUGH_STATE_DONE:
            printf( "Finished executing SQL Passthrough scripts\n" );
            break;
        default:
            printf( "SQL Passthrough script execution has failed\n" );
            break;
    }
}

int main() {
    ULSqlca sqlca;

    sqlca.Initialize();
    ULRegisterSQLPassthroughCallback( sqlca.GetCA(), passthroughCallback,
```

```
NULL );
    DatabaseManager * dm = ULInitDatabaseManager( sqlca );
    ...
}
```

ULRegisterSynchronizationCallback

Registers a function to be called when synchronization is executed via the SQL SYNCHRONIZE statement. If a synchronization callback function is defined and registered with UltraLite, whenever a SYNCHRONIZE statement is executed, progress information for that synchronization is passed to the callback function. If no callback is registered, progress information is suppressed.

Syntax

void **ULRegisterSynchronizationCallback (**
SQLCA * *sqlca*,
ul_synch_observer_fn*callback*,
ul_void * *user_data*
);

Parameters

- **sqlca** A pointer to the SQL Communications Area.

  In the C++ API use the Sqlca.GetCA method.

- **callback** The name of your callback function.

  A callback value of UL_NULL disables any previously registered callback function.

- **user_data** An alternative to global variables to make any context information globally accessible. This is required because you can call the callback function from any location in your application. UltraLite does not modify the supplied data; it simply passes it to your callback function when it is invoked.

  You can declare any data type and cast it into the correct type in your callback function. For example, you could include a line of the following form in your callback function:

  ```
  MyContextType * context = (MyContextType *) user_data;
  ```

See also

- "UltraLite SYNCHRONIZE statement" [*UltraLite - Database Management and Reference*]

Macros and compiler directives for UltraLite C/C++ applications

Unless otherwise stated otherwise, directives apply to both embedded SQL and C++ API applications.

You can supply compiler directives:

- On your compiler command line. You commonly set a directive with the /D option. For example, to compile an UltraLite application with user authentication, a makefile for the Microsoft Visual C++ compiler may look as follows:

  ```
  CompileOptions=/c /DPRWIN32 /Od /Zi /DWIN32
  /DUL_USE_DLL

  IncludeFolders= \
  /I"$(VCDIR)\include" \
  /I"$(SQLANY11)\SDK\Include"

  sample.obj: sample.cpp
   cl $(CompileOptions) $(IncludeFolders) sample.cpp
  ```

  VCDIR is your Visual C++ directory and *SQLANY11* is your SQL Anywhere installation directory.

- In the compiler settings window of your user interface.
- In source code. You supply directives with the `#define` statement.

UL_AS_SYNCHRONIZE macro

Provides the name of the callback message used to indicate an ActiveSync synchronization.

Remarks

Applies to Windows Mobile applications using ActiveSync only.

See also

- “Adding ActiveSync synchronization to your application” on page 92

UL_SYNC_ALL macro

Provides a publication list string that refers to all tables in the database, including those not explicitly referred to by a publication. Tables marked as "no sync" are not including in this list.

See also

- “ul_synch_info_a struct” on page 133
- “ul_synch_info_w2 struct” on page 136
- “ULGetLastDownloadTime function” on page 276
- “ULCountUploadRows function” on page 268
- “UL_SYNC_ALL_PUBS macro” on page 126

UL_SYNC_ALL_PUBS macro

Provides a publication list that refers to all tables in the database that are referenced in a publication.

See also

- “ul_synch_info_a struct” on page 133
- “ul_synch_info_w2 struct” on page 136
- “ULGetLastDownloadTime function” on page 276
- “ULCountUploadRows function” on page 268
- “UL_SYNC_ALL macro” on page 125

UL_TEXT macro

Prepares constant strings to be compiled as single-byte strings or wide-character strings. Use this macro to enclose all constant strings if you plan to compile the application to use Unicode and non-Unicode representations of strings. This macro properly defines strings in all environments and platforms.

UL_USE_DLL macro

Sets the application to use the runtime library DLL, rather than a static runtime library.

Remarks

Applies to Windows Mobile and Windows applications.

UNDER_CE macro

By default, this macro is defined in all new embedded Visual C++ projects by the Microsoft embedded Visual C++ compiler.

Remarks

Applies to Windows Mobile applications.

Example

```
/D UNDER_CE=$(CEVersion)
```

See also

- "Developing UltraLite applications for Windows Mobile" on page 83

UNDER_PALM_OS macro

This macro is defined in the *ulpalmos.h* header file included in UltraLite Palm OS applications by the UltraLite plug-in. See "Using the UltraLite plug-in for CodeWarrior" on page 71.

Remarks

Applies to a compiler directive for Palm OS only.

See also

- "Developing UltraLite applications for the Palm OS" on page 67

UltraLite C++ API reference

Contents

ul_sql_passthrough_status struct

SQL Passthrough status information.

Syntax

public **ul_sql_passthrough_status**

Properties

Name	Type	Description
cur_script	ul_u_long	The current script being executed. (1-based)
script_count	ul_u_long	The total number of scripts that will be executed.
sqlca	SQLCA *	A pointer to the SQL Communications Area. See "GetCA function" on page 147.
state	ul_sql_passthrough_state	The current state. See "Handling synchronization status information" on page 59.
stop	ul_bool	Set to true to stop script execution.
user_data	ul_void *	The user data provided in the register call. See "ULRegisterSQLPassthroughCallback" on page 122.

ul_stream_error struct

Synchronization communication stream error information.

Syntax

public **ul_stream_error**

Properties

Name	Type	Description
error_string	char	Array for the stream_error_code field.
stream_error_code	ss_error_code	Not required. The value is always 0.
system_error_code	asa_int32	A system-specific error code. For more information about error codes, consult your platform documentation. For Windows platforms, See Microsoft Developer Network web site.

Remarks

The following are common system errors on Windows:

- 10048 (WSAADDRINUSE) Address already in use.
- 10053 (WSAECONNABORTED) Software caused connection abort.
- 10054 (WSAECONNRESET) The other side of the communication closed the socket.
- 10060 (WSAETIMEDOUT) Connection timed out.
- 10061 (WSAECONNREFUSED) Connection refused. Typically, this means that the MobiLink server is not running or is not listening on the specified port. See the Microsoft Developer Network web site.

ul_synch_info_a struct

The structure used to describe synchronization data.

Syntax

public **ul_synch_info_a**

Properties

Name	Type	Description
addition-al_parms	const char *	A string of additional synchronization parameters coded as a semicolon delimited list of keyword=value pairs. This field usually contains synchronization parameters that are infrequently used. See "Additional Parameters synchronization parameter" [*UltraLite - Database Management and Reference*].
auth_parms	char **	An array of authentication parameters in MobiLink events.
auth_status	ul_auth_status	The status of MobiLink user authentication. The MobiLink server provides this information to the client.
auth_value	ul_s_long	The results of a custom MobiLink user authentication script. The MobiLink server provides this information to the client to determine the authentication status.
download_only	ul_bool	Do not upload any changes from the UltraLite database during the current synchronization.
ignored_rows	ul_bool	The status of ignored rows. This read-only field reports true if any rows were ignored by the MobiLink server during synchronization because of absent scripts.
init_verify	ul_synch_info_a *	Initialize verification.
keep_parti-al_download	ul_bool	When a download fails because of a communications error during synchronization, this parameter controls whether UltraLite holds on to the partial download rather than rolling back the changes.
new_password	char *	A string specifying a new MobiLink password associated with the user name. This parameter is optional.
num_auth_parms	ul_byte	The number of authentication parameters being passed to authentication parameters in MobiLink events.
observer	ul_synch_observer_fn	A pointer to a callback function or event handler that monitors synchronization. This parameter is optional.

Name	Type	Description
partial_download_retained	ul_bool	When a download fails because of a communications error during synchronization, this parameter indicates whether UltraLite applied those changes that were downloaded rather than rolling back the changes.
password	char *	A string specifying the existing MobiLink password associated with the user name. This parameter is optional.
ping	ul_bool	Confirm communications between the UltraLite client and the MobiLink server. When this parameter is set to true, no synchronization takes place.
publications	const char *	A comma separated list of publications indicating what data to include in the synchronization.
resume_partial_download	ul_bool	Resume a failed download. The synchronization does not upload changes; it only downloads those changes that were to be downloaded in the failed download.
send_column_names	ul_bool	Instructs the application that column names should be sent to the MobiLink server in the upload.
send_download_ack	ul_bool	Instructs the MobiLink server whether the client will provide a download acknowledgement.
stream	const char *	The MobiLink network protocol to use for synchronization.
stream_error	ul_stream_error	The structure to hold communications error reporting information.
stream_parms	char *	The options to configure the network protocol you selected.
upload_ok	ul_bool	The status of data uploaded to the MobiLink server. This field reports true if upload succeeded.
upload_only	ul_bool	Do not download any changes from the consolidated database during the current synchronization. This can save communication time, especially over slow communication links.
user_data	ul_void *	Make application-specific information available to the synchronization observer. This parameter is optional.
user_name	char *	A string that the MobiLink server uses to identify a unique MobiLink user.
version	char *	The version string allows an UltraLite application to choose from a set of synchronization scripts.

Remarks

Synchronization parameters control the synchronization behavior between an UltraLite database and the MobiLink server. The Stream Type synchronization parameter, User Name synchronization parameter, and Version synchronization parameter are required. If you do not set them, the synchronization function returns an error (SQLE_SYNC_INFO_INVALID or its equivalent). You can only specify one of Download Only, Ping, or Upload Only at a time. If you set more than one of these parameters to true, the synchronization function returns an error (SQLE_SYNC_INFO_INVALID or its equivalent).

ul_synch_info_w2 struct

The wide character structure used to describe synchronization.

Syntax

public **ul_synch_info_w2**

Properties

Name	Type	Description
addition-al_parms	const ul_wchar *	A string of additional synchronization parameters coded as a semicolon delimited list of keyword=value pairs. This field usually contains synchronization parameters that are infrequently used. See “Additional Parameters synchronization parameter” [*UltraLite - Database Management and Reference*].
auth_parms	ul_wchar **	An array of authentication parameters in MobiLink events.
auth_status	ul_auth_sta tus	The status of MobiLink user authentication. The MobiLink server provides this information to the client.
auth_value	ul_s_long	The MobiLink server provides this information to the client to determine the authentication status.
download_on-ly	ul_bool	Do not upload any changes from the UltraLite database during the current synchronization.
ignored_rows	ul_bool	The status of ignored rows. This read-only field reports true if any rows were ignored by the MobiLink server during synchronization because of absent scripts.
init_verify	ul_synch_i nfo_w2 *	Initialize verification.
keep_parti-al_download	ul_bool	When a download fails because of a communications error during synchronization, this parameter controls whether UltraLite holds on to the partial download rather than rolling back the changes.
new_pass-word	ul_wchar *	A string specifying a new MobiLink password associated with the user name. This parameter is optional.
num_auth_par ms	ul_byte	The number of authentication parameters being passed to authentication parameters in MobiLink events.
observer	ul_synch_o bserver_fn	A pointer to a callback function or event handler that monitors synchronization. This parameter is optional.

Name	Type	Description
partial_down-load_retained	ul_bool	When a download fails because of a communications error during synchronization, this parameter indicates whether UltraLite applied those changes that were downloaded rather than rolling back the changes.
password	ul_wchar *	A string specifying the existing MobiLink password associated with the user name. This parameter is optional.
ping	ul_bool	Confirm communications between the UltraLite client and the MobiLink server. When this parameter is set to true, no synchronization takes place.
publications	const ul_wchar *	A comma separated list of publications indicating what data to include in the synchronization.
resume_parti-al_download	ul_bool	Resume a failed download. The synchronization does not upload changes; it only downloads those changes that were to be downloaded in the failed download.
send_col-umn_names	ul_bool	Instructs the application that column names should be sent to the MobiLink server in the upload.
send_down-load_ack	ul_bool	Instructs the MobiLink server whether the client will provide a download acknowledgement.
stream	const char *	The MobiLink network protocol to use for synchronization.
stream_error	ul_stream_ error	The structure to hold communications error reporting information.
stream_parms	ul_wchar *	The options to configure the network protocol you selected.
upload_ok	ul_bool	The status of data uploaded to the MobiLink server. This field reports true if upload succeeded.
upload_only	ul_bool	Do not download any changes from the consolidated database during the current synchronization. This can save communication time, especially over slow communication links.
user_data	ul_void *	Make application-specific information available to the synchronization observer. This parameter is optional.
user_name	ul_wchar *	A string that the MobiLink server uses to identify a unique MobiLink user.
version	ul_wchar *	The version string allows an UltraLite application to choose from a set of synchronization scripts.

Remarks

Synchronization parameters control the synchronization behavior between an UltraLite database and the MobiLink server. The Stream Type synchronization parameter, User Name synchronization parameter, and Version synchronization parameter are required. If you do not set them, the synchronization function returns an error (SQLE_SYNC_INFO_INVALID or its equivalent). You can only specify one of Download Only, Ping, or Upload Only at a time. If you set more than one of these parameters to true, the synchronization function returns an error (SQLE_SYNC_INFO_INVALID or its equivalent). See "ul_synch_info_a struct" on page 133.

ul_synch_result struct

A structure to hold the synchronization result, so that appropriate action can be taken in the application.

Syntax

public **ul_synch_result**

Properties

Name	Type	Description
auth_status	ul_auth_status	The synchronization authentication status.
auth_value	ul_s_long	The value used by the MobiLink server to determine the auth_status result.
ignored_rows	ul_bool	Set to true if uploaded rows were ignored; false otherwise.
partial_download_retained	ul_bool	The value that tells you that a partial download was retained. See keep_partial_download.
sql_code	an_sql_code	The SQL code from the last synchronization.
sql_error_string	char	The error text associated with the error code in sql_code.
status	ul_synch_status	The status information used by the observer function. See observer.
stream_error	ul_stream_error	The communication stream error information.
timestamp	SQLDATETIME	The time and date of the last synchronization.
upload_ok	ul_bool	Set to true if the upload was successful; false otherwise.

ul_synch_stats struct

Reports the statistics of the synchronization stream.

Syntax

public **ul_synch_stats**

Properties

Name	Type	Description
bytes	ul_u_long	The number of bytes currently sent.
deletes	ul_u_long	The number of deleted rows current sent.
inserts	ul_u_long	The number of rows currently inserted.
updates	ul_u_long	The number of updated rows currently sent.

ul_synch_status struct

Returns synchronization progress monitoring data.

Syntax

public **ul_synch_status**

Properties

Name	Type	Description
db_table_count	ul_u_short	Returns the number of tables in database.
flags	ul_u_short	Returns the current synchronization flags indicating additional information relating to the current state.
info	ul_synch_info_a *	A pointer to the ul_synch_info_a structure. See "ul_synch_info_a struct" on page 133.
received	ul_synch_stats	Returns download statistics.
sent	ul_synch_stats	Returns upload statistics.
sqlca	SQLCA *	The connection's active SQLCA.
state	ul_synch_state	One of the many supported states. See "Handling synchronization status information" on page 59.
stop	ul_bool	A boolean that cancel synchronization. A value of true means that synchronization is canceled.
sync_table_count	ul_u_short	Returns the number of tables being synchronized.
sync_table_index	ul_u_short	Values range from 1 to the number specified in sync_table_count.
table_id	ul_u_short	The current table id which is being uploaded or downloaded (1-based). This number may skip values when not all tables are being synchronized, and is not necessarily increasing.
table_name	char	Name of the current table.

Name	Type	Description
ta-ble_name_w2	char	Name of the current table (wide character format). This field is only populated on Windows desktop and Mobile platforms.
user_data	ul_void *	User data passed in to ULRegisterSynchronizationCallback or set in the ul_synch_info structure.

ul_validate_data struct

Validation status information.

Syntax

public **ul_validate_data**

Properties

Name	Type	Description
I	ul_u_long	Parameter as an integer.
parm_count	ul_u_short	The number of parameters in the structure.
parm_type	enum	Indicates the type of each parameter in the parameter array (integer or string).
parms	struct ul_validate_data::@12	Array of parameters.
s	char	Parameter as a string (note that this is not a wide char).
status_id	ul_validate_status_id	Indicates what is being reported in the validation process.
stop	ul_bool	A boolean that cancels the validation. A value of true means that validation is canceled.
type	parm_type	Type of parameter stored.
user_data	ul_void *	User-defined data pointer passed into validation routine.

ULSqlca class

The "ULSqlcaBase class" on page 146 contains a SQLCA structure, so an external one is not required.

Syntax

public **ULSqlca**

Base classes

- "ULSqlcaBase class" on page 146

Members

All members of ULSqlca, including all inherited members.

- "Finalize function" on page 146
- "GetCA function" on page 147
- "GetParameter function" on page 147
- "GetParameter function" on page 147
- "GetParameterCount function" on page 148
- "GetSQLCode function" on page 148
- "GetSQLCount function" on page 148
- "GetSQLErrorOffset function" on page 149
- "Initialize function" on page 149
- "LastCodeOK function" on page 149
- "LastFetchOK function" on page 150
- "ULSqlca function" on page 144
- "~ULSqlca function" on page 144

Remarks

This class is used in most C++ component applications. You must initialize the SQLCA before you call any other functions. Each thread requires its own SQLCA.

ULSqlca function

This function is the SQLCA constructor.

Syntax

ULSqlca::ULSqlca()

~ULSqlca function

This function is the SQLCA destructor.

Syntax

ULSqlca::~ULSqlca()

ULSqlcaBase class

Defines the communication area between the interface library and the application.

Syntax

public **ULSqlcaBase**

Derived classes

- "ULSqlca class" on page 144
- "ULSqlcaWrap class" on page 151

Members

All members of ULSqlcaBase, including all inherited members.

- "Finalize function" on page 146
- "GetCA function" on page 147
- "GetParameter function" on page 147
- "GetParameter function" on page 147
- "GetParameterCount function" on page 148
- "GetSQLCode function" on page 148
- "GetSQLCount function" on page 148
- "GetSQLErrorOffset function" on page 149
- "Initialize function" on page 149
- "LastCodeOK function" on page 149
- "LastFetchOK function" on page 150

Remarks

Use a subclass of this class (which is typically the "ULSqlca class" on page 144) to create your communication area. This API always requires an underlying SQLCA object. You must initialize the SQLCA before you call any other functions. Each thread requires its own SQLCA.

Finalize function

Finalizes this SQLCA.

Syntax

void **ULSqlcaBase::Finalize()**

Remarks

Until you initialize the communication area again, you cannot use it.

GetCA function

Gets the SQLCA structure for direct access to additional fields.

Syntax

SQLCA * **ULSqlcaBase::GetCA()**

Returns

A raw SQLCA structure.

GetParameter function

Gets the error parameter string.

Syntax

size_t **ULSqlcaBase::GetParameter(**
  ul_u_long *parm_num*,
  char * *buffer*,
  size_t *size*
)

Parameters

- **parm_num** A 1-based parameter number.
- **buffer** The buffer to receive parameter string.
- **size** The size, in chars, of the buffer.

Returns

- If the function succeeds, the return value is the buffer size required to hold the entire parameter string.
- If the function fails, the return value is zero. The function fails if an invalid (out of range) parameter number is given.

Remarks

The output parameter string is always null-terminated, even if the buffer is too small and the parameter is truncated. The parameter number is 1-based.

GetParameter function

Gets the error parameter string.

Syntax

size_t **ULSqlcaBase::GetParameter(**
  ul_u_long *parm_num*,
  ul_wchar * *buffer*,

size_t *size*
)

Parameters

- **parm_num** A 1-based parameter number.
- **buffer** The buffer to receive parameter string.
- **size** The size, in ul_wchars, of the buffer.

Returns

- If the function succeeds, the return value is the buffer size required to hold the entire parameter string.
- If the function fails, the return value is zero. The function fails if an invalid (out of range) parameter number is given.

Remarks

The output parameter string is always null-terminated, even if the buffer is too small and the parameter is truncated. The parameter number is 1-based.

GetParameterCount function

Gets the error parameter count for last operation.

Syntax

ul_u_long **ULSqlcaBase::GetParameterCount()**

Returns

The number of parameters for the current error.

GetSQLCode function

Gets the error code (SQLCODE) for last operation.

Syntax

an_sql_code **ULSqlcaBase::GetSQLCode()**

Returns

The sqlcode value.

GetSQLCount function

Gets the sql count variable (SQLCOUNT) for the last operation.

Syntax

an_sql_code **ULSqlcaBase::GetSQLCount()**

Returns

The number of rows affected by an INSERT, DELETE, or UPDATE operation. 0 if none are affected.

GetSQLErrorOffset function

Gets the error offset in dynamic SQL statement.

Syntax

ul_s_long **ULSqlcaBase::GetSQLErrorOffset()**

Returns

- When applicable, the return value is the offset into the associated dynamic SQL statement (which is then passed to the PrepareStatement function) that corresponds to the current error.
- When not applicable, the return value is -1.

Initialize function

Initializes this SQLCA.

Syntax

bool **ULSqlcaBase::Initialize()**

Returns

- True if the SQLCA was initialized.
- False if the SQLCA failed to initialize. This method can fail if basic interface library initialization fails. Library failure occurs if system resources are depleted.

Remarks

You must initialize the SQLCA before any other operations occur.

LastCodeOK function

Tests the error code for the last operation.

Syntax

bool **ULSqlcaBase::LastCodeOK()**

Returns

- TRUE if the sqlcode is SQLE_NOERROR or a warning.
- FALSE if sqlcode indicates an error.

LastFetchOK function

Tests the error code for last fetch operation.

Syntax

bool **ULSqlcaBase::LastFetchOK()**

Returns

- TRUE if the sqlcode indicates that a row was fetched successfully by the last operation.
- FALSE if the sqlcode indicates the row was not fetched.

Remarks

Use this function only immediately after performing a fetch operation.

ULSqlcaWrap class

The “ULSqlcaBase class” on page 146 which attaches to an existing SQLCA object.

Syntax

public **ULSqlcaWrap**

Base classes

- “ULSqlcaBase class” on page 146

Members

All members of ULSqlcaWrap, including all inherited members.

- “Finalize function” on page 146
- “GetCA function” on page 147
- “GetParameter function” on page 147
- “GetParameter function” on page 147
- “GetParameterCount function” on page 148
- “GetSQLCode function” on page 148
- “GetSQLCount function” on page 148
- “GetSQLErrorOffset function” on page 149
- “Initialize function” on page 149
- “LastCodeOK function” on page 149
- “LastFetchOK function” on page 150
- “ULSqlcaWrap function” on page 151
- “~ULSqlcaWrap function” on page 152

Remarks

This can be used with a previously-initialized SQLCA object (in which case, you would not call the Initialize function again). You must initialized the SQLCA before you call any other functions. Each thread requires its own SQLCA.

ULSqlcaWrap function

The constructor.

Syntax

```
 ULSqlcaWrap::ULSqlcaWrap(
  SQLCA * sqlca
)
```

Parameters

- **sqlca**   The SQLCA object to use.

Remarks

You can initialize the given SQLCA object before creating this object. In this case, do not call "Initialize function" on page 149 again.

~ULSqlcaWrap function

The destructor.

Syntax

ULSqlcaWrap::~ULSqlcaWrap()

UltraLite_Connection class

Represents a connection to an UltraLite database.

Syntax

public **UltraLite_Connection**

Base classes

- "UltraLite_SQLObject_iface class" on page 215
- "UltraLite_Connection_iface class" on page 156

Members

All members of UltraLite_Connection, including all inherited members.

- "AddRef function" on page 215
- "CancelGetNotification function" on page 158
- "ChangeEncryptionKey function" on page 158
- "Checkpoint function" on page 159
- "Commit function" on page 159
- "CountUploadRows function" on page 159
- "CreateNotificationQueue function" on page 159
- "DeclareEvent function" on page 160
- "DestroyNotificationQueue function" on page 161
- "ExecuteNextSQLPassthroughScript function" on page 161
- "ExecuteSQLPassthroughScripts function" on page 162
- "GetConnection function" on page 215
- "GetConnectionNum function" on page 162
- "GetDatabaseID function" on page 162
- "GetDatabaseProperty function" on page 162
- "GetDatabaseProperty function" on page 163
- "GetIFace function" on page 216
- "GetLastDownloadTime function" on page 163
- "GetLastIdentity function" on page 163
- "GetNewUUID function" on page 164
- "GetNewUUID function" on page 164
- "GetNotification function" on page 164
- "GetNotificationParameter function" on page 165
- "GetSchema function" on page 166
- "GetSqlca function" on page 166
- "GetSQLPassthroughScriptCount function" on page 166
- "GetSuspend function (deprecated)" on page 166
- "GetSynchResult function" on page 166
- "GetUtilityULValue function" on page 167
- "GlobalAutoincUsage function" on page 167
- "GrantConnectTo function" on page 167
- "InitSynchInfo function" on page 168
- "InitSynchInfo function" on page 168
- "OpenTable function" on page 168
- "OpenTableEx function" on page 169
- "OpenTableWithIndex function" on page 169
- "PrepareStatement function" on page 170
- "RegisterForEvent function" on page 170
- "Release function" on page 216
- "ResetLastDownloadTime function" on page 171
- "RevokeConnectFrom function" on page 171
- "Rollback function" on page 172
- "RollbackPartialDownload function" on page 172
- "SendNotification function" on page 172

- “SetDatabaseID function” on page 173
- “SetDatabaseOption function” on page 173
- “SetDatabaseOption function” on page 173
- “SetSuspend function (deprecated)” on page 174
- “SetSynchInfo function” on page 174
- “SetSynchInfo function” on page 175
- “Shutdown function” on page 175
- “StartSynchronizationDelete function” on page 175
- “StopSynchronizationDelete function” on page 176
- “StrToUUID function” on page 176
- “StrToUUID function” on page 176
- “Synchronize function” on page 177
- “Synchronize function” on page 177
- “SynchronizeFromProfile function” on page 177
- “TriggerEvent function” on page 178
- “UUIDToStr function” on page 178
- “UUIDToStr function” on page 179
- “ValidateDatabase function” on page 179

UltraLite_Connection_iface class

The connection interface.

Syntax

public **UltraLite_Connection_iface**

Derived classes

- "UltraLite_Connection class" on page 153

Members

All members of UltraLite_Connection_iface, including all inherited members.

- “CancelGetNotification function” on page 158
- “ChangeEncryptionKey function” on page 158
- “Checkpoint function” on page 159
- “Commit function” on page 159
- “CountUploadRows function” on page 159
- “CreateNotificationQueue function” on page 159
- “DeclareEvent function” on page 160
- “DestroyNotificationQueue function” on page 161
- “ExecuteNextSQLPassthroughScript function” on page 161
- “ExecuteSQLPassthroughScripts function” on page 162
- “GetConnectionNum function” on page 162
- “GetDatabaseID function” on page 162
- “GetDatabaseProperty function” on page 162
- “GetDatabaseProperty function” on page 163
- “GetLastDownloadTime function” on page 163
- “GetLastIdentity function” on page 163
- “GetNewUUID function” on page 164
- “GetNewUUID function” on page 164
- “GetNotification function” on page 164
- “GetNotificationParameter function” on page 165
- “GetSchema function” on page 166
- “GetSqlca function” on page 166
- “GetSQLPassthroughScriptCount function” on page 166
- “GetSuspend function (deprecated)” on page 166
- “GetSynchResult function” on page 166
- “GetUtilityULValue function” on page 167
- “GlobalAutoincUsage function” on page 167
- “GrantConnectTo function” on page 167
- “InitSynchInfo function” on page 168
- “InitSynchInfo function” on page 168
- “OpenTable function” on page 168
- “OpenTableEx function” on page 169
- “OpenTableWithIndex function” on page 169
- “PrepareStatement function” on page 170
- “RegisterForEvent function” on page 170
- “ResetLastDownloadTime function” on page 171
- “RevokeConnectFrom function” on page 171
- “Rollback function” on page 172
- “RollbackPartialDownload function” on page 172
- “SendNotification function” on page 172
- “SetDatabaseID function” on page 173
- “SetDatabaseOption function” on page 173
- “SetDatabaseOption function” on page 173
- “SetSuspend function (deprecated)” on page 174

- "SetSynchInfo function" on page 174
- "SetSynchInfo function" on page 175
- "Shutdown function" on page 175
- "StartSynchronizationDelete function" on page 175
- "StopSynchronizationDelete function" on page 176
- "StrToUUID function" on page 176
- "StrToUUID function" on page 176
- "Synchronize function" on page 177
- "Synchronize function" on page 177
- "SynchronizeFromProfile function" on page 177
- "TriggerEvent function" on page 178
- "UUIDToStr function" on page 178
- "UUIDToStr function" on page 179
- "ValidateDatabase function" on page 179

CancelGetNotification function

Cancel any pending get-notification calls on all queues matching the given name.

Syntax

ul_u_long **UltraLite_Connection_iface::CancelGetNotification(**
  const ULValue & *queue_name*
)

Parameters

- **queue_name**   Name of the queue to cancel.

Returns

The number of affected queues (not the number of blocked reads necessarily).

See also

- "DeclareEvent function" on page 160
- "DestroyNotificationQueue function" on page 161
- "GetNotification function" on page 164
- "RegisterForEvent function" on page 170
- "SendNotification function" on page 172
- "TriggerEvent function" on page 178

ChangeEncryptionKey function

Changes the encryption key.

Syntax

bool **UltraLite_Connection_iface::ChangeEncryptionKey(**
  const ULValue & *new_key*
)

Parameters

- **new_key** The new encryption key value for the database.

Checkpoint function

Checkpoints the database.

Syntax

bool **UltraLite_Connection_iface::Checkpoint()**

Commit function

Commits the current transaction.

Syntax

bool **UltraLite_Connection_iface::Commit()**

CountUploadRows function

Determines the number of rows that need to be uploaded.

Syntax

ul_u_long **UltraLite_Connection_iface::CountUploadRows(**
  const ULValue & *pub_list*,
  ul_u_long *threshold*
)

Parameters

- **pub_list** A comma separated list of publications to consider.
- **threshold** The limit on the number of rows to count.

CreateNotificationQueue function

Create an event notification queue for this connection.

Syntax

bool **UltraLite_Connection_iface::CreateNotificationQueue(**
  const ULValue & *name*,
  const ULValue & *parameters*
)

Parameters

- **name** Name for new queue.
- **parameters** creation parameters; currently unused, set to NULL.

Remarks

Queue names are scoped per-connection, so different connections can create queues with the same name. When an event notification is sent, all queues in the database with a matching name receive (a separate instance of) the notification. Names are case insensitive. A default queue is created on demand for each connection when calling "RegisterForEvent function" on page 170 if no queue is specified. This call fails with an error if the name already exists or isn't valid.

See also

- "CancelGetNotification function" on page 158
- "DeclareEvent function" on page 160
- "DestroyNotificationQueue function" on page 161
- "GetNotification function" on page 164

DeclareEvent function

Declare an event which can then be registered for and triggered.

Syntax

bool **UltraLite_Connection_iface::DeclareEvent(**
  const ULValue & *event_name*
)

Parameters

- **event_name** name for new user-defined event

Returns

ul_true if the event was declared successfully, ul_false if the name is already used or not valid.

Remarks

UltraLite predefines some system events triggered by operations on the database or the environment. This function declares user-defined events. User-defined events are triggered with "TriggerEvent function" on page 178. The event name must be unique. Names are case insensitive.

See also

- “CancelGetNotification function” on page 158
- “DestroyNotificationQueue function” on page 161
- “GetNotification function” on page 164
- “RegisterForEvent function” on page 170
- “SendNotification function” on page 172
- “TriggerEvent function” on page 178

DestroyNotificationQueue function

Destroy the given event notification queue.

Syntax

bool **UltraLite_Connection_iface::DestroyNotificationQueue(**
  const ULValue & *name*
)

Parameters

- **name**   name of queue to destroy

Remarks

A warning is signaled if unread notifications remain in the queue. Unread notifications are discarded. A connection's default event queue, if created, is destroyed when the connection is closed.

See also

- “CancelGetNotification function” on page 158
- “DeclareEvent function” on page 160
- “GetNotification function” on page 164
- “RegisterForEvent function” on page 170
- “SendNotification function” on page 172
- “TriggerEvent function” on page 178

ExecuteNextSQLPassthroughScript function

Executes the next SQL Passthrough script.

Syntax

bool **UltraLite_Connection_iface::ExecuteNextSQLPassthroughScript()**

Returns

False if any errors occurred while executing the script

See also

- "ExecuteSQLPassthroughScripts function" on page 162
- "GetSQLPassthroughScriptCount function" on page 166

ExecuteSQLPassthroughScripts function

Executes all the available SQL Passthrough scripts.

Syntax

bool **UltraLite_Connection_iface::ExecuteSQLPassthroughScripts()**

Returns

False if an errors occurred while executing the scripts

See also

- "ExecuteNextSQLPassthroughScript function" on page 161
- "GetSQLPassthroughScriptCount function" on page 166

GetConnectionNum function

Gets the connection number.

Syntax

ul_connection_num **UltraLite_Connection_iface::GetConnectionNum()**

GetDatabaseID function

Gets the database ID used for global autoincrement columns.

Syntax

ul_u_long **UltraLite_Connection_iface::GetDatabaseID()**

GetDatabaseProperty function

Gets the Database Property.

Syntax

ULValue **UltraLite_Connection_iface::GetDatabaseProperty(**
  ul_database_property_id *id*
)

Parameters

- **id** The ID of the property being requested.

Returns

The value of the requested property.

GetDatabaseProperty function

Gets the Database Property.

Syntax

ULValue **UltraLite_Connection_iface::GetDatabaseProperty(**
  const ULValue & *prop_name*
)

Parameters

- **prop_name** The string name of the property being requested.

Returns

The value of the requested property.

GetLastDownloadTime function

Gets the time of the last download.

Syntax

bool **UltraLite_Connection_iface::GetLastDownloadTime(**
  const ULValue & *pub_list*,
  DECL_DATETIME * *value*
)

Parameters

- **pub_list** A comma separated list of publications to consider.
- **value** The last download time.

GetLastIdentity function

Gets the @@identity value.

Syntax

ul_u_big **UltraLite_Connection_iface::GetLastIdentity()**

GetNewUUID function

Creates a new UUID.

Syntax

bool **UltraLite_Connection_iface::GetNewUUID(**
  p_ul_binary *uuid*
)

Parameters

- **uuid** The new UUID value.

GetNewUUID function

Creates a new UUID.

Syntax

bool **UltraLite_Connection_iface::GetNewUUID(**
  GUID * *uuid*
)

Parameters

- **uuid** The new UUID value.

GetNotification function

Read an event notification.

Syntax

ULValue **UltraLite_Connection_iface::GetNotification(**
  const ULValue & *queue_name*,
  ul_u_long *wait_ms*
)

Parameters

- **queue_name** queue to read or NULL for default connection queue
- **wait_ms** time to wait (block) before returning

Returns

The name of the event read or empty string on error.

Remarks

This call blocks until a notification is received or until the given wait period expires. To wait indefinitely, pass UL_READ_WAIT_INFINITE for wait_ms. To cancel a wait, send another notification to the given queue or use "CancelGetNotification function" on page 158. After reading a notification, use ReadNotificationParameter() to retrieve additional parameters by name.

See also

- "CancelGetNotification function" on page 158
- "DeclareEvent function" on page 160
- "DestroyNotificationQueue function" on page 161
- "RegisterForEvent function" on page 170
- "SendNotification function" on page 172
- "TriggerEvent function" on page 178

GetNotificationParameter function

Get a parameter for the event notification just read by "GetNotification function" on page 164.

Syntax

```
ULValue UltraLite_Connection_iface::GetNotificationParameter(
  const ULValue & queue_name,
  const ULValue & parameter_name
)
```

Parameters

- **queue_name** queue name matching "GetNotification function" on page 164 call
- **parameter_name** name of parameter to read (or "*")

Returns

The parameter value or empty string on error.

Remarks

Only the parameters from the most-recently read notification on the given queue are available. Parameters are retrieved by name. A parameter name of "*" retrieves the entire parameter string.

See also

- "CancelGetNotification function" on page 158
- "DeclareEvent function" on page 160
- "DestroyNotificationQueue function" on page 161
- "GetNotification function" on page 164
- "RegisterForEvent function" on page 170
- "SendNotification function" on page 172
- "TriggerEvent function" on page 178

GetSQLPassthroughScriptCount function

Gets the number of SQL Passthrough scripts that can be run.

Syntax

ul_u_long **UltraLite_Connection_iface::GetSQLPassthroughScriptCount()**

See also

- "ExecuteNextSQLPassthroughScript function" on page 161
- "ExecuteSQLPassthroughScripts function" on page 162

GetSchema function

Gets the database schema.

Syntax

UltraLite_DatabaseSchema * **UltraLite_Connection_iface::GetSchema()**

GetSqlca function

Gets the communication area associated with this connection.

Syntax

ULSqlcaBase const & **UltraLite_Connection_iface::GetSqlca()**

GetSuspend function (deprecated)

Gets the Suspend property.

Syntax

bool **UltraLite_Connection_iface::GetSuspend()**

Returns

- True if this connection is suspended.
- False if the connection is not suspended.

GetSynchResult function

Gets the result of the last synchronization.

Syntax

bool **UltraLite_Connection_iface::GetSynchResult(**
  ul_synch_result * *synch_result*
)

Parameters

- **synch_result** A pointer to the "ul_synch_result struct" on page 139 structure that holds the synchronization results.

GetUtilityULValue function

Gets a new "ULValue class" on page 241.

Syntax

ULValue **UltraLite_Connection_iface::GetUtilityULValue()**

Remarks

A "ULValue class" on page 241 object must be bound to a connection in order for many of its methods to succeed.

GlobalAutoincUsage function

Gets the percent of the global autoincrement values used by the counter.

Syntax

ul_u_short **UltraLite_Connection_iface::GlobalAutoincUsage()**

GrantConnectTo function

Grants connection access to a given user.

Syntax

bool **UltraLite_Connection_iface::GrantConnectTo(**
  const ULValue & *uid*,
  const ULValue & *pwd*
)

Parameters

- **uid** The user ID for which access to connect is granted.
- **pwd** The password for the authorized user ID.

Remarks

To create a new user, specify both a new user ID and a password.

To change a password, specify an existing user ID, but set a new password for that user.

InitSynchInfo function

Initializes the synchronization information structure.

Syntax

void **UltraLite_Connection_iface::InitSynchInfo(**
  ul_synch_info_a * *info*
)

Parameters

- **info**   A pointer to the ul_synch_info structure that holds the synchronization parameters.

InitSynchInfo function

Initializes the synchronization information structure.

Syntax

void **UltraLite_Connection_iface::InitSynchInfo(**
  ul_synch_info_w2 * *info*
)

Parameters

- **info**   A pointer to the ul_synch_info structure that holds the synchronization parameters.

OpenTable function

Opens a table.

Syntax

UltraLite_Table * **UltraLite_Connection_iface::OpenTable(**
  const ULValue & *table_id*,
  const ULValue & *persistent_name*
)

Parameters

- **table_id**   The table name or ordinal.
- **persistent_name**   The instance name used for suspending.

Remarks

When the application first opens a table, the cursor position is set to BeforeFirst().

OpenTableEx function

Opens a table to retrieve rows.

Syntax

UltraLite_Table * **UltraLite_Connection_iface::OpenTableEx(**
  const ULValue & *table_id*,
  ul_table_open_type *open_type*,
  const ULValue & *parms*,
  const ULValue & *persistent_name*
)

Parameters

- **table_id** The table name or ordinal.
- **open_type** Controls how the rows are returned.
- **parms** Optional parameters depending on the open type (like index name)
- **persistent_name** The instance name used for suspending.

Remarks

The order of the rows depends on index used to open the table. If no index is used, the rows are in arbitrary order. When the application first opens a table, the cursor position is set to BeforeFirst().

There can be a performance benefit to opening the table without using an index. However, when no index is used the table returned cannot be used to alter data and lookups cannot be performed.

OpenTableWithIndex function

Opens a table, with the named index to order the rows.

Syntax

UltraLite_Table * **UltraLite_Connection_iface::OpenTableWithIndex(**
  const ULValue & *table_id*,
  const ULValue & *index_id*,
  const ULValue & *persistent_name*
)

Parameters

- **table_id** The table name or ordinal.
- **index_id** The index name or ordinal.
- **persistent_name** The instance name used for suspending.

Remarks

When the application first opens a table, the cursor position is set to BeforeFirst().

PrepareStatement function

Prepares a SQL statement.

Syntax

UltraLite_PreparedStatement * **UltraLite_Connection_iface::PrepareStatement(**
  const ULValue & *sql*,
  const ULValue & *persistent_name*
)

Parameters

- **sql** The SQL statement as a string.
- **persistent_name** The instance name used for suspending.

RegisterForEvent function

Register or unregister (a queue) to receive notifications of an event.

Syntax

bool **UltraLite_Connection_iface::RegisterForEvent(**
  const ULValue & *event_name*,
  const ULValue & *object_name*,
  const ULValue & *queue_name*,
  bool *register_not_unreg*
)

Parameters

- **event_name** system or user-defined event to register for
- **object_name** object to which event applies (like table name)
- **queue_name** NULL means default connection queue
- **register_not_unreg** true to register, false to unregister

Returns

True if the registration succeeded, false if the queue or event does not exist.

Remarks

If no queue name is supplied, the default connection queue is implied, and created if required. Certain system events allow specification of an object name to which the event applies. For example, the TableModified event can specify the table name. Unlike "SendNotification function" on page 172, only the specific queue registered will receive notifications of the event - other queues with the same name on different connections will not (unless they are also explicit registered).

The predefined system events are:

- TableModified - triggered when rows in a table are inserted, updated, or deleted. One notification is sent per request, no matter how many rows were affected by the request. The `object_name` parameter specifies the table to monitor. A value of "*" means all tables in the database. This event has a parameter named 'table_name' whose value is the name of the modified table.
- Commit - triggered after any commit completes. This event has no parameters.
- SyncComplete - triggered after synchronization completes. This event has no parameters.

See also

- “CancelGetNotification function” on page 158
- “DeclareEvent function” on page 160
- “DestroyNotificationQueue function” on page 161
- “GetNotification function” on page 164
- “SendNotification function” on page 172
- “TriggerEvent function” on page 178

ResetLastDownloadTime function

Resets the time of the last download of the named publication.

Syntax

bool **UltraLite_Connection_iface::ResetLastDownloadTime(**
  const ULValue & *pub_list*
)

Parameters

- **pub_list** The publication to reset.

RevokeConnectFrom function

Deletes an existing user.

Syntax

bool **UltraLite_Connection_iface::RevokeConnectFrom(**
  const ULValue & *uid*
)

Parameters

- **uid** The user ID for whom the authority to connect is being revoked.

Rollback function

Rolls back the current transaction.

Syntax

bool **UltraLite_Connection_iface::Rollback()**

RollbackPartialDownload function

Rolls back a partial download.

Syntax

bool **UltraLite_Connection_iface::RollbackPartialDownload()**

SendNotification function

Send a notification to all queues matching the given name.

Syntax

ul_u_long **UltraLite_Connection_iface::SendNotification(**
  const ULValue & *queue_name*,
  const ULValue & *event_name*,
  const ULValue & *parameters*
)

Parameters

- **queue_name** target queue name (or "*")
- **event_name** identity for notification
- **parameters** parameters option list or NULL

Returns

The number of notifications sent (the number of matching queues).

Remarks

This includes any such queue on the current connection. This call does not block. Use the special queue name "*" to send to all queues. The given event name does not need to correspond to any system or user-defined event; it is simply passed through to identify the notification when read and has meaning only to the sender and receiver. The parameters argument specifies a semicolon delimited name=value pairs option list. After the notification is read, the parameter values are read with "GetNotificationParameter function" on page 165.

See also

- “CancelGetNotification function” on page 158
- “DeclareEvent function” on page 160
- “DestroyNotificationQueue function” on page 161
- “GetNotification function” on page 164
- “RegisterForEvent function” on page 170
- “TriggerEvent function” on page 178

SetDatabaseID function

Sets the database ID used for global autoincrement columns.

Syntax

```
bool UltraLite_Connection_iface::SetDatabaseID(
  ul_u_long value
)
```

Parameters

- **value** The database ID, which determines the starting value for global autoincrement columns.

SetDatabaseOption function

Sets the specified Database Option.

Syntax

```
bool UltraLite_Connection_iface::SetDatabaseOption(
  ul_database_option_id id,
  const ULValue & value
)
```

Parameters

- **id** The ID of the option being set.
- **value** The new value of the option.

SetDatabaseOption function

Sets the specified Database Option.

Syntax

```
bool UltraLite_Connection_iface::SetDatabaseOption(
  const ULValue & option_name,
  const ULValue & value
)
```

Parameters

- **option_name** The string name of the option being set.
- **value** The new value of the option.

SetSuspend function (deprecated)

Sets the Suspend property.

Syntax

void **UltraLite_Connection_iface::SetSuspend(**
  bool *suspend*
)

Parameters

- **suspend** Set to true to suspend the connection, so that its state can be restored when the database is reopened.

Returns

True if the connection is suspended.

Remarks

The connection name (or lack thereof) identifies suspended connections.

SetSynchInfo function

Create a synchronization profile using the given name based on the given ul_synch_info structure.

Syntax

bool **UltraLite_Connection_iface::SetSynchInfo(**
  char const * *profile_name*,
  ul_synch_info_a * *info*
)

Parameters

- **profile_name** The name of a synchronization profile containing the options for the synchronization. If profile_name is null then no profile is used and the info structure should contain all the options for the synchronization.
- **info** A pointer to the ul_synch_info structure that holds the synchronization parameters.

Remarks

This will replace any previous sync profile with this name. Specifying a null pointer for the ul_synch_info will delete the named profile.

See also

- “SynchronizeFromProfile function” on page 177

SetSynchInfo function

Create a synchronization profile using the given name based on the given ul_synch_info structure.

Syntax

bool **UltraLite_Connection_iface::SetSynchInfo(**
  ul_wchar const * *profile_name*,
  ul_synch_info_w2 * *info*
)

Parameters

- **profile_name** The name of a synchronization profile containing the options for the synchronization. If profile_name is null then no profile is used and the info structure should contain all the options for the synchronization.
- **info** A pointer to the ul_synch_info structure that holds the synchronization parameters.

See also

- “SynchronizeFromProfile function” on page 177

Shutdown function

Destroys this connection and any remaining associated objects.

Syntax

void **UltraLite_Connection_iface::Shutdown()**

Remarks

If you do not set the connection to suspend, this connection is rolled back.

StartSynchronizationDelete function

Sets START SYNCHRONIZATION DELETE for this connection.

Syntax

bool **UltraLite_Connection_iface::StartSynchronizationDelete()**

See also

- “StopSynchronizationDelete function” on page 176

StopSynchronizationDelete function

Sets STOP SYNCHRONIZATION DELETE for this connection.

Syntax

bool **UltraLite_Connection_iface::StopSynchronizationDelete()**

See also

- “StartSynchronizationDelete function” on page 175

StrToUUID function

Converts a string to a binary UUID.

Syntax

bool **UltraLite_Connection_iface::StrToUUID(**
  p_ul_binary *dst*,
  size_t *len*,
  const ULValue & *src*
)

Parameters

- **dst**   The UUID value being returned.
- **len**   The length of the ul_binary array.
- **src**   A string holding the UUID value to be converted.

StrToUUID function

Converts a string to a GUID struct.

Syntax

bool **UltraLite_Connection_iface::StrToUUID(**
  GUID * *dst*,
  const ULValue & *src*
)

Parameters

- **dst**   The GUID value being returned.
- **src**   A string holding the UUID value to be converted.

Synchronize function

Synchronizes the database.

Syntax

bool **UltraLite_Connection_iface::Synchronize(**
  ul_synch_info_a * *info*
)

Parameters

- **info** A pointer to the ul_synch_info structure that holds the synchronization parameters.

Example

```
ul_synch_info info;
conn->InitSynchInfo( &info );
info.user_name = UL_TEXT( user_name"user_name" );
info.version = UL_TEXT( version"test" );
conn->Synchronize( &info );
```

Synchronize function

Synchronizes the database.

Syntax

bool **UltraLite_Connection_iface::Synchronize(**
  ul_synch_info_w2 * *info*
)

Parameters

- **info** A pointer to the ul_synch_info structure that holds the synchronization parameters.

See also

- "Synchronize function" on page 177

SynchronizeFromProfile function

Synchronize the database using the given profile and merge parameters.

Syntax

bool **UltraLite_Connection_iface::SynchronizeFromProfile(**
  ULValue const & *profile_name*,
  ULValue const & *merge_parms*
)

Parameters

- **profile_name** Profile name to synchronize.
- **merge_parms** Merge parameters for the synchronization

Remarks

This is identical to executing the SYNCHRONIZE statement.

TriggerEvent function

Trigger a user-defined event (and send notification to all registered queues).

Syntax

ul_u_long **UltraLite_Connection_iface::TriggerEvent(**
  const ULValue & *event_name*,
  const ULValue & *parameters*
)

Parameters

- **event_name** name of system or user-defined event to trigger
- **parameters** parameters option list or NULL

Returns

The number of event notifications sent.

Remarks

The parameters value specifies a semicolon delimited name=value pairs option list. After the notification is read, the parameter values are read with GetNotificationParameter.

See also

- "CancelGetNotification function" on page 158
- "DeclareEvent function" on page 160
- "DestroyNotificationQueue function" on page 161
- "GetNotification function" on page 164
- "RegisterForEvent function" on page 170
- "SendNotification function" on page 172

UUIDToStr function

Converts a UUID to an ANSI string.

Syntax

bool **UltraLite_Connection_iface::UUIDToStr(**
  char * *dst*,

size_t *len*,
const ULValue & *src*
)

Parameters

- **dst** The string being returned.
- **len** The length of the ul_binary array.
- **src** The UUID value to be converted to a string.

UUIDToStr function

Converts a UUID to a Unicode string.

Syntax

bool **UltraLite_Connection_iface::UUIDToStr(**
ul_wchar * *dst*,
size_t *len*,
const ULValue & *src*
)

Parameters

- **dst** The Unicode string being returned.
- **len** The length of the ul_binary array.
- **src** The UUID value to be converted to a string.

ValidateDatabase function

Validates the database on this connection.

Syntax

bool **UltraLite_Connection_iface::ValidateDatabase(**
ul_u_short *flags*,
ul_validate_callback_fn *fn*,
ul_void * *user_data*,
const ULValue & *table_id*
)

Parameters

- **flags** Flags controlling the type of validation
- **fn** Function to receive validation progress information
- **user_data** User data to send back to the caller via the callback
- **table_id** Optionally, a specific table to validate

Returns

False if there were errors during the validation.

Remarks

Depending on the flags passed to this routine, the low level store and/or the indexes can be validated. To receive information during the validation, implement a callback function and pass the address to this routine. To limit the validation to a specific table, pass in the table name or ID as the last parameter.

UltraLite_Cursor_iface class

Represents a bi-directional cursor in an UltraLite database.

Syntax

public **UltraLite_Cursor_iface**

Derived classes

- "UltraLite_ResultSet class" on page 207
- "UltraLite_Table class" on page 222

Members

All members of UltraLite_Cursor_iface, including all inherited members.

- "AfterLast function" on page 181
- "BeforeFirst function" on page 182
- "Delete function" on page 182
- "First function" on page 182
- "Get function" on page 182
- "GetRowCount function" on page 182
- "GetState function" on page 183
- "GetStreamReader function" on page 183
- "GetStreamWriter function" on page 183
- "GetSuspend function (deprecated)" on page 184
- "IsNull function" on page 184
- "Last function" on page 184
- "Next function" on page 184
- "Previous function" on page 185
- "Relative function" on page 185
- "Set function" on page 185
- "SetDefault function" on page 186
- "SetNull function" on page 186
- "SetSuspend function (deprecated)" on page 186
- "Update function" on page 187
- "UpdateBegin function" on page 187

Remarks

Cursors are sets of rows from either a table or the result set from a query.

AfterLast function

Moves the cursor after the last row.

Syntax

bool **UltraLite_Cursor_iface::AfterLast()**

BeforeFirst function

Moves the cursor before the first row.

Syntax

bool **UltraLite_Cursor_iface::BeforeFirst()**

Delete function

Deletes the current row and moves it to the next valid row.

Syntax

bool **UltraLite_Cursor_iface::Delete()**

Remarks

If this cursor is a table opened without an index, then the data is considered read only and rows cannot be deleted.

First function

Moves the cursor to the first row.

Syntax

bool **UltraLite_Cursor_iface::First()**

Get function

Fetches a value from a column.

Syntax

```
ULValue UltraLite_Cursor_iface::Get(
  const ULValue & column_id
)
```

Parameters

- **column_id** The name or ordinal of the column.

GetRowCount function

Gets the number of rows in the table.

Syntax

ul_u_long **UltraLite_Cursor_iface::GetRowCount(**
   [ul_u_long *threshold*]
)

Parameters

- **threshold** Optional parameter setting a threshold number of rows to count. If threshold is supplied and is greater than zero, the maximum row count returned is the threshold value. The actual number of rows may be equal to or greater than the threshold value.

Remarks

Setting a threshold value to limit the number of rows to count can be useful in situations where it is important to know if more than a specific number of rows exist. For example, code that is populating a list that holds 25 items can determine whether it is necessary to permit the user to have an option to view additional rows.

Calling this method is equivalent to executing "SELECT COUNT(*) FROM table".

GetState function

Gets the internal state of the cursor.

Syntax

UL_RS_STATE **UltraLite_Cursor_iface::GetState()**

Remarks

See the UL_RS_STATE enumeration in ulglobal.h

GetStreamReader function

Gets a stream reader object for reading string or binary column data in chunks.

Syntax

UltraLite_StreamReader * **UltraLite_Cursor_iface::GetStreamReader(**
 const ULValue & *id*
)

Parameters

- **id** A column identifier, which may be either a 1-based ordinal number or a column name.

GetStreamWriter function

Gets a stream writer for streaming string/binary data into a column.

Syntax

UltraLite_StreamWriter * **UltraLite_Cursor_iface::GetStreamWriter(**
  const ULValue & *column_id*
)

Parameters

- **column_id** A column identifier, which may be either a 1-based ordinal number or a column name.

GetSuspend function (deprecated)

Gets the value of the Suspend property.

Syntax

bool **UltraLite_Cursor_iface::GetSuspend()**

Returns

- True if this cursor is suspended.
- False if it is not.

IsNull function

Checks if a column is NULL.

Syntax

bool **UltraLite_Cursor_iface::IsNull(**
  const ULValue & *column_id*
)

Parameters

- **column_id** The name or ordinal of the column.

Last function

Moves the cursor to the last row.

Syntax

bool **UltraLite_Cursor_iface::Last()**

Next function

Moves the cursor forward one row.

Syntax

bool **UltraLite_Cursor_iface::Next()**

Returns

- True, if the cursor successfully moves forward. Despite returning true, an error may be signaled even when the cursor moves successfully to the next row. For example, there could be conversion errors while evaluating the SELECT expressions. In this case, errors are also returned when retrieving the column values.
- False, if it fails to move forward. For example, there may not be a next row. In this case, the resulting cursor position is "AfterLast function" on page 181.

Previous function

Moves the cursor back one row.

Syntax

bool **UltraLite_Cursor_iface::Previous()**

Remarks

On failure, the resulting cursor position is "BeforeFirst function" on page 182.

Relative function

Moves the cursor by offset rows from the current cursor position.

Syntax

bool **UltraLite_Cursor_iface::Relative(**
  ul_fetch_offset *offset*
)

Parameters

- **offset** The number of rows to move.

Set function

Sets a column value.

Syntax

bool **UltraLite_Cursor_iface::Set(**
  const ULValue & *column_id*,
  const ULValue & *value*
)

Parameters

- **column_id** A 1-based ordinal number identifying the column.
- **value** The value to which the column is set.

SetDefault function

Sets column(s) to their default value.

Syntax

bool **UltraLite_Cursor_iface::SetDefault(**
  const ULValue & *column_id*
)

Parameters

- **column_id** A 1-based ordinal number identifying the column.

SetNull function

Sets a column to null.

Syntax

bool **UltraLite_Cursor_iface::SetNull(**
  const ULValue & *column_id*
)

Parameters

- **column_id** A 1-based ordinal number identifying the column.

SetSuspend function (deprecated)

Sets the value of the Suspend property.

Syntax

void **UltraLite_Cursor_iface::SetSuspend(**
  bool *suspend*
)

Parameters

- **suspend** True suspends the connection. This allows you to restore the database's state when the database is reopened.

Returns

- True, if the cursor is suspended and restored when the application reopens the database.
- False, if the cursor is not suspended.

Remarks

Use the persistent name parameter when opening the associated object to identify suspended cursors. If you do not supply a persistent name parameter for this cursor, you cannot suspend it.

Update function

Updates the current row.

Syntax

bool **UltraLite_Cursor_iface::Update()**

Remarks

The table must be in update mode for this operation to succeed. Use "UpdateBegin function" on page 187 to switch to update mode.

UpdateBegin function

Selects update mode for setting columns.

Syntax

bool **UltraLite_Cursor_iface::UpdateBegin()**

Remarks

Columns in the primary key may not be modified when in update mode. If this cursor is a table opened without an index, then the data is considered read only and cannot be modified.

UltraLite_DatabaseManager class

Manages synchronization listeners and allows you to drop (delete) UltraLite databases.

Syntax

public **UltraLite_DatabaseManager**

Base classes

- “UltraLite_DatabaseManager_iface class” on page 189

Members

All members of UltraLite_DatabaseManager, including all inherited members.

- “CreateDatabase function” on page 189
- “DropDatabase function” on page 190
- “OpenConnection function” on page 190
- “Shutdown function” on page 191
- “ValidateDatabase function” on page 191

UltraLite_DatabaseManager_iface class

Manages connections and databases.

Syntax

public **UltraLite_DatabaseManager_iface**

Derived classes

- "UltraLite_DatabaseManager class" on page 188

Members

All members of UltraLite_DatabaseManager_iface, including all inherited members.

- "CreateDatabase function" on page 189
- "DropDatabase function" on page 190
- "OpenConnection function" on page 190
- "Shutdown function" on page 191
- "ValidateDatabase function" on page 191

Remarks

Creating a database and establishing a connection to it is a necessary first step in using UltraLite. You should ensure that you are connected properly before attempting any DML with the database.

CreateDatabase function

Creates a new database.

Syntax

```
bool UltraLite_DatabaseManager_iface::CreateDatabase(
  ULSqlcaBase & sqlca,
  ULValue const & access_parms,
  void const * coll,
  ULValue const & create_parms,
  void * reserved
)
```

Parameters

- **sqlca** The initialized sqlca.
- **access_parms** Connection parameters used to access database
- **coll** Collation sequence
- **create_parms** Parameters used to create the database
- **reserved** Reserved (not used currently)

DropDatabase function

Erases an existing database that is already stopped.

Syntax

bool **UltraLite_DatabaseManager_iface::DropDatabase(**
  ULSqlcaBase & *sqlca*,
  const ULValue & *parms_string*
)

Parameters

- **sqlca** The initialized sqlca.
- **parms_string** The database identification parameters.

Remarks

You cannot erase a running database.

OpenConnection function

Opens a new connection to an existing database.

Syntax

UltraLite_Connection * **UltraLite_DatabaseManager_iface::OpenConnection(**
  ULSqlcaBase & *sqlca*,
  ULValue const & *parms_string*
)

Parameters

- **sqlca** The initialized sqlca to associate with the new connection.
- **parms_string** The connection string.

Remarks

The given sqlca is associated with the new connection.

- SQLE_CONNECTION_ALREADY_EXISTS - A connection with the given SQLCA and name (or no name) already exists. Before connecting you must disconnect the existing connection, or specify a different connection name with the CON parameter.
- SQLE_INVALID_LOGON - You supplied an invalid user ID or an incorrect password.
- SQLE_INVALID_SQL_IDENTIFIER - An invalid identifier was supplied through the C language interface. For example, you may have supplied a NULL string for a cursor name.
- SQLE_TOO_MANY_CONNECTIONS - You exceeded the number of concurrent database connections.

To get error information, use the associated "ULSqlca class" on page 144 object. Possible errors include:

Returns

- If the function succeeds, a new connection object is returned.
- If the function fails, NULL is returned.

Shutdown function

Closes all databases and releases the database manager.

Syntax

void **UltraLite_DatabaseManager_iface::Shutdown(**
  ULSqlcaBase & *sqlca*
)

Parameters

- **sqlca** The initialized sqlca.

Remarks

Any remaining associated objects are destroyed. After calling this function, the database manager can no longer be used (nor can any other previously obtained objects).

ValidateDatabase function

Performs low level and index validation on a database.

Syntax

bool **UltraLite_DatabaseManager_iface::ValidateDatabase(**
  ULSqlcaBase & *sqlca*,
  ULValue const & *start_parms*,
  ul_u_short *flags*,
  ul_validate_callback_fn *fn*,
  ul_void * *user_data*
)

Parameters

- **sqlca** The initialized sqlca.
- **start_parms** Parameters for starting the database
- **flags** Flags controlling the type of validation
- **fn** Function to receive validation progress information
- **user_data** User data to send back to the caller via the callback

UltraLite_DatabaseSchema class

Represents the schema of an UltraLite database.

Syntax

public **UltraLite_DatabaseSchema**

Base classes

- "UltraLite_SQLObject_iface class" on page 215
- "UltraLite_DatabaseSchema_iface class" on page 193

Members

All members of UltraLite_DatabaseSchema, including all inherited members.

- "AddRef function" on page 215
- "GetCollationName function" on page 193
- "GetConnection function" on page 215
- "GetIFace function" on page 216
- "GetPublicationCount function" on page 193
- "GetPublicationID function" on page 194
- "GetPublicationName function" on page 194
- "GetTableCount function" on page 194
- "GetTableName function" on page 194
- "GetTableSchema function" on page 195
- "IsCaseSensitive function" on page 195
- "Release function" on page 216

UltraLite_DatabaseSchema_iface class

DatabaseSchema interface.

Syntax

public **UltraLite_DatabaseSchema_iface**

Derived classes

- "UltraLite_DatabaseSchema class" on page 192

Members

All members of UltraLite_DatabaseSchema_iface, including all inherited members.

- "GetCollationName function" on page 193
- "GetPublicationCount function" on page 193
- "GetPublicationID function" on page 194
- "GetPublicationName function" on page 194
- "GetTableCount function" on page 194
- "GetTableName function" on page 194
- "GetTableSchema function" on page 195
- "IsCaseSensitive function" on page 195

GetCollationName function

Gets the name of the current collation sequence.

Syntax

ULValue **UltraLite_DatabaseSchema_iface::GetCollationName()**

Returns

A "ULValue class" on page 241 that contains a string.

GetPublicationCount function

Gets the number of publications in the database.

Syntax

ul_publication_count **UltraLite_DatabaseSchema_iface::GetPublicationCount()**

Remarks

Publication IDs range from 1 to "GetPublicationCount function" on page 193

GetPublicationID function

Gets a 1-based id for the publication given its name.

Syntax

```
ul_u_short UltraLite_DatabaseSchema_iface::GetPublicationID(
  const ULValue & pub_id
)
```

Parameters

- **pub_id**   A 1-based ordinal number.

GetPublicationName function

Gets the name of a publication given, its 1-based index ID.

Syntax

```
ULValue UltraLite_DatabaseSchema_iface::GetPublicationName(
  const ULValue & pub_id
)
```

Parameters

- **pub_id**   A 1-based ordinal number.

GetTableCount function

Returns the number of tables in the database.

Syntax

```
ul_table_num UltraLite_DatabaseSchema_iface::GetTableCount()
```

Returns

- An integer that represents the number of tables.
- 0 if the connection is not open.

GetTableName function

Gets the name of a table given its 1-based table ID.

Syntax

```
ULValue UltraLite_DatabaseSchema_iface::GetTableName(
  ul_table_num tableID
)
```

Parameters

- **tableID** A 1-based ordinal number.

Returns

The name of the table identified by the specified table ID.

Remarks

Table IDs may change during a schema upgrade. To correctly identify a table, either access it by name, or refresh any cached IDs after a schema upgrade. The "ULValue class" on page 241 object returned is empty if the table does not exist.

GetTableSchema function

Gets a TableSchema object given a 1-based table ID or name.

Syntax

UltraLite_TableSchema * **UltraLite_DatabaseSchema_iface::GetTableSchema(**
 const ULValue & *table_id*
)

Parameters

- **table_id** A 1-based ordinal number.

Returns

UL_NULL if the table does not exist.

IsCaseSensitive function

Gets the case sensitivity of the database.

Syntax

bool **UltraLite_DatabaseSchema_iface::IsCaseSensitive()**

Returns

- True if the database is case sensitive.
- False otherwise.

Remarks

Database case sensitivity affects how indexes on tables and result sets are sorted.

UltraLite_IndexSchema class

Represents the schema of an UltraLite table index.

Syntax

public **UltraLite_IndexSchema**

Base classes

- “UltraLite_SQLObject_iface class” on page 215
- “UltraLite_IndexSchema_iface class” on page 197

Members

All members of UltraLite_IndexSchema, including all inherited members.

- “AddRef function” on page 215
- “GetColumnCount function” on page 197
- “GetColumnName function” on page 197
- “GetConnection function” on page 215
- “GetID function” on page 198
- “GetIFace function” on page 216
- “GetName function” on page 198
- “GetReferencedIndexName function” on page 198
- “GetReferencedTableName function” on page 199
- “GetTableName function” on page 199
- “IsColumnDescending function” on page 199
- “IsForeignKey function” on page 199
- “IsForeignKeyCheckOnCommit function” on page 200
- “IsForeignKeyNullable function” on page 200
- “IsPrimaryKey function” on page 200
- “IsUniqueIndex function” on page 201
- “IsUniqueKey function” on page 201
- “Release function” on page 216

UltraLite_IndexSchema_iface class

Represents an IndexSchema interface.

Syntax

public **UltraLite_IndexSchema_iface**

Derived classes

- "UltraLite_IndexSchema class" on page 196

Members

All members of UltraLite_IndexSchema_iface, including all inherited members.

- "GetColumnCount function" on page 197
- "GetColumnName function" on page 197
- "GetID function" on page 198
- "GetName function" on page 198
- "GetReferencedIndexName function" on page 198
- "GetReferencedTableName function" on page 199
- "GetTableName function" on page 199
- "IsColumnDescending function" on page 199
- "IsForeignKey function" on page 199
- "IsForeignKeyCheckOnCommit function" on page 200
- "IsForeignKeyNullable function" on page 200
- "IsPrimaryKey function" on page 200
- "IsUniqueIndex function" on page 201
- "IsUniqueKey function" on page 201

GetColumnCount function

Gets the number of columns in the index.

Syntax

ul_column_num **UltraLite_IndexSchema_iface::GetColumnCount()**

GetColumnName function

Gets the name of the column given the position of the column in the index.

Syntax

ULValue **UltraLite_IndexSchema_iface::GetColumnName(**
  ul_column_num *col_id_in_index*
)

Parameters

- **col_id_in_index** The 1-based ordinal number indicating the position of the column in the index.

Returns

An empty "ULValue class" on page 241 object if the column does not exist.

Remarks

SQLE_COLUMN_NOT_FOUND if the column name does not exist.

GetID function

Gets the index ID.

Syntax

ul_index_num **UltraLite_IndexSchema_iface::GetID()**

Returns

The ID of the index.

GetName function

Gets the name of the index.

Syntax

ULValue **UltraLite_IndexSchema_iface::GetName()**

GetReferencedIndexName function

Gets the associated primary index name.

Syntax

ULValue **UltraLite_IndexSchema_iface::GetReferencedIndexName()**

Returns

An empty "ULValue class" on page 241 object if the index is not a foreign key.

Remarks

This function is for foreign keys only.

GetReferencedTableName function

Gets the associated primary table name.

Syntax

ULValue **UltraLite_IndexSchema_iface::GetReferencedTableName()**

Returns

An empty "ULValue class" on page 241 object if the index is not a foreign key.

Remarks

This method is for foreign keys only.

GetTableName function

Gets the name of the table containing the index.

Syntax

ULValue **UltraLite_IndexSchema_iface::GetTableName()**

IsColumnDescending function

Determines if the column is in descending order.

Syntax

```
bool UltraLite_IndexSchema_iface::IsColumnDescending(
  const ULValue & column_name
)
```

Parameters

- **column_name** The column name.

Returns

True if the column is in descending order.

Remarks

Sets SQLE_COLUMN_NOT_FOUND if the column name does not exist.

IsForeignKey function

Checks whether the index is a foreign key.

Syntax

bool **UltraLite_IndexSchema_iface::IsForeignKey()**

Returns

- True if the index is a foreign key.
- False if the index is not a foreign key.

Remarks

Columns in a foreign key may reference another table's non-null, unique index.

IsForeignKeyCheckOnCommit function

Checks whether referential integrity for the foreign key is performed on commits or on inserts and updates.

Syntax

bool **UltraLite_IndexSchema_iface::IsForeignKeyCheckOnCommit()**

Returns

- True if this foreign key checks referential integrity on commit.
- False if this foreign key checks referential integrity on insert.

IsForeignKeyNullable function

Checks whether the foreign key is nullable.

Syntax

bool **UltraLite_IndexSchema_iface::IsForeignKeyNullable()**

Returns

- True if the index is a unique foreign key constraint.
- False if the foreign key is not nullable

IsPrimaryKey function

Checks whether the index is a primary key.

Syntax

bool **UltraLite_IndexSchema_iface::IsPrimaryKey()**

Returns

- True if the index is a primary key.
- False if the index is not a primary key.

Remarks

Columns in the primary key may not be null.

IsUniqueIndex function

Checks whether the index is unique.

Syntax

bool **UltraLite_IndexSchema_iface::IsUniqueIndex()**

Returns

- True if the index is unique.
- False if the index is not unique.

IsUniqueKey function

Checks whether the index is a unique key.

Syntax

bool **UltraLite_IndexSchema_iface::IsUniqueKey()**

Returns

- True if the index is a primary key or a unique constraint.
- False if the index is neither a primary key nor a unique constraint.

Remarks

Columns in a unique key may not be null.

UltraLite_PreparedStatement class

Prepares a statement with placeholders, and then assigns values to the placeholders after executing the statement.

Syntax

public **UltraLite_PreparedStatement**

Base classes

- "UltraLite_SQLObject_iface class" on page 215
- "UltraLite_PreparedStatement_iface class" on page 203

Members

All members of UltraLite_PreparedStatement, including all inherited members.

- "AddRef function" on page 215
- "ExecuteQuery function" on page 203
- "ExecuteStatement function" on page 203
- "GetConnection function" on page 215
- "GetIFace function" on page 216
- "GetPlan function" on page 204
- "GetPlan function" on page 204
- "GetSchema function" on page 204
- "GetStreamWriter function" on page 205
- "HasResultSet function" on page 205
- "Release function" on page 216
- "SetParameter function" on page 205
- "SetParameterNull function" on page 206

UltraLite_PreparedStatement_iface class

The PreparedStatement interface.

Syntax

public **UltraLite_PreparedStatement_iface**

Derived classes

- "UltraLite_PreparedStatement class" on page 202

Members

All members of UltraLite_PreparedStatement_iface, including all inherited members.

- "ExecuteQuery function" on page 203
- "ExecuteStatement function" on page 203
- "GetPlan function" on page 204
- "GetPlan function" on page 204
- "GetSchema function" on page 204
- "GetStreamWriter function" on page 205
- "HasResultSet function" on page 205
- "SetParameter function" on page 205
- "SetParameterNull function" on page 206

ExecuteQuery function

Executes a SQL SELECT statement as a query.

Syntax

UltraLite_ResultSet * **UltraLite_PreparedStatement_iface::ExecuteQuery()**

Returns

The result set of the query, as a set of rows.

ExecuteStatement function

Executes a statement that does not return a result set, such as a SQL INSERT, DELETE or UPDATE statement.

Syntax

ul_s_long **UltraLite_PreparedStatement_iface::ExecuteStatement()**

GetPlan function

Gets a text-based description of query execution plan.

Syntax

size_t **UltraLite_PreparedStatement_iface::GetPlan(**
  char * *buffer*,
  size_t *size*
)

Parameters

- **buffer** The buffer in which to receive the plan description.
- **size** The size, in ASCII characters, of the buffer.

Returns

A string describing the access plan UltraLite will use to execute a query.

Remarks

This function is intended primarily for use during development.

GetPlan function

Gets a text-based description of query execution plan in wide characters.

Syntax

size_t **UltraLite_PreparedStatement_iface::GetPlan(**
  ul_wchar * *buffer*,
  size_t *size*
)

Parameters

- **buffer** The buffer to receive the plan description.
- **size** The size, in ul_wchars, of the buffer.

Returns

A string describing the access plan UltraLite will use to execute a query.

Remarks

This function is intended primarily for use during development.

GetSchema function

Gets the schema for the result set.

Syntax

UltraLite_ResultSetSchema * **UltraLite_PreparedStatement_iface::GetSchema()**

GetStreamWriter function

Gets a stream writer for streaming string/binary data into a parameter.

Syntax

UltraLite_StreamWriter * **UltraLite_PreparedStatement_iface::GetStreamWriter(**
ul_column_num *parameter_id*
)

Parameters

- **parameter_id** A column identifier, which may be either a 1-based ordinal number or a column name.

HasResultSet function

Determines if the SQL statement has a result set.

Syntax

bool **UltraLite_PreparedStatement_iface::HasResultSet()**

Returns

- True if a result set is generated when this statement is executed.
- False if no result set is generated.

SetParameter function

Sets a parameter for the SQL statement.

Syntax

void **UltraLite_PreparedStatement_iface::SetParameter(**
ul_column_num *parameter_id*,
ULValue const & *value*
)

Parameters

- **parameter_id** The 1-based ordinal of the parameter.
- **value** The value to set the parameter.

SetParameterNull function

Sets a parameter to null.

Syntax

void **UltraLite_PreparedStatement_iface::SetParameterNull(**
  ul_column_num *parameter_id*
)

Parameters

- **parameter_id** The 1-based ordinal of the parameter.

UltraLite_ResultSet class

Represents an editable result set in an UltraLite database.

Syntax

public **UltraLite_ResultSet**

Base classes

- "UltraLite_SQLObject_iface class" on page 215
- "UltraLite_ResultSet_iface class" on page 208
- "UltraLite_Cursor_iface class" on page 181

Members

All members of UltraLite_ResultSet, including all inherited members.

- "AddRef function" on page 215
- "AfterLast function" on page 181
- "BeforeFirst function" on page 182
- "Delete function" on page 182
- "DeleteNamed function" on page 208
- "First function" on page 182
- "Get function" on page 182
- "GetConnection function" on page 215
- "GetIFace function" on page 216
- "GetRowCount function" on page 182
- "GetSchema function" on page 208
- "GetState function" on page 183
- "GetStreamReader function" on page 183
- "GetStreamWriter function" on page 183
- "GetSuspend function (deprecated)" on page 184
- "IsNull function" on page 184
- "Last function" on page 184
- "Next function" on page 184
- "Previous function" on page 185
- "Relative function" on page 185
- "Release function" on page 216
- "Set function" on page 185
- "SetDefault function" on page 186
- "SetNull function" on page 186
- "SetSuspend function (deprecated)" on page 186
- "Update function" on page 187
- "UpdateBegin function" on page 187

Remarks

An editable result set allows you to perform positioned updates and deletes.

UltraLite_ResultSet_iface class

The ResultSet interface.

Syntax

public **UltraLite_ResultSet_iface**

Derived classes

- "UltraLite_ResultSet class" on page 207

Members

All members of UltraLite_ResultSet_iface, including all inherited members.

- "DeleteNamed function" on page 208
- "GetSchema function" on page 208

DeleteNamed function

Deletes the current row and moves it to the next valid row.

Syntax

bool **UltraLite_ResultSet_iface:DeleteNamed(**
  const ULValue & *table_name*
)

Parameters

- **table_name** A table name or its correlation (required when the database has multiple columns that share the same table name.

GetSchema function

Gets the schema for this result set.

Syntax

UltraLite_ResultSetSchema * **UltraLite_ResultSet_iface::GetSchema()**

UltraLite_ResultSetSchema class

Retrieves schema information about a result set. For example, column names, total number of columns, column scales, column sizes, and column SQL types.

Syntax

public **UltraLite_ResultSetSchema**

Base classes

- “UltraLite_SQLObject_iface class” on page 215
- “UltraLite_RowSchema_iface class” on page 210

Members

All members of UltraLite_ResultSetSchema, including all inherited members.

- “AddRef function” on page 215
- “GetBaseColumnName function” on page 210
- “GetColumnCount function” on page 211
- “GetColumnID function” on page 211
- “GetColumnName function” on page 211
- “GetColumnPrecision function” on page 212
- “GetColumnScale function” on page 213
- “GetColumnSize function” on page 213
- “GetColumnSQLName function” on page 212
- “GetColumnSQLType function” on page 212
- “GetColumnType function” on page 214
- “GetConnection function” on page 215
- “GetIFace function” on page 216
- “Release function” on page 216

UltraLite_RowSchema_iface class

The RowSchema interface.

Syntax

public **UltraLite_RowSchema_iface**

Derived classes

- "UltraLite_ResultSetSchema class" on page 209
- "UltraLite_TableSchema class" on page 230

Members

All members of UltraLite_RowSchema_iface, including all inherited members.

- "GetBaseColumnName function" on page 210
- "GetColumnCount function" on page 211
- "GetColumnID function" on page 211
- "GetColumnName function" on page 211
- "GetColumnPrecision function" on page 212
- "GetColumnScale function" on page 213
- "GetColumnSize function" on page 213
- "GetColumnSQLName function" on page 212
- "GetColumnSQLType function" on page 212
- "GetColumnType function" on page 214

GetBaseColumnName function

Gets the combined base and column name of a column of a result set, even if this column has a correlation name or alias.

Syntax

ULValue **UltraLite_RowSchema_iface::GetBaseColumnName(**
 ul_column_num *column_id*
)

Parameters

- **column_id** A 1-based ordinal number.

Returns

- A combined "ULValue class" on page 241 object.
- An empty name if the column is not part of a table.

Remarks

SQLE_COLUMN_NOT_FOUND is set if the column name does not exist.

GetColumnCount function

Gets the number of columns in the table.

Syntax

ul_column_num **UltraLite_RowSchema_iface::GetColumnCount()**

GetColumnID function

Gets the 1-based column ID.

Syntax

ul_column_num **UltraLite_RowSchema_iface::GetColumnID(**
  const ULValue & *column_name*
)

Parameters

- **column_name** The column name.

Returns

0 if the column does not exist.

Remarks

Sets SQLE_COLUMN_NOT_FOUND if the column name does not exist.

GetColumnName function

Gets the name of a column given its 1-based ID.

Syntax

ULValue **UltraLite_RowSchema_iface::GetColumnName(**
  ul_column_num *column_id*
)

Parameters

- **column_id** A 1-based ordinal number.

Returns

An empty "ULValue class" on page 241 object if the column does not exist.

Remarks

This will be the alias or correlation name for SELECT statements.

Sets SQLE_COLUMN_NOT_FOUND if the column name does not exist

GetColumnPrecision function

Gets the precision of a numeric column.

Syntax

size_t **UltraLite_RowSchema_iface::GetColumnPrecision(**
  const ULValue & *column_id*
)

Parameters

- **column_id** A 1-based ordinal number.

Returns

0 if the column is not a numeric type or if the column does not exist.

Remarks

Sets SQLE_COLUMN_NOT_FOUND if the column name does not exist.

Sets SQLE_DATATYPE_NOT_ALLOWED if the column type is not a numeric column.

GetColumnSQLName function

Gets the SQL name for a column in a result set.

Syntax

ULValue **UltraLite_RowSchema_iface::GetColumnSQLName(**
  ul_column_num *column_id*
)

Parameters

- **column_id** A 1-based ordinal number.

Returns

A combined "ULValue class" on page 241 object.

Remarks

If the column has an alias, then that name is used. Otherwise, if the column in the result set corresponds to a column in a table, then the column name is used. Otherwise, the combined name is empty.

SQLE_COLUMN_NOT_FOUND is set if the column name does not exist.

GetColumnSQLType function

Gets the SQL type of a column.

Syntax

ul_column_sql_type **UltraLite_RowSchema_iface::GetColumnSQLType(**
  const ULValue & *column_id*
)

Parameters

- **column_id** A 1-based ordinal number.

Returns

UL_SQLTYPE_BAD_INDEX if the column does not exist.

Remarks

See the ul_column_sql_type in ulprotos.h.

GetColumnScale function

Gets the scale of a numeric column.

Syntax

size_t **UltraLite_RowSchema_iface::GetColumnScale(**
  const ULValue & *column_id*
)

Parameters

- **column_id** A 1-based ordinal number.

Returns

0 if the column is not a numeric type or if the column does not exist.

Remarks

Sets SQLE_COLUMN_NOT_FOUND if the column name does not exist.

Sets SQLE_DATATYPE_NOT_ALLOWED if the column type is not a numeric.

GetColumnSize function

Gets the size of the column.

Syntax

size_t **UltraLite_RowSchema_iface::GetColumnSize(**
  const ULValue & *column_id*
)

Parameters

- **column_id** A 1-based ordinal number.

Returns

0 if the column does not exist or if the column type does not have a variable length.

Remarks

Sets SQLE_COLUMN_NOT_FOUND if the column name does not exist.

Sets SQLE_DATATYPE_NOT_ALLOWED the column type is not either UL_SQLTYPE_CHAR or UL_SQLTYPE_BINARY.

GetColumnType function

Gets the type of a column.

Syntax

```
ul_column_storage_type UltraLite_RowSchema_iface::GetColumnType(
  const ULValue & column_id
)
```

Parameters

- **column_id** A 1-based ordinal number.

Returns

UL_TYPE_BAD_INDEX if the column does not exist.

Remarks

See the ul_column_storage_type enum in ulprotos.h.

UltraLite_SQLObject_iface class

The SQLObject interface.

Syntax

public **UltraLite_SQLObject_iface**

Derived classes

- "UltraLite_Connection class" on page 153
- "UltraLite_DatabaseSchema class" on page 192
- "UltraLite_IndexSchema class" on page 196
- "UltraLite_PreparedStatement class" on page 202
- "UltraLite_ResultSet class" on page 207
- "UltraLite_ResultSetSchema class" on page 209
- "UltraLite_StreamReader class" on page 217
- "UltraLite_StreamWriter class" on page 221
- "UltraLite_Table class" on page 222
- "UltraLite_TableSchema class" on page 230

Members

All members of UltraLite_SQLObject_iface, including all inherited members.

- "AddRef function" on page 215
- "GetConnection function" on page 215
- "GetIFace function" on page 216
- "Release function" on page 216

AddRef function

Increases the internal reference count for an object.

Syntax

ul_ret_void **UltraLite_SQLObject_iface::AddRef()**

Remarks

To free the object you must match each call to this function with a call to "Release function" on page 216.

GetConnection function

Gets the Connection object.

Syntax

UltraLite_Connection * **UltraLite_SQLObject_iface::GetConnection()**

Returns

The connection associated with this object.

GetIFace function

Reserved for future use.

Syntax

ul_void * **UltraLite_SQLObject_iface::GetIFace(**
  ul_iface_id *iface*
)

Parameters

- **iface** Reserved for future use.

Release function

Releases a reference to an object.

Syntax

ul_u_long **UltraLite_SQLObject_iface::Release()**

Remarks

The object is freed once you have removed all references. You must call this function at least once. If you use "AddRef function" on page 215 you also need a matching call from each "AddRef function" on page 215.

UltraLite_StreamReader class

Represents an UltraLite StreamReader.

Syntax

public **UltraLite_StreamReader**

Base classes

- "UltraLite_SQLObject_iface class" on page 215
- "UltraLite_StreamReader_iface class" on page 218

Members

All members of UltraLite_StreamReader, including all inherited members.

- "AddRef function" on page 215
- "GetByteChunk function" on page 218
- "GetConnection function" on page 215
- "GetIFace function" on page 216
- "GetLength function" on page 219
- "GetStringChunk function" on page 219
- "GetStringChunk function" on page 219
- "Release function" on page 216
- "SetReadPosition function" on page 220

UltraLite_StreamReader_iface class

StreamReader interface.

Syntax

public **UltraLite_StreamReader_iface**

Derived classes

- "UltraLite_StreamReader class" on page 217

Members

All members of UltraLite_StreamReader_iface, including all inherited members.

- "GetByteChunk function" on page 218
- "GetLength function" on page 219
- "GetStringChunk function" on page 219
- "GetStringChunk function" on page 219
- "SetReadPosition function" on page 220

Remarks

This interface supports reading/retrieving of VARCHAR and BINARY columns.

GetByteChunk function

Gets a byte chunk from current StreamReader offset by copying buffer_len bytes in to buffer data.

Syntax

bool **UltraLite_StreamReader_iface::GetByteChunk(**
 ul_byte * *data*,
 size_t *buffer_len*,
 size_t * *len_retn*,
 bool * *morebytes*
)

Parameters

- **data** A pointer to an array of bytes.
- **buffer_len** The length of the buffer, or array. The buffer_len must be greater than or equal to 0.
- **len_retn** An output parameter. The length returned.
- **morebytes** An output parameter. True if there are more bytes to read.

Remarks

The bytes are read from where the last read left off unless you use "SetReadPosition function" on page 220.

GetLength function

Gets the length of a string/binary value.

Syntax

size_t **UltraLite_StreamReader_iface::GetLength(**
  bool *fetch_as_chars*
)

Parameters

- **fetch_as_chars** False for byte length, true for char length.

Returns

- The number of bytes for binary values (fetch_as_chars is ignored for binaries).
- The number of characters or bytes for string values.

GetStringChunk function

Gets a string chunk from current StreamReader offset by copying buffer_len wide characters in to buffer str.

Syntax

bool **UltraLite_StreamReader_iface::GetStringChunk(**
  ul_wchar * *str*,
  size_t *buffer_len*,
  size_t * *len_retn*,
  bool * *morebytes*
)

Parameters

- **str** A pointer to an array of wide characters.
- **buffer_len** The length of the buffer.
- **len_retn** An output parameter. The length returned.
- **morebytes** An output parameter. True if there are more characters to read.

Remarks

Characters are read from where the last read left off unless you use "SetReadPosition function" on page 220.

GetStringChunk function

Gets a string chunk from current StreamReader offset by copying buffer_len bytes in to buffer str.

Syntax

bool **UltraLite_StreamReader_iface::GetStringChunk(**
  char * *str*,
  size_t *buffer_len*,
  size_t * *len_retn*,
  bool * *morebytes*
)

Parameters

- **str** A pointer to an array of characters.
- **buffer_len** The length of the buffer, or array. The buffer_len must be greater than or equal to 0.
- **len_retn** An output parameter. The length returned.
- **morebytes** An output parameter. True if there are more characters to read.

Remarks

The characters are read from where the last read left off unless you use "SetReadPosition function" on page 220.

SetReadPosition function

Sets the offset in the data for the next read.

Syntax

bool **UltraLite_StreamReader_iface::SetReadPosition(**
  size_t *offset*,
  bool *offset_in_chars*
)

Parameters

- **offset** The offset.
- **offset_in_chars** True if offset is in characters. False if the offset is in bytes.

UltraLite_StreamWriter class

Represents an UltraLite StreamWriter.

Syntax

public **UltraLite_StreamWriter**

Base classes

- "UltraLite_SQLObject_iface class" on page 215

Members

All members of UltraLite_StreamWriter, including all inherited members.

- "AddRef function" on page 215
- "GetConnection function" on page 215
- "GetIFace function" on page 216
- "Release function" on page 216

UltraLite_Table class

Represents a table in an UltraLite database.

Syntax

public **UltraLite_Table**

Base classes

- "UltraLite_SQLObject_iface class" on page 215
- "UltraLite_Table_iface class" on page 224
- "UltraLite_Cursor_iface class" on page 181

Members

All members of UltraLite_Table, including all inherited members.

- “AddRef function” on page 215
- “AfterLast function” on page 181
- “BeforeFirst function” on page 182
- “Delete function” on page 182
- “DeleteAllRows function” on page 224
- “Find function” on page 225
- “FindBegin function” on page 225
- “FindFirst function” on page 225
- “FindLast function” on page 226
- “FindNext function” on page 226
- “FindPrevious function” on page 226
- “First function” on page 182
- “Get function” on page 182
- “GetConnection function” on page 215
- “GetIFace function” on page 216
- “GetRowCount function” on page 182
- “GetSchema function” on page 227
- “GetState function” on page 183
- “GetStreamReader function” on page 183
- “GetStreamWriter function” on page 183
- “GetSuspend function (deprecated)” on page 184
- “Insert function” on page 227
- “InsertBegin function” on page 227
- “IsNull function” on page 184
- “Last function” on page 184
- “Lookup function” on page 228
- “LookupBackward function” on page 228
- “LookupBegin function” on page 228
- “LookupForward function” on page 229
- “Next function” on page 184
- “Previous function” on page 185
- “Relative function” on page 185
- “Release function” on page 216
- “Set function” on page 185
- “SetDefault function” on page 186
- “SetNull function” on page 186
- “SetSuspend function (deprecated)” on page 186
- “TruncateTable function” on page 229
- “Update function” on page 187
- “UpdateBegin function” on page 187

UltraLite_Table_iface class

Represents a table interface.

Syntax

public **UltraLite_Table_iface**

Derived classes

- "UltraLite_Table class" on page 222

Members

All members of UltraLite_Table_iface, including all inherited members.

- "DeleteAllRows function" on page 224
- "Find function" on page 225
- "FindBegin function" on page 225
- "FindFirst function" on page 225
- "FindLast function" on page 226
- "FindNext function" on page 226
- "FindPrevious function" on page 226
- "GetSchema function" on page 227
- "Insert function" on page 227
- "InsertBegin function" on page 227
- "Lookup function" on page 228
- "LookupBackward function" on page 228
- "LookupBegin function" on page 228
- "LookupForward function" on page 229
- "TruncateTable function" on page 229

DeleteAllRows function

Deletes all rows from table.

Syntax

bool **UltraLite_Table_iface::DeleteAllRows()**

Returns

- True on success.
- False on failure. For example, the table is not open, or there was a SQL error, and so on.

Remarks

In some applications, you may want to delete all rows from a table before downloading a new set of data into the table. If you set the stop sync property on the connection, the deleted rows are not synchronized.

Any uncommitted inserts from other connections are not deleted. Also, any uncommitted deletes from other connections are not deleted, if the other connection does a rollback after it calls "DeleteAllRows function" on page 224.

If this table has been opened without an index then it is considered read only and data cannot be deleted.

Find function

This function is the equivalent to "FindFirst function" on page 225. It performs an exact match lookup based on the current index scanning forward through the table.

Syntax

bool **UltraLite_Table_iface::Find(**
  ul_column_num *ncols*
)

Parameters

- **ncols** For composite indexes, the number of columns to use in the lookup.

FindBegin function

Prepares to perform a new Find on a table by entering find mode.

Syntax

bool **UltraLite_Table_iface::FindBegin()**

Remarks

You may only set columns in the index that the table was opened with. If the table was opened without an index, this method cannot be called.

FindFirst function

Does an exact match lookup based on the current index scanning forward through the table.

Syntax

bool **UltraLite_Table_iface::FindFirst(**
  ul_column_num *ncols*
)

Parameters

- **ncols** For composite indexes, the number of columns to use in the lookup.

Remarks

To specify the value to search for, set the column value for each column in the index. The cursor is positioned on the first row that exactly matches the index value. If no row matches the index value, the cursor position is AfterLast() and the function returns false.

FindLast function

Does an exact match lookup based on the current index scanning backward through the table.

Syntax

bool **UltraLite_Table_iface::FindLast(**
  ul_column_num *ncols*
)

Parameters

- **ncols** For composite indexes, the number of columns to use in the lookup.

Remarks

To specify the value to search for, set the column value for each column in the index. The cursor is left positioned on the first row that exactly matches the index value. If no row matches the index value, the cursor position is BeforeFirst() and the function returns false.

FindNext function

Gets the next row that exactly matches the index.

Syntax

bool **UltraLite_Table_iface::FindNext(**
  ul_column_num *ncols*
)

Parameters

- **ncols** For composite indexes, the number of columns to use in the lookup.

Returns

False if no more rows match the index.

FindPrevious function

Gets the previous row that exactly matches the index.

Syntax

bool **UltraLite_Table_iface::FindPrevious(**
  ul_column_num *ncols*
)

Parameters

- **ncols** For composite indexes, the number of columns to use in the lookup.

Returns

False if no more rows match the index. In this case the cursor is positioned before the first row.

GetSchema function

Gets a schema object for this table.

Syntax

UltraLite_TableSchema * **UltraLite_Table_iface::GetSchema()**

Insert function

Inserts a new row into the table.

Syntax

bool **UltraLite_Table_iface::Insert()**

Remarks

The table must be in insert mode for this operation to succeed. Use “InsertBegin function” on page 227 to switch to insert mode.

InsertBegin function

Selects insert mode for setting columns.

Syntax

bool **UltraLite_Table_iface::InsertBegin()**

Remarks

All columns may be modified in this mode. If this table has been opened without an index, then the data is considered read only and rows cannot be inserted.

Lookup function

This function is the equivalent to LookupForward. It performs a lookup based on the current index scanning forward through the table.

Syntax

bool **UltraLite_Table_iface::Lookup(**
  ul_column_num *ncols*
)

Parameters

- **ncols** For composite indexes, the number of columns to use in the lookup.

Remarks

If resulting cursor position is AfterLast(), the return value is false.

LookupBackward function

Does a lookup based on the current index scanning backward through the table.

Syntax

bool **UltraLite_Table_iface::LookupBackward(**
  ul_column_num *ncols*
)

Parameters

- **ncols** For composite indexes, the number of columns to use in the lookup.

Returns

If resulting cursor position is BeforeFirst(), the return value is false.

Remarks

To specify the value to search for, set the column value for each column in the index. The cursor is positioned on the last row that matches or is less than the index value. For composite indexes, ncols specifies the number of columns to use in the lookup.

LookupBegin function

Prepares to perform a new Find on a table by entering lookup mode.

Syntax

bool **UltraLite_Table_iface::LookupBegin()**

Remarks

You may only set columns in the index that the table was opened with. If the table was opened without an index, this method cannot be called.

LookupForward function

Does a lookup based on the current index scanning forward through the table.

Syntax

```
bool UltraLite_Table_iface::LookupForward(
  ul_column_num ncols
)
```

Parameters

- **ncols** For composite indexes, the number of columns to use in the lookup.

Returns

If resulting cursor position is AfterLast(), the return value is false.

Remarks

To specify the value to search for, set the column value for each column in the index. The cursor is positioned on the last row that matches or is less than the index value. For composite indexes, ncols specifies the number of columns to use in the lookup.

TruncateTable function

Truncates the table and temporarily activates STOP SYNCHRONIZATION DELETE.

Syntax

```
bool UltraLite_Table_iface::TruncateTable()
```

Remarks

If this table has been opened without an index then it is considered read only and data cannot be deleted.

UltraLite_TableSchema class

Represents a table schema.

Syntax

public **UltraLite_TableSchema**

Base classes

- "UltraLite_SQLObject_iface class" on page 215
- "UltraLite_TableSchema_iface class" on page 232
- "UltraLite_RowSchema_iface class" on page 210

Members

All members of UltraLite_TableSchema, including all inherited members.

- “AddRef function” on page 215
- “GetBaseColumnName function” on page 210
- “GetColumnCount function” on page 211
- “GetColumnDefault function” on page 232
- “GetColumnID function” on page 211
- “GetColumnName function” on page 211
- “GetColumnPrecision function” on page 212
- “GetColumnScale function” on page 213
- “GetColumnSize function” on page 213
- “GetColumnSQLName function” on page 212
- “GetColumnSQLType function” on page 212
- “GetColumnType function” on page 214
- “GetConnection function” on page 215
- “GetGlobalAutoincPartitionSize function” on page 233
- “GetID function” on page 233
- “GetIFace function” on page 216
- “GetIndexCount function” on page 233
- “GetIndexName function” on page 234
- “GetIndexSchema function” on page 234
- “GetName function” on page 235
- “GetOptimalIndex function” on page 235
- “GetPrimaryKey function” on page 235
- “GetPublicationPredicate function” on page 235
- “GetUploadUnchangedRows function” on page 236
- “InPublication function” on page 236
- “IsColumnAutoinc function” on page 237
- “IsColumnCurrentDate function” on page 237
- “IsColumnCurrentTime function” on page 237
- “IsColumnCurrentTimestamp function” on page 238
- “IsColumnGlobalAutoinc function” on page 238
- “IsColumnInIndex function” on page 239
- “IsColumnNewUUID function” on page 239
- “IsColumnNullable function” on page 240
- “IsNeverSynchronized function” on page 240
- “Release function” on page 216

UltraLite_TableSchema_iface class

The TableSchema interface.

Syntax

public **UltraLite_TableSchema_iface**

Derived classes

- "UltraLite_TableSchema class" on page 230

Members

All members of UltraLite_TableSchema_iface, including all inherited members.

- "GetColumnDefault function" on page 232
- "GetGlobalAutoincPartitionSize function" on page 233
- "GetID function" on page 233
- "GetIndexCount function" on page 233
- "GetIndexName function" on page 234
- "GetIndexSchema function" on page 234
- "GetName function" on page 235
- "GetOptimalIndex function" on page 235
- "GetPrimaryKey function" on page 235
- "GetPublicationPredicate function" on page 235
- "GetUploadUnchangedRows function" on page 236
- "InPublication function" on page 236
- "IsColumnAutoinc function" on page 237
- "IsColumnCurrentDate function" on page 237
- "IsColumnCurrentTime function" on page 237
- "IsColumnCurrentTimestamp function" on page 238
- "IsColumnGlobalAutoinc function" on page 238
- "IsColumnInIndex function" on page 239
- "IsColumnNewUUID function" on page 239
- "IsColumnNullable function" on page 240
- "IsNeverSynchronized function" on page 240

GetColumnDefault function

Gets the default value for the column if it exists.

Syntax

ULValue **UltraLite_TableSchema_iface::GetColumnDefault(**
  const ULValue & *column_id*
)

Parameters

- **column_id** The ID number of the column. The value must be a 1-based ordinal number. The first column in the table has an ID value of one.

Returns

- The default contained as a string.
- Is empty if the column does not contain a default value.

Remarks

Sets SQLE_COLUMN_NOT_FOUND if the column name does not exist.

GetGlobalAutoincPartitionSize function

Gets the partition size.

Syntax

bool **UltraLite_TableSchema_iface::GetGlobalAutoincPartitionSize(**
  const ULValue & *column_id*,
  ul_u_big * *size*
)

Parameters

- **column_id** The ID number of the column. The value must be a 1-based ordinal number. The first column in the table has an ID value of one.
- **size** An output parameter. The partition size for the column. All global autoincrement columns in a given table share the same global autoincrement partition.

Returns

The partition size for a global autoincrement column.

GetID function

Gets the table ID.

Syntax

ul_table_num **UltraLite_TableSchema_iface::GetID()**

GetIndexCount function

Gets the number of indexes in the table.

Syntax

ul_index_num **UltraLite_TableSchema_iface::GetIndexCount()**

Returns

The number of indexes in the table.

Remarks

Index IDs and counts may change during a schema upgrade. To correctly identify an index, access it by name or refresh any cached IDs and counts after a schema upgrade.

GetIndexName function

Gets the index name given its 1-based ID.

Syntax

ULValue **UltraLite_TableSchema_iface::GetIndexName(**
  ul_index_num *index_id*
)

Parameters

- **index_id** A 1-based ordinal number.

Returns

A "ULValue class" on page 241 object is empty if the index does not exist.

Remarks

Index IDs and counts may change during a schema upgrade. To correctly identify an index, access it by name or refresh any cached IDs and counts after a schema upgrade.

GetIndexSchema function

Gets an IndexSchema object with the given name or ID.

Syntax

UltraLite_IndexSchema * **UltraLite_TableSchema_iface::GetIndexSchema(**
  const ULValue & *index_id*
)

Parameters

- **index_id** The name or ID number identifying the index.

Returns

UL_NULL if the index does not exist.

GetName function

Gets the name of the table.

Syntax

ULValue **UltraLite_TableSchema_iface::GetName()**

Returns

The name of the table as a string.

GetOptimalIndex function

Determines the best index to use for searching for a column value.

Syntax

ULValue **UltraLite_TableSchema_iface::GetOptimalIndex(**
 const ULValue & *column_id*
)

Parameters

- **column_id** The ID number of the column. The value must be a 1-based ordinal number. The first column in the table has an ID value of one.

Returns

The name of the index.

GetPrimaryKey function

Gets the primary key for the table.

Syntax

UltraLite_IndexSchema * **UltraLite_TableSchema_iface::GetPrimaryKey()**

GetPublicationPredicate function

Gets the publication predicate as a string.

Syntax

ULValue **UltraLite_TableSchema_iface::GetPublicationPredicate(**
 const ULValue & *publication_name*
)

Parameters

- **publication_name** The name of the publication.

Returns

The publication predicate string for the specified publication.

Remarks

Sets SQLE_PUBLICATION_NOT_FOUND if the publication does not exist.

GetUploadUnchangedRows function

Checks whether the database has been configured to upload rows that have not changed.

Syntax

```
bool UltraLite_TableSchema_iface::GetUploadUnchangedRows()
```

Returns

- True if the table is marked to always upload all rows during synchronization.
- False if the table is marked to upload only changed rows.

Remarks

Tables set to upload unchanged and changed rows are sometimes referred to as allsync tables.

InPublication function

Checks whether the table is contained in the named publication.

Syntax

```
bool UltraLite_TableSchema_iface::InPublication(
  const ULValue & publication_name
)
```

Parameters

- **publication_name** The name of the publication.

Returns

- True if the table is contained in the publication.
- False if the table is not in the publication.

Remarks

Sets SQLE_PUBLICATION_NOT_FOUND if the publication does not exist.

IsColumnAutoinc function

Checks whether the specified column's default is set to autoincrement.

Syntax

```
bool UltraLite_TableSchema_iface::IsColumnAutoinc(
  const ULValue & column_id
)
```

Parameters

- **column_id** The ID number of the column. The value must be a 1-based ordinal number. The first column in the table has an ID value of one.

Returns

- True if the column default is set to be autoincremented.
- False if the column is not autoincremented.

Remarks

Sets SQLE_COLUMN_NOT_FOUND if the column name does not exist.

IsColumnCurrentDate function

Checks whether the specified column's default is set to the current date.

Syntax

```
bool UltraLite_TableSchema_iface::IsColumnCurrentDate(
  const ULValue & column_id
)
```

Parameters

- **column_id** The ID number of the column. The value must be a 1-based ordinal number. The first column in the table has an ID value of one.

Returns

- True if the column has a current date default.
- False if the column does not default to the current date.

IsColumnCurrentTime function

Checks whether the specified column's default is set to the current time.

Syntax

bool **UltraLite_TableSchema_iface::IsColumnCurrentTime(**
 const ULValue & *column_id*
)

Parameters

- **column_id** The ID number of the column. The value must be a 1-based ordinal number. The first column in the table has an ID value of one.

Returns

- True if the column has a current time default.
- False if it does not.

IsColumnCurrentTimestamp function

Checks whether the specified column's default is set to the current timestamp.

Syntax

bool **UltraLite_TableSchema_iface::IsColumnCurrentTimestamp(**
 const ULValue & *column_id*
)

Parameters

- **column_id** The ID number of the column. The value must be a 1-based ordinal number. The first column in the table has an ID value of one.

Returns

- True if the column has a current timestamp default.
- False if it does not.

IsColumnGlobalAutoinc function

Checks whether the specified column's default is set to autoincrement.

Syntax

bool **UltraLite_TableSchema_iface::IsColumnGlobalAutoinc(**
 const ULValue & *column_id*
)

Parameters

- **column_id** The ID number of the column. The value must be a 1-based ordinal number. The first column in the table has an ID value of one.

Returns

- True if the column is autoincremented.
- False if it is not autoincremented.

Remarks

SQLE_COLUMN_NOT_FOUND is set if the column name does not exist.

IsColumnInIndex function

Checks whether the table is contained in the named index.

Syntax

bool **UltraLite_TableSchema_iface::IsColumnInIndex(**
  const ULValue & *column_id*,
  const ULValue & *index_id*
)

Parameters

- **column_id**   A 1-based ordinal number identifying the column. You can get the column_id by calling "GetColumnCount function" on page 211.
- **index_id**   A 1-based ordinal number identifying the index. You can get the number of indexes in a table by calling "GetIndexCount function" on page 233.

Returns

- True if the column is contained in the index.
- False if the column is not contained in the index.

Remarks

Sets SQLE_COLUMN_NOT_FOUND if the column name does not exist.

Sets SQLE_INDEX_NOT_FOUND if the index does not exist.

IsColumnNewUUID function

Checks whether the specified column's default is set to a new UUID.

Syntax

bool **UltraLite_TableSchema_iface::IsColumnNewUUID(**
  const ULValue & *column_id*
)

Parameters

- **column_id** The ID number of the column. The value must be a 1-based ordinal number. The first column in the table has an ID value of one.

Returns

- True if the column has a new UUID default.
- False if the column does not default to a new UUID.

IsColumnNullable function

Checks whether the specified column is nullable.

Syntax

bool **UltraLite_TableSchema_iface::IsColumnNullable(**
 const ULValue & *column_id*
)

Parameters

- **column_id** The ID number of the column. The value must be a 1-based ordinal number. The first column in the table has an ID value of one. The specified column is the first column in the index, but the index may have more than one column.

Returns

- True if the column is nullable.
- False if it is not nullable.

Remarks

Sets SQLE_COLUMN_NOT_FOUND if the column name does not exist.

IsNeverSynchronized function

Checks whether the table is marked as never being synchronized.

Syntax

bool **UltraLite_TableSchema_iface::IsNeverSynchronized()**

Returns

- True if the table is omitted from synchronization. Tables marked as never being synchronized are never synchronized, even if they are included in a publication. These tables are sometimes referred to as nosync tables.
- False if the table is included as a synchronizable table.

ULValue class

Syntax

public **ULValue**

Remarks

“ULValue class” on page 241 class.

The “ULValue class” on page 241 class is a wrapper for the data types stored in an UltraLite cursor. This class allows you to store data without having to worry about the data type, and is used to pass values to and from the UltraLite C++ Component.

“ULValue class” on page 241 contains many constructors and cast operators, so you can use “ULValue class” on page 241 seamlessly (in most cases) without explicitly instantiating a “ULValue class” on page 241.

You can construct the object or assign it from any basic C++ data type. You can also cast it into any basic C++ data type.

```
     x( 5 );        ULValue// Example of ULValue's constructor
     y = 5;         ULValue// Example of ULValue's assignment operator
    int z = y;             // Example of ULValue's cast operator
```

This sample works for strings as well:

```
     x( UL_TEXT( ULValue"hello" ) );
     y = UL_TEXT( ULValue"hello" );
    y.( buffer, BUFFER_LEN );    GetString// NOTE, there is no cast operator
```

You do not need to explicitly construct a “ULValue class” on page 241 object as the compiler often does this automatically. For example, to fetch a value from a column, you can use the following:

```
    int x = table->Get( UL_TEXT( "my_column" ) );
```

The table->Get() call returns a “ULValue class” on page 241 object. C++ automatically calls the cast operator to convert it to an integer. Similarly, the table->Get() call takes a “ULValue class” on page 241 parameter as the column identifier. This determines which column to fetch. C++ automatically converts the "my_column" literal string into a “ULValue class” on page 241 object.

Members

All members of ULValue, including all inherited members.

- "GetBinary function" on page 243
- "GetBinary function" on page 243
- "GetBinaryLength function" on page 244
- "GetCombinedStringItem function" on page 244
- "GetCombinedStringItem function" on page 244
- "GetString function" on page 245
- "GetString function" on page 245
- "GetStringLength function" on page 246
- "InDatabase function" on page 246
- "IsNull function" on page 246
- "bool operator" on page 255
- "DECL_DATETIME operator" on page 254
- "double operator" on page 255
- "float operator" on page 255
- "GUID operator" on page 254
- "int operator" on page 255
- "long operator" on page 255
- "short operator" on page 255
- "ul_s_big operator" on page 256
- "ul_u_big operator" on page 256
- "unsigned char operator" on page 256
- "unsigned int operator" on page 256
- "unsigned long operator" on page 256
- "unsigned short operator" on page 257
- "operator= function" on page 257
- "SetBinary function" on page 247
- "SetString function" on page 247
- "SetString function" on page 247
- "StringCompare function" on page 248
- "ULValue function" on page 248
- "ULValue function" on page 248
- "ULValue function" on page 249
- "ULValue function" on page 249
- "ULValue function" on page 249
- "ULValue function" on page 250
- "ULValue function" on page 250
- "ULValue function" on page 250
- "ULValue function" on page 250
- "ULValue function" on page 251
- "ULValue function" on page 251
- "ULValue function" on page 251
- "ULValue function" on page 252
- "ULValue function" on page 252
- "ULValue function" on page 252

- "ULValue function" on page 252
- "ULValue function" on page 253
- "ULValue function" on page 253
- "ULValue function" on page 253
- "ULValue function" on page 254
- "ULValue function" on page 254
- "~ULValue function" on page 257

GetBinary function

Retrieves the current value into a binary buffer, casting as required.

Syntax

```
void ULValue::GetBinary(
  p_ul_binary bin,
  size_t len
)
```

Parameters

- **bin** The binary structure to receive bytes.
- **len** The length of the buffer.

Remarks

If the buffer is too small, then the value is truncated. Up to len characters are copied to the given buffer.

GetBinary function

Retrieves the current value into a binary buffer, casting as required If the buffer is too small, then the value is truncated.

Syntax

```
void ULValue::GetBinary(
  ul_byte * dst,
  size_t len,
  size_t * retr_len
)
```

Parameters

- **dst** The buffer to receive bytes.
- **len** The length of the buffer.
- **retr_len** An output parameter. The actual number of bytes returned.

Remarks

Up to len bytes are copied to the given buffer. The number of bytes actually copied is returned in retr_len.

GetBinaryLength function

Gets the length of a Binary value.

Syntax

size_t **ULValue::GetBinaryLength()**

Returns

The number of bytes necessary to hold the binary value returned by "GetBinary function" on page 243.

GetCombinedStringItem function

Retrieves parts of a combined name into a string buffer, casting as required.

Syntax

void **ULValue::GetCombinedStringItem(**
  ul_u_short *selector*,
  char * *dst*,
  size_t *len*
)

Parameters

- **selector** Selected internal value.
- **dst** The buffer to receive string value.
- **len** The length, in bytes, of dst.

Remarks

If the value is not combined, an empty string is copied. The output string is always null-terminated. If the buffer is too small, then the value is truncated. Up to len characters are copied to the given buffer, including the null terminator.

GetCombinedStringItem function

Gets selected portion of a combined String value.

Syntax

void **ULValue::GetCombinedStringItem(**
  ul_u_short *selector*,
  ul_wchar * *dst*,

```
  size_t len
)
```

Parameters

- **selector** Selected internal value.
- **dst** The buffer to receive string value.
- **len** The length, in wide chars, of dst.

GetString function

Retrieves the current value into a string buffer, casting as required.

Syntax

```
void ULValue::GetString(
  char * dst,
  size_t len
)
```

Parameters

- **dst** The buffer to receive string value.
- **len** The length, in bytes, of dst.

Remarks

The output string is always null-terminated. If the buffer is too small then the value is truncated. Up to len characters are copied to the given buffer, including the null terminator.

GetString function

Retrieves the current value into a string buffer, casting as required.

Syntax

```
void ULValue::GetString(
  ul_wchar * dst,
  size_t len
)
```

Parameters

- **dst** The buffer to receive string value.
- **len** The length, in wide chars, of dst.

GetStringLength function

Gets the length of a String.

Syntax

size_t **ULValue::GetStringLength(**
  bool *fetch_as_chars*
)

Parameters

- **fetch_as_chars**   False for byte length, true for char length.

Returns

The number of bytes or characters required to hold the string returned by one of the "GetString function" on page 245 methods, not including the null-terminator.

Example

Intended usage is as follows:

```
len = v.GetStringLength();
dst = new char[ len + 1 ];
( dst, len + 1 ); GetString
```

For wide character applications the usage is:

```
len = v.GetStringLength( true );
dst = new ul_wchar[ len + 1 ];
( dst, len + 1 ); GetString
```

InDatabase function

Checks if value is in the database.

Syntax

bool **ULValue::InDatabase()**

Returns

- True if this object is referencing a cursor field.
- False if it is not.

IsNull function

Checks if the "ULValue class" on page 241 object is empty.

Syntax

bool **ULValue::IsNull()**

Returns

- True if this object is either an empty "ULValue class" on page 241 object, or if it references a cursor field that is set to NULL.
- False otherwise.

SetBinary function

Sets the value to reference the binary buffer provided.

Syntax

void **ULValue::SetBinary(**
  ul_byte * *src*,
  size_t *len*
)

Parameters

- **src** A buffer of bytes.
- **len** The length of the buffer.

Remarks

No bytes are copied from the provided buffer until the value is used.

SetString function

Casts a "ULValue class" on page 241 to a string.

Syntax

void **ULValue::SetString(**
  const char * *val*,
  size_t *len*
)

Parameters

- **val** A pointer to the null-terminated string representation of this "ULValue class" on page 241.
- **len** The length of the string.

SetString function

Casts a "ULValue class" on page 241 to a UNICODE string.

Syntax

```
void ULValue::SetString(
  const ul_wchar * val,
  size_t len
)
```

Parameters

- **val** A pointer to the null-terminated unicode string representation of this "ULValue class" on page 241.
- **len** The length of the string.

StringCompare function

Compares strings, or string representations of "ULValue class" on page 241 objects.

Syntax

```
ul_compare ULValue::StringCompare(
  const ULValue & value
)
```

Parameters

- **value** The comparison string.

Returns

- 0 if the strings are equal.
- -1 if the current value compares less than value.
- 1 if the current value compares greater than value.
- -3 if the sqlca of either "ULValue class" on page 241 object is not set.
- -2 if the string representation of either "ULValue class" on page 241 object is UL_NULL.

ULValue function

Constructs a "ULValue class" on page 241.

Syntax

```
ULValue::ULValue()
```

ULValue function

Constructs a "ULValue class" on page 241, copying from an existing one.

Syntax

```
 ULValue::ULValue(
  const ULValue & vSrc
)
```

Parameters

- **vSrc** A value to be treated as a "ULValue class" on page 241.

ULValue function

Constructs a "ULValue class" on page 241 from a bool.

Syntax

```
 ULValue::ULValue(
  bool val
)
```

Parameters

- **val** A boolean value to be treated as a "ULValue class" on page 241.

ULValue function

Constructs a "ULValue class" on page 241 from a short.

Syntax

```
 ULValue::ULValue(
  short val
)
```

Parameters

- **val** A short value to be treated as a "ULValue class" on page 241.

ULValue function

Constructs a "ULValue class" on page 241 from a long.

Syntax

```
 ULValue::ULValue(
  long val
)
```

Parameters

- **val** A long value to be treated as a "ULValue class" on page 241.

ULValue function

Construct a "ULValue class" on page 241 from an int.

Syntax

```
ULValue::ULValue(
  int val
)
```

Parameters

- **val** An INTEGER value to be treated as a "ULValue class" on page 241.

ULValue function

Constructs a "ULValue class" on page 241 from an unsigned INTEGER.

Syntax

```
ULValue::ULValue(
  unsigned int val
)
```

Parameters

- **val** An unsigned INTEGER value to be treated as a "ULValue class" on page 241.

ULValue function

Constructs a "ULValue class" on page 241 from a FLOAT.

Syntax

```
ULValue::ULValue(
  float val
)
```

Parameters

- **val** A FLOAT value to be treated as a "ULValue class" on page 241.

ULValue function

Constructs a "ULValue class" on page 241 from a DOUBLE.

Syntax

```
ULValue::ULValue(
 double val
)
```

Parameters

- **val** A DOUBLE value to be treated as a "ULValue class" on page 241.

ULValue function

Constructs a "ULValue class" on page 241 from an unsigned CHAR.

Syntax

```
ULValue::ULValue(
 unsigned char val
)
```

Parameters

- **val** An unsigned CHAR to be treated as a "ULValue class" on page 241.

ULValue function

Construct a "ULValue class" on page 241 from an unsigned SHORT.

Syntax

```
ULValue::ULValue(
 unsigned short val
)
```

Parameters

- **val** An unsigned SHORT to be treated as a "ULValue class" on page 241.

ULValue function

Construct a "ULValue class" on page 241 from an unsigned LONG.

Syntax

```
ULValue::ULValue(
 unsigned long val
)
```

Parameters

- **val** An unsigned LONG to be treated as a "ULValue class" on page 241.

ULValue function

Construct a "ULValue class" on page 241 from a ul_u_big.

Syntax

```
ULValue::ULValue(
 const ul_u_big & val
)
```

Parameters

- **val** A ul_u_big value to be treated as a "ULValue class" on page 241.

ULValue function

Construct a "ULValue class" on page 241 from a ul_s_big.

Syntax

```
ULValue::ULValue(
 const ul_s_big & val
)
```

Parameters

- **val** A ul_s_big value to be treated as a "ULValue class" on page 241.

ULValue function

Construct a "ULValue class" on page 241 from a ul_binary.

Syntax

```
ULValue::ULValue(
 const p_ul_binary val
)
```

Parameters

- **val** A ul_binary value to be treated as a "ULValue class" on page 241.

ULValue function

Construct a "ULValue class" on page 241 from a datetime.

Syntax

```
 ULValue::ULValue(
  DECL_DATETIME & val
)
```

Parameters

- **val** A DATETIME value to be treated as a "ULValue class" on page 241.

ULValue function

Construct a "ULValue class" on page 241 from a STRING.

Syntax

```
 ULValue::ULValue(
  const char * val
)
```

Parameters

- **val** A pointer to a string to be treated as a "ULValue class" on page 241.

ULValue function

Construct a "ULValue class" on page 241 from a UNICODE string.

Syntax

```
 ULValue::ULValue(
  const ul_wchar * val
)
```

Parameters

- **val** A pointer to a UNICODE string to be treated as a "ULValue class" on page 241.

ULValue function

Construct a "ULValue class" on page 241 from a buffer of characters.

Syntax

```
 ULValue::ULValue(
  const char * val,
  size_t len
)
```

Parameters

- **val** A buffer holding the string to be treated as a "ULValue class" on page 241.

- **len** The length of the buffer.

ULValue function

Construct a “ULValue class” on page 241 from a buffer of unicode characters.

Syntax

ULValue::ULValue(
const ul_wchar * *val*,
size_t *len*
)

Parameters

- **val** A buffer holding the string to be treated as a “ULValue class” on page 241.
- **len** The length of the buffer.

ULValue function

Construct a “ULValue class” on page 241 from a GUID structure.

Syntax

ULValue::ULValue(
GUID & *val*
)

Parameters

- **val** A GUID value to be treated as a “ULValue class” on page 241.

DECL_DATETIME operator

Cast a “ULValue class” on page 241 to a datetime.

Syntax

ULValue::operator DECL_DATETIME()

GUID operator

Cast a “ULValue class” on page 241 to a GUID structure.

Syntax

ULValue::operator GUID()

bool operator

Cast a "ULValue class" on page 241 to a boolean.

Syntax

ULValue::operator bool()

double operator

Cast a "ULValue class" on page 241 to a double.

Syntax

ULValue::operator double()

float operator

Cast a "ULValue class" on page 241 to a float.

Syntax

ULValue::operator float()

int operator

Cast a "ULValue class" on page 241 to an int.

Syntax

ULValue::operator int()

long operator

Cast a "ULValue class" on page 241 to a long.

Syntax

ULValue::operator long()

short operator

Cast a "ULValue class" on page 241 to a short.

Syntax

ULValue::operator short()

ul_s_big operator

Cast a "ULValue class" on page 241 to a signed bigint.

Syntax

ULValue::operator ul_s_big()

ul_u_big operator

Cast a "ULValue class" on page 241 to an unsigned bigint.

Syntax

ULValue::operator ul_u_big()

unsigned char operator

Cast a "ULValue class" on page 241 to a char.

Syntax

ULValue::operator unsigned char()

unsigned int operator

Cast a "ULValue class" on page 241 to an unsigned int.

Syntax

ULValue::operator unsigned int()

unsigned long operator

Cast a "ULValue class" on page 241 to an unsigned long.

Syntax

ULValue::operator unsigned long()

unsigned short operator

Cast a "ULValue class" on page 241 to an unsigned short.

Syntax

ULValue::operator unsigned short()

operator= function

Override the = operator for ULValues.

Syntax

ULValue & **ULValue::operator=(**
  const ULValue & *other*
)

Parameters

- **other** The value to be assigned to a "ULValue class" on page 241.

~ULValue function

The destructor for "ULValue class" on page 241.

Syntax

ULValue::~ULValue()

Embedded SQL API reference

Contents

This section lists functions that support UltraLite functionality in embedded SQL applications.

For general information about SQL statements that can be used, see “Developing applications using Embedded SQL” on page 31.

Use the EXEC SQL INCLUDE SQLCA command to include prototypes for the functions in this chapter.

db_fini function

Frees resources used by the UltraLite runtime library.

Syntax

unsigned short **db_fini(**
SQLCA * *sqlca*
);

Returns

- 0 if an error occurs during processing. The error code is set in SQLCA.
- Non-zero if there are no errors.

Remarks

You must not make any other UltraLite library call or execute any embedded SQL command after db_fini is called.

Call db_fini once for each SQLCA being used.

See also

- "db_init function" on page 262

db_init function

Initializes the UltraLite runtime library.

Syntax

unsigned short **db_init(**
SQLCA * *sqlca*
);

Returns

- 0 if an error occurs during processing (for example, during initialization of the persistent store). The error code is set in SQLCA.
- Non-zero if there are no errors. You can begin using embedded SQL commands and functions.

Remarks

You must call this function before you make any other UltraLite library call, and before you execute any embedded SQL command.

In most cases, you should only call this function once, passing the address of the global sqlca variable (as defined in the *sqlca.h* header file). If you have multiple execution paths in your application, you can use more than one db_init call, as long as each one has a separate sqlca pointer. This separate SQLCA pointer can be a user-defined one, or could be a global SQLCA that has been freed using db_fini.

In multi-threaded applications, each thread must call db_init to obtain a separate SQLCA. Carry out subsequent connections and transactions that use this SQLCA on a single thread.

Initializing the SQLCA also resets any settings from previously called ULEnable functions. If you re-initialize a SQLCA, you must issue any ULEnable functions the application requires.

See also

- "db_fini function" on page 261

db_start_database function

Starts a database if the database is not already running.

Syntax

unsigned int **db_start_database(**
SQLCA * *sqlca***,**
char * *parms*
);

Parameters

sqlca A pointer to a SQLCA structure.

parms A null-terminated string containing a semicolon-delimited list of parameter settings, each of the form **KEYWORD**=*value*. Typically, only a file name is required. For example:

```
"DBF=c:\\db\\mydatabase.db"
```

Returns

- True if the database was already running or was successfully started. In this case, SQLCODE is set to 0.
- Error information is also returned in the SQLCA.

Remarks

Required when developing applications that combine embedded SQL and the C++ component.

See also

- "UltraLite connection parameters" [*UltraLite - Database Management and Reference*]
- "Initializing the SQL Communications Area" on page 34

db_stop_database function

Stop a database.

Syntax

unsigned int **db_stop_database(**
SQLCA * *sqlca***,**
char * *parms*
);

Parameters

sqlca A pointer to a SQLCA structure.

parms A null-terminated string containing a semicolon-delimited list of parameter settings, each of the form **KEYWORD**=*value*. Typically, only a database file name is needed. For example,

```
"DBF=c:\\db\\mydatabase.db"
```

Returns

- TRUE if there were no errors.

Remarks

This function is not commonly needed, as UltraLite automatically stops the database when all connections are closed. However, this function may be useful when developing applications that combine embedded SQL and the C++ component.

This function does not stop a database that has existing connections.

See also

- "UltraLite connection parameters" [*UltraLite - Database Management and Reference*]
- "Initializing the SQL Communications Area" on page 34

ULChangeEncryptionKey function

Changes the encryption key for an UltraLite database.

Syntax

ul_bool **ULChangeEncryptionKey(**
SQLCA **sqlca*,
ul_char **new_key*
);

Remarks

Applications that call this function must first ensure that the user has either synchronized the database or created a reliable backup copy of the database. It is important to have a reliable backup of the database because ULChangeEncryptionKey is an operation that must run to completion. When the database encryption key is changed, every row in the database is first decrypted with the old key and then encrypted with the new key and rewritten. *This operation is not recoverable.* If the encryption change operation does not complete, the database is left in an invalid state and you cannot access it again.

See also

- "Encrypting data" on page 54

ULCheckpoint function

Performs a checkpoint operation: flushing any pending committed transactions to the database. Any current transaction is not committed by calling ULCheckpoint. The ULCheckpoint function is used in conjunction with deferring automatic transaction checkpoints as a performance enhancement. For more information, see “Flushing single or grouped transactions” [*UltraLite - Database Management and Reference*].

Syntax

void **ULCheckpoint(**
SQLCA * *sqlca*
);

Remarks

The ULCheckpoint function ensures that all pending committed transactions have been written to the database storage.

See also

- “UltraLite transaction processing” [*UltraLite - Database Management and Reference*]

ULClearEncryptionKey function

Clears the encryption key.

Syntax

ul_bool **ULClearEncryptionKey(**
ul_u_long * *creator***,**
ul_u_long * *feature-num* **);**

Parameters

creator A pointer to the creator ID of the feature that holds the encryption key. The default value is NULL.

feature-num A pointer to the feature number that holds the encryption key. If feature-num is NULL, the application uses the UltraLite default value of 100.

Remarks

On the Palm OS the encryption key is saved in dynamic memory as a Palm feature. Features are indexed by creator and a feature number.

See also

- "ULRetrieveEncryptionKey function" on page 285
- "ULSaveEncryptionKey function" on page 288

ULCountUploadRows function

Counts the number of rows that need to be uploaded for synchronization.

Syntax

ul_u_long **ULCountUploadRows (**
SQLCA * *sqlca***,**
ul_char *pub-list***,**
ul_u_long *threshold*
);

Parameters

sqlca A pointer to the SQLCA.

pub-list A string containing a comma-separated list of publications to check. An empty string (UL_SYNC_ALL macro) implies all tables except tables marked as "no sync". A string containing just an asterisk (UL_SYNC_ALL_PUBS macro) implies all tables referred to in any publication. Some tables may not be part of any publication and are not included if the pub-list string is "*".

threshold Determines the maximum number of rows to count, thereby limiting the amount of time taken by the call.

- A threshold of 0 corresponds to no limit (that is, count all rows that need to be synchronized).
- A threshold of 1 can be used to quickly determine if any rows need to be synchronized.

Returns

- The number of rows that need to be synchronized, either in a specified set of publications or in the whole database.

Remarks

Use this function to prompt users to synchronize.

Example

The following call checks the entire database for the number of rows to be synchronized:

```
count = ULCountUploadRows( sqlca, UL_SYNC_ALL_PUBS, 0 );
```

The following call checks publications PUB1 and PUB2 for a maximum of 1000 rows:

```
count = ULCountUploadRows( sqlca, UL_TEXT("PUB1,PUB2"), 1000 );
```

The following call checks to see if any rows need to be synchronized in publications PUB1 and PUB2:

```
count = ULCountUploadRows( sqlca, UL_TEXT("PUB1,PUB2"), 1 );
```

See Also

- “UL_SYNC_ALL macro” on page 125
- “UL_SYNC_ALL_PUBS macro” on page 126

ULDropDatabase function

Deletes an UltraLite database file and any associated temporary or work files.

Syntax

ul_bool **ULDropDatabase (**
SQLCA * *sqlca*,
ul_char * *store-parms*
);

Parameters

sqlca A pointer to the SQLCA.

store-parms A null-terminated connection string containing a semicolon-delimited list of parameter settings, each of the form **KEYWORD**=*value*.

Returns

- **ul_true** The database file was successfully deleted.
- **ul_false** The database file could not be deleted. A detailed error message is defined by the sqlcode field in the SQLCA. The usual reason for failure is an incorrect file name or access to the file is denied (perhaps because an application has the file open).

Remarks

Only call this function:

- when there is no open database connection
- before calling db_init or after calling db_fini.

On the Palm OS, only call this function:

- when not connected to the database
- after calling any ULEnable.

Caution
This function deletes the database file and all data in it. This operation is not recoverable. Therefore, use this function with care.

Example

The following call deletes the UltraLite database file *myfile.udb*.

```
if( ULDropDatabase(&sqlca, UL_TEXT("file_name=myfile.udb") ) ){
    // success
};
```

ULExecuteNextSQLPassthroughScript

Executes the next SQL Passthrough script.

Syntax

bool**ULExecuteNextSQLPassthroughScript(**
SQLCA * *sqlca*
)

Parameters

sqlca A pointer to the SQLCA.

Returns

- False if any errors occurred while executing the script.

See also

- "UltraLite global_database_id option" [*UltraLite - Database Management and Reference*]

ULExecuteSQLPassthroughScripts

Executes all the available SQL Passthrough scripts.

Syntax

bool**ULExecuteSQLPassthroughScripts(**
SQLCA * *sqlca*
)

Parameters

sqlca A pointer to the SQLCA.

Returns

- False if any errors occurred while executing the script.

See also

- "UltraLite global_database_id option" [*UltraLite - Database Management and Reference*]

ULGetDatabaseID function

Gets the current database ID.

Syntax

ul_u_long **ULGetDatabaseID(**
SQLCA * *sqlca*
)

Parameters

sqlca A pointer to the SQLCA.

Returns

- The value set by the last call to SetDatabaseID.
- UL_INVALID_DATABASE_ID if the ID was never set.

Remarks

The current database ID used for global autoincrement.

See also

- "UltraLite global_database_id option" [*UltraLite - Database Management and Reference*]

ULGetDatabaseProperty function

Obtains the value of a database property.

Syntax

void **ULGetDatabaseProperty (**
SQLCA * *sqlca***,**
ul_database_property_id *id***,**
char * *dst***,**
size_t *buffer-size***,**
ul_bool * *null-indicator*
);

Parameters

sqlca A pointer to the SQLCA.

id The identifier for the database property.

dst A character array to store the value of the property.

buffer-size The size of the character array *dst*.

null-indicator An indicator that the database parameter is null.

See also

- "UltraLite database properties" [*UltraLite - Database Management and Reference*]

ULGetErrorParameter function

Retrieve error parameter via ordinal.

Syntax

size_t **ULGetErrorParameter (**
SQLCA * *sqlca*,
ul_u_long *parm_num*,
ul_char * *buffer*,
size_t *size*
);

Parameters

sqlca A pointer to the sqlca.

parm_num The ordinal parameter number.

buffer Pointer to a buffer to contain the error parameter.

size The size in bytes of the buffer.

Returns

This function returns the number of characters copied to the supplied buffer.

See also

- "ULGetErrorParameterCount function" on page 275

ULGetErrorParameterCount function

Obtain count of the number of error parameters.

Syntax

ul_u_long **ULGetErrorParameterCount (**
SQLCA * *sqlca*
);

Parameters

sqlca A pointer to the SQLCA.

Returns

This function returns the number of error parameters. Unless the result is zero, values from 1 through this result can be used to call ULGetErrorParameter to retrieve the corresponding error parameter value.

See also

- "ULGetErrorParameter function" on page 274

ULGetLastDownloadTime function

Obtains the last time a specified publication was downloaded.

Syntax

ul_bool **ULGetLastDownloadTime (**
SQLCA * *sqlca***,**
ul_string *pub-name***,**
DECL_DATETIME * *value*
);

Parameters

sqlca A pointer to the SQLCA.

pub-name A string containing a publication name for which the last download time is retrieved.

value A pointer to the **DECL_DATETIME** structure to be populated. For example, value of January 1, 1990 indicates that the publication has yet to be synchronized.

Returns

- **true** The *value* is successfully populated by the last download time of the publication specified by *pub-name*.
- **false** The *pub-name* specified more than one publication or the specified publication is undefined. The contents of *value* are not meaningful.

Examples

The following call populates the dt structure with the date and time that publication UL_PUB_PUB1 was downloaded:

```
DECL_DATETIME dt;
ret = ULGetLastDownloadTime( &sqlca, UL_TEXT("UL_PUB_PUB1"), &dt );
```

See also

- "UL_SYNC_ALL macro" on page 125
- "UL_SYNC_ALL_PUBS macro" on page 126

GetSQLPassthroughScriptCount

Gets the number of SQL Passthrough scripts that can be run.

Syntax

ul_u_long**ULExecuteNextSQLPassthroughScript(**
SQLCA * *sqlca*
)

Parameters

sqlca A pointer to the SQLCA.

Returns

- Returns the number of SQL Passthrough scripts that can be run.

See also

- "UltraLite global_database_id option" [*UltraLite - Database Management and Reference*]

ULGetSynchResult function

A structure containing the results of the most recent synchronization, so that appropriate action can be taken in the application program.

Syntax

```
ul_bool ULGetSynchResult(
ul_synch_result  * synch-result
);
```

Parameters

synch-result A structure to hold the synchronization result. This structure is defined in *ulglobal.h* as follows:

```
typedef struct ul_synch_result {
   an_sql_code        sql_code;
   char               sql_error_string[];
   ul_stream_error    stream_error;
   ul_bool            upload_ok;
   ul_bool            ignored_rows;
   ul_auth_status     auth_status;
   ul_s_long          auth_value;
   SQLDATETIME        timestamp;
   ul_bool            partial_download_retained;
   ul_synch_status    status;
} ul_synch_result, * p_ul_synch_result;
```

The individual parameters include:

- **sql_code** The SQL code from the last synchronization. For a list of SQL codes, see "SQL Anywhere error messages sorted by SQLSTATE" [*Error Messages*].
- **sql_error_string** The error message text associated with the error reported in the field **sql_code**.
- **stream_error** A structure of type ul_stream_error. See "Stream Error synchronization parameter" [*UltraLite - Database Management and Reference*].
- **upload_ok** True if the upload was successful; false otherwise.
- **ignored_rows** True if uploaded rows were ignored; false otherwise.
- **auth_status** The synchronization authentication status. See "Authentication Status synchronization parameter" [*UltraLite - Database Management and Reference*].
- **auth_value** The value used by the MobiLink server to determine the **auth_status** result. See "Authentication Value synchronization parameter" [*UltraLite - Database Management and Reference*].
- **timestamp** The time and date of the last synchronization.
- **partial_download_retained** Flag to indicate whether a partial download has been retained.

- **status** The status information used by the observer function. See "Observer synchronization parameter" [*UltraLite - Database Management and Reference*].

Returns

- True if the operation succeeded.
- False if the operation failed.

Remarks

The application must allocate a ul_synch_result object before passing it to ULGetSynchResult. The function fills the ul_synch_result with the result of the last synchronization. These results are stored persistently in the database.

The function is of particular use when synchronizing applications on the Palm OS using HotSync, as the synchronization takes place outside the application itself. The **SQLCODE** value set in the connection reflects the result of the connecting operation itself. The synchronization status and results are written to the HotSync log only. To obtain extended synchronization result information, call ULGetSynchResult when connected to the database.

Examples

The following code checks for success of the previous synchronization.

```
ul_synch_result synch_result;
memset( &synch_result, 0, sizeof( ul_synch_result ) );
db_init( &sqlca );
EXEC SQL CONNECT "dba" IDENTIFIED BY "sql";
if( !ULGetSynchResult( &sqlca, &synch_result ) ) {
    prMsg( "ULGetSynchResult failed" );
}
```

ULGlobalAutoincUsage function

Obtains the percent of the default values used in all the columns having global autoincrement defaults.

Syntax

ul_u_short **ULGlobalAutoincUsage(**
SQLCA * *sqlca*
);

Returns

- A short in the range 0-100.

Remarks

If the database contains more than one column with this default, this value is calculated for all columns and the maximum is returned. For example, a return value of 99 indicates that very few default values remain for at least one of the columns.

See also

- "ULSetDatabaseID function" on page 289
- "UltraLite global_database_id option" [*UltraLite - Database Management and Reference*]

ULGrantConnectTo function

Grants access to an UltraLite database for a new or existing user ID with the given password.

Syntax

void **ULGrantConnectTo(**
SQLCA * *sqlca***,**
ul_char * *userid***,**
ul_char * *password*
);

Parameters

sqlca A pointer to the SQLCA.

userid A character array that holds the user ID. The maximum length is 16 characters.

password A character array that holds the password for the user ID. The maximum length is 16 characters.

Remarks

If you specify an existing user ID, this function then updates the password for the user.

See also

- "Authenticating users" on page 52
- "ULRevokeConnectFrom function" on page 286

ULInitSynchInfo function

Initializes the synchronization information structure.

Syntax

void **ULInitSynchInfo(**
ul_synch_info * *synch_info*
);

Parameters

synch_info A synchronization structure. For more information about the members of this structure, see "Synchronization parameters for UltraLite" [*UltraLite - Database Management and Reference*].

ULIsSynchronizeMessage function

Checks a message to see if it is a synchronization message from the MobiLink provider for ActiveSync, so that code to handle such a message can be called. When the processing of a synchronization message is complete, the ULSignalSyncIsComplete function should be called.

Syntax

ul_bool **ULIsSynchronizeMessage(**
ul_u_long *uMsg*
);

Remarks

You should include a call to this function in the WindowProc function of your application.

Applies to Windows Mobile for ActiveSync.

Example

The following code snippet illustrates how to use ULIsSynchronizeMessage to handle a synchronization message.

```
LRESULT CALLBACK WindowProc( HWND hwnd,
         UINT uMsg,
         WPARAM wParam,
         LPARAM lParam )
{
  if( ULIsSynchronizeMessage( uMsg ) ) {
    // execute synchronization code
    if( wParam == 1 ) DestroyWindow( hWnd );
    return 0;
  }

  switch( uMsg ) {

  // code to handle other windows messages

  default:
    return DefWindowProc( hwnd, uMsg, wParam, lParam );
  }
  return 0;
}
```

See also

- "Adding ActiveSync synchronization to your application" on page 92
- "ActiveSync on Windows Mobile" [*UltraLite - Database Management and Reference*]
- "ULSignalSyncIsComplete function" on page 293

ULResetLastDownloadTime function

Resets the last download time of a publication so that the application resynchronizes previously downloaded data.

Syntax

void **ULResetLastDownloadTime(**
SQLCA * *sqlca***,**
ul_string *pub-list*
);

Parameters

sqlca A pointer to the SQLCA.

pub-list A string containing a comma-separated list of publications to reset. An empty string implies all tables except tables marked as "no sync". A string containing just an asterisk ("*") implies all publications. Some tables may not be part of any publication and are not included if the pub-list string is "*".

Example

The following function call resets the last download time for all tables:

```
ULResetLastDownloadTime( &sqlca, UL_TEXT("") );
```

See also

- "ULGetLastDownloadTime function" on page 276
- "Timestamp-based downloads" [*MobiLink - Server Administration*]
- "UL_SYNC_ALL macro" on page 125
- "UL_SYNC_ALL_PUBS macro" on page 126

ULRetrieveEncryptionKey function

Retrieves the encryption key from memory.

Syntax

ul_bool **ULRetrieveEncryptionKey(**
ul_char * *key***,**
ul_u_short *len***,**
ul_u_long * *creator***,**
ul_u_long * *feature-num*
);

Parameters

key A pointer to a buffer in which to hold the retrieved encryption key.

len The length of the buffer that holds the encryption key with a terminating null character.

creator A pointer to the creator ID of the feature holding the encryption key. The default value is NULL.

feature-num A pointer to the feature number holding the encryption key. If feature-num is NULL, the application uses the UltraLite default value of 100.

Returns

- True if the operation is successful.
- False if the operation failed. This occurs if the feature was not found or if the supplied buffer length is insufficient to hold the key plus a terminating null character.

Remarks

On Palm OS, the encryption key is saved in dynamic memory as a Palm feature. Features are indexed by creator and a feature number.

Applies to Palm OS.

See also

- "ULClearEncryptionKey function" on page 267
- "ULSaveEncryptionKey function" on page 288

ULRevokeConnectFrom function

Revokes access from an UltraLite database for a user ID.

Syntax

void **ULRevokeConnectFrom(**
SQLCA * *sqlca***,**
ul_char * *userid*
);

Parameters

sqlca A pointer to the SQLCA.

userid A character array holding the user ID to be excluded from database access. The maximum length is 16 characters.

See also

- "Authenticating users" on page 52
- "ULGrantConnectTo function" on page 281

ULRollbackPartialDownload function

Rolls back the changes from a failed synchronization.

Syntax

void **ULRollbackPartialDownload(**
SQLCA * *sqlca*
);

Parameters

- **sqlca** A pointer to the SQL Communications Area. In the C++ API, use the Sqlca.GetCA method.

Remarks

When a communication error occurs during the download phase of synchronization, UltraLite can apply the downloaded changes, so that the application can resume the synchronization from the place it was interrupted. If the download changes are not needed (the user or application does not want to resume the download at this point), ULRollbackPartialDownload rolls back the failed download transaction.

See also

- "GetCA function" on page 147
- "Resuming failed downloads" [*MobiLink - Server Administration*]
- "Keep Partial Download synchronization parameter" [*UltraLite - Database Management and Reference*]
- "Partial Download Retained synchronization parameter" [*UltraLite - Database Management and Reference*]
- "Resume Partial Download synchronization parameter" [*UltraLite - Database Management and Reference*]

ULSaveEncryptionKey function

Saves the encryption key in Palm dynamic memory.

Syntax

ul_bool **ULSaveEncryptionKey(**
ul_char * *key***,**
ul_u_long * *creator***,**
ul_u_long * *feature-num*
);

Parameters

key A pointer to the encryption key.

creator A pointer to the creator ID of the feature that holds the encryption key. The default value is NULL.

feature-num A pointer to the feature number that holds the encryption key. If feature-num is NULL, the application uses the UltraLite default value of 100.

Returns

- True if the operation is successful.
- False if the operation failed. This occurs if the feature was not found or if the supplied buffer length is insufficient to hold the key plus a terminating null character.

Remarks

On the Palm OS the encryption key is saved in dynamic memory as a Palm feature. Features are indexed by creator and a feature number. They are not backed up and are cleared on any reset of the device.

Applies to Palm OS applications.

See also

- "ULClearEncryptionKey function" on page 267
- "ULRetrieveEncryptionKey function" on page 285

ULSetDatabaseID function

Sets the database identification number.

Syntax

void **ULSetDatabaseID(**
SQLCA * *sqlca***,**
ul_u_long *id*
);

Parameters

sqlca A pointer to the SQLCA.

id A positive integer that uniquely identifies a particular database in a replication or synchronization setup.

See also

- "UltraLite global_database_id option" [*UltraLite - Database Management and Reference*]
- "ULGlobalAutoincUsage function" on page 280

ULSetDatabaseOptionString function

Sets a database option from a string value.

Syntax

void **ULSetDatabaseOptionString (**
SQLCA * *sqlca***,**
ul_database_option_id *id***,**
ul_char * *value*
);

Parameters

sqlca A pointer to the SQLCA.

id The identifier for the database option to be set.

value The value of the database option.

ULSetDatabaseOptionULong function

Sets a numeric database option.

Syntax

void **ULSetDatabaseOptionULong(**
SQLCA * *sqlca***,**
ul_database_option_id *id***,**
ul_u_long * *value*
);

Parameters

sqlca A pointer to the SQLCA.

id The identifier for the database option to be set.

value The value of the database option.

ULSetSynchInfo function

Stores the synchronization parameters for use with HotSync.

Syntax

```
ul_bool ULSetSynchInfo(
SQLCA * sqlca,
PROFILE sync-profile-name
ul_synch_info *  synch_info
);
```

Parameters

sqlca A pointer to the SQLCA.

sync-profile-name The name of the synchronization profile to which to store the synchronization information. This can then be referenced by name as part of a HotSync. See “UltraLite HotSync Conduit Installation utility for Palm OS (ulcond11)” [*UltraLite - Database Management and Reference*].

synch_info A synchronization structure. For more information about the members of this structure, see “ul_synch_info_a struct” on page 133.

Remarks

Typically, you call ULSetSynchInfo just before closing the application with db_fini.

Applies to Palm OS applications with HotSync.

See also

- “db_fini function” on page 261

ULSignalSyncIsComplete function

This function is called to indicate that processing a synchronization message is complete.

Synopsis

void **ULSignalSyncIsComplete();**

Remarks

Applications which are registered with the ActiveSync provider need to call this method in their WNDPROC when processing a synchronization message is complete.

See also

- “ULIsSynchronizeMessage function” on page 283

ULSynchronize function

Initiates synchronization in an UltraLite application.

Syntax

void **ULSynchronize(**
SQLCA * *sqlca***,**
ul_synch_info * *synch_info*
);

Parameters

sqlca A pointer to the SQLCA.

synch_info A synchronization structure. Synchronization specifics are controlled through a set of synchronization parameters. For more information about these parameters, see "ul_synch_info_a struct" on page 133.

Remarks

For TCP/IP or HTTP synchronization, the ULSynchronize function initiates synchronization. Errors during synchronization that are not handled by the handle_error script are reported as SQL errors. Application programs should test the SQLCODE return value of this function.

UltraLite ODBC API reference

Contents

This section describes the components of the ODBC interface supported by UltraLite.

This is not a comprehensive ODBC reference; it is intended as a quick reference to complement the main reference for ODBC, which is the Microsoft ODBC Programmer's Reference.

SQLAllocHandle function

Allocates a handle, for UltraLite ODBC.

Syntax

UL_FN_SPEC SQLRETURN UL_FN_MOD **SQLAllocHandle(**
SQLSMALLINT *HandleType*,
SQLHANDLE *InputHandle*,
SQLHANDLE * *OutputHandle* **)**;

Parameters

- **HandleType** The type of handle to be allocated. UltraLite supports the following handle types:
  - SQL_HANDLE_ENV (environment handle)
  - SQL_HANDLE_DBC (connection handle)
  - SQL_HANDLE_STMT (statement handle)
- **InputHandle** The handle in whose context the new handle is to be allocated. For a connection handle, this is the environment handle; for a statement handle, this is the connection handle.
- **OutputHandle** A pointer to a buffer in which to return the new handle.

Remarks

ODBC uses handles to provide the context for database operations. An environment handle provides the context for communication with a data source, like the SQL Communications Area in other interfaces. A connection handle provides a context for all database operations. A statement handle manages result sets and data modification. A descriptor handle manages the handling of result set data types.

SQLBindCol function

Binds a result set column to an application data buffer, for UltraLite ODBC.

Syntax

UL_FN_SPEC SQLRETURN UL_FN_MOD **SQLBindCol (**
SQLHSTMT *StatementHandle*,
SQLUSMALLINT *ColumnNumber*,
SQLSMALLINT *TargetType*,
SQLPOINTER *TargetValue*,
SQLLEN *BufferLength*,
SQLLEN * *StrLen_or_Ind* **)**;

Parameters

- **StatementHandle** A handle for the statement that is to return a result set.
- **ColumnNumber** The number of the column in the result set to bind to an application data buffer.
- **TargetType** The identifier of the data type of the *TargetValue* pointer.
- **TargetValue** A pointer to the data buffer to bind to the column.
- **BufferLength** The length of the *TargetValue* buffer in bytes.
- **StrLen_or_Ind** A pointer to the length or indicator buffer to bind to the column. For strings, the length buffer holds the length of the actual string that was returned, which may be less than the length allowed by the column.

Remarks

To exchange information between your application and the database, ODBC binds buffers in the application to database objects such as columns. SQLBindCol is used when executing a query to identify a buffer in your application as a place that UltraLite puts the value of a specified column.

SQLBindParameter function

Binds a buffer parameter to a parameter marker in a SQL statement, for UltraLite ODBC.

Syntax

UL_FN_SPEC SQLRETURN UL_FN_MOD **SQLBindParameter (**
SQLHSTMT *StatementHandle*,
SQLUSMALLINT *ParameterNumber*,
SQLSMALLINT *ParamType*,
SQLSMALLINT *CType*,
SQLSMALLINT *SqlType*,
SQLULEN *ColDef*,
SQLSMALLINT *Scale*,
SQLPOINTER *rgbValue*,
SQLLEN *cbValueMax*,
SQLLEN * *StrLen_or_Ind* **)**;

Parameters

- **StatementHandle** A statement handle.
- **ParameterNumber** The number of the parameter marker in the statement, in sequential order counting from 1.
- **ParamType** The parameter type. One of the following:
  - SQL_PARAM_INPUT
  - SQL_PARAM_INPUT_OUTPUT
  - SQL_PARAM_OUTPUT
- **CType** The C data type of the parameter.
- **SQLType** The SQL data type of the parameter.
- **ColDef** The size of the column or expression of the parameter marker.
- **Scale** The number of decimal digits for the column or expression of the parameter marker.
- **rgbValue** A pointer to a buffer for the parameter's data.
- **cbValueMax** The length of the *rgbValue*rgbValue buffer.
- **StrLen_or_Ind** A pointer to a buffer for the parameter's length.

Remarks

To exchange information between your application and the database, ODBC binds buffers in the application to database objects such as columns. SQLBindParameter is used when executing a statement, to identify a buffer in your application as a place that UltraLite gets or sets the value of a specified parameter in a query.

SQLConnect function

Connects to a database, for UltraLite ODBC.

Syntax

UL_FN_SPEC SQLRETURN UL_FN_MOD **SQLConnect (**
SQLHDBC *ConnectionHandle*,
SQLTCHAR * *ServerName*,
SQLSMALLINT *NameLength1*,
SQLTCHAR * *UserName*,
SQLSMALLINT *NameLength2*,
SQLTCHAR * *Authentication*,
SQLSMALLINT *NameLength3* **)**;

Parameters

- **ConnectionHandle** The connection handle.
- **ServerName** A connection string that defines the database to which your application connects. UltraLite ODBC does not use ODBC data sources. Instead, supply a connection string containing the database connection parameters, together with optional other parameters.

  The following is an example of a ServerName parameter:

  ```
  (SQLTCHAR*)UL_TEXT(
      "dbf=customer.udb"
  )
  ```

  For a complete list of connection parameters, see "UltraLite connection parameters" [*UltraLite - Database Management and Reference*].
- **NameLength1** The length of **ServerName*.
- **UserName** The user ID to use when connecting. The user ID can alternatively be specified in the connection string supplied to the ServerName parameter.
- **NameLength2** The length of **UserName*.
- **Authentication** The password to use when connecting. The password can alternatively be specified in the connection string supplied to the ServerName parameter.
- **NameLength3** The length of **Authentication*.

Remarks

Connects to a database. For more information about UltraLite connection parameters, see "UltraLite connection parameters" [*UltraLite - Database Management and Reference*].

SQLDescribeCol function

Returns the result descriptor for a column in the result set, for UltraLite ODBC.

The result descriptor includes the column name, column size, data type, number of decimal digits, and nullability.

Syntax

```
UL_FN_SPEC SQLRETURN UL_FN_MOD SQLDescribeCol (
   SQLHSTMT StatementHandle,
   SQLUSMALLINT ColumnNumber,
   SQLTCHAR * ColumnName,
   SQLSMALLINT BufferLength,
   SQLSMALLINT * NameLength,
   SQLSMALLINT * DataType,
   SQLULEN * ColumnSize,
   SQLSMALLINT * DecimalDigits,
   SQLSMALLINT * Nullable );
```

Parameters

- **StatementHandle** A statement handle.
- **ColumnNumber** The 1-based column number of result data.
- **ColumnName** A pointer to a buffer in which to return the column name.
- **BufferLength** The length of **ColumnName*, in characters.
- **NameLength** A pointer to a buffer in which to return the total number of bytes (excluding the null-termination byte) available to return in **ColumnName*.
- **DataType** A pointer to a buffer in which to return the SQL data type of the column.
- **ColumnSize** A pointer to a buffer in which to return the size of the column on the data source.
- **DecimalDigits** A pointer to a buffer in which to return the number of decimal digits of the column on the data source.
- **Nullable** A pointer to a buffer in which to return a value that indicates whether the column allows NULL values.

SQLDisconnect function

Disconnects the application from a database, for UltraLite ODBC.

Syntax

UL_FN_SPEC SQLRETURN UL_FN_MOD **SQLDisconnect (**
SQLHDBC *ConnectionHandle* **)**;

Parameters

- **ConnectionHandle** The handle for the connection to be closed.

Remarks

Once SQLDisconnect is called, no further operations can be carried out against the database without opening a new connection.

See also

- "SQLConnect function" on page 299

SQLEndTran function

Commits or rolls back a transaction, for UltraLite ODBC.

Syntax

UL_FN_SPEC SQLRETURN UL_FN_MOD **SQLEndTran (**
SQLSMALLINT *HandleType*,
SQLHANDLE *Handle*,
SQLSMALLINT *CompletionType* **)**;

Parameters

- **HandleType** The type of handle to be allocated. UltraLite supports the following handle types:
  - SQL_HANDLE_ENV
  - SQL_HANDLE_DBC
  - SQL_HANDLE_STMT
- **Handle** The connection handle indicating the scope of the transaction.
- **CompletionType** One of the following two values:
  - SQL_COMMIT
  - SQL_ROLLBACK

SQLExecDirect function

Executes a SQL statement, for UltraLite ODBC.

Syntax

UL_FN_SPEC SQLRETURN UL_FN_MOD **SQLExecDirect (**
SQLHSTMT *StatementHandle*,
SQLTCHAR * *StatementText*,
SQLINTEGER *TextLength* **)**;

Parameters

- **StatementHandle** A statement handle.
- **StatementText** The text of the SQL statement.
- **TextLength** The length of **StatementText*.

Remarks

Unlike SQLExecute, the statement does not need to be prepared before being executed using SQLExecDirect.

SQLExecDirect has slower performance than SQLExecute for statements executed repeatedly.

See also

- "SQLExecute function" on page 304

SQLExecute function

Executes a prepared SQL statement, for UltraLite ODBC.

Syntax

UL_FN_SPEC SQLRETURN UL_FN_MOD **SQLExecute (**
SQLHSTMT *StatementHandle* **)**;

Parameters

- **StatementHandle** The handle for the statement to be executed.

Remarks

The statement must be prepared using SQLPrepare before it can be executed. If the statement has parameter markers, they must be bound to variables using SQLBindParameter before execution.

You can use SQLExecDirect to execute a statement without preparing it first. SQLExecDirect has slower performance than SQLExecute for statements executed repeatedly.

See also

- "SQLBindParameter function" on page 298
- "SQLPrepare function" on page 313
- "SQLExecDirect function" on page 303

SQLFetch function

Fetches the next row from a result set and returns data for all bound columns, for UltraLite ODBC.

Syntax

UL_FN_SPEC SQLRETURN UL_FN_MOD **SQLFetch (**
SQLHSTMT *StatementHandle* **)**;

Parameters

- **StatementHandle** A statement handle.

Remarks

Before fetching rows, you must have bound the columns in the result set to buffers using SQLBindCol. To fetch a row other than the next row in the result set, use SQLFetchScroll.

See also

- "SQLFetchScroll function" on page 306
- "SQLBindCol function" on page 297

SQLFetchScroll function

Fetches the specified row from the result set and returns data for all bound columns, for UltraLite ODBC.

Syntax

UL_FN_SPEC SQLRETURN UL_FN_MOD **SQLFetchScroll (**
SQLHSTMT *StatementHandle*,
SQLSMALLINT *FetchOrientation*,
SQLLEN *FetchOffset* **)**;

Parameters

- **StatementHandle** A statement handle.
- **FetchOrientation** The type of fetch.
- **FetchOffset** The number of the row to fetch. The interpretation depends on the value of *FetchOrientation*.

Remarks

Before fetching rows, you must have bound the columns in the result set to buffers using SQLBindCol. SQLFetchScroll is for use in those cases where the more straightforward SQLFetch is not appropriate.

See also

- "SQLFetch function" on page 305
- "SQLBindCol function" on page 297

SQLFreeHandle function

Frees resources for a handle allocated, for UltraLite ODBC.

Syntax

UL_FN_SPEC SQLRETURN UL_FN_MOD **SQLFreeHandle (**
SQLSMALLINT *HandleType*,
SQLHANDLE *Handle* **)**;

Parameters

- **HandleType**   The type of handle to be allocated. UltraLite supports the following handle types:

  - SQL_HANDLE_ENV
  - SQL_HANDLE_DBC
  - SQL_HANDLE_STMT

- **Handle**   The handle to be freed.

Remarks

SQLFreeHandle should be called for each handle allocated using SQLAllocHandle, when the handle is no longer needed.

See also

- "SQLAllocHandle function" on page 296

SQLGetCursorName function

Returns the name associated with a cursor for a specified statement, for UltraLite ODBC.

Syntax

UL_FN_SPEC SQLRETURN UL_FN_MOD **SQLGetCursorName (**
  SQLHSTMT *StatementHandle*,
  SQLTCHAR * *CursorName*,
  SQLSMALLINT *BufferLength*,
  SQLSMALLINT * *NameLength* **)**;

Parameters

- **StatementHandle** A statement handle.
- **CursorName** A pointer to a buffer in which to return the name of the cursor associated with *StatementHandle*.
- **BufferLength** The length of **CursorName*.
- **NameLength** A pointer to memory in which to return the total number of bytes (excluding the null-termination character) available to return in **CursorName*.

See also

- "SQLSetCursorName function" on page 316

SQLGetData function

Retrieves data for a single column in the result set, for UltraLite ODBC.

Syntax

UL_FN_SPEC SQLRETURN UL_FN_MOD **SQLGetData (**
SQLHSTMT *StatementHandle*,
SQLUSMALLINT *ColumnNumber*,
SQLSMALLINT *TargetType*,
SQLPOINTER *TargetValue*,
SQLLEN *BufferLength*,
SQLLEN * *StrLen_or_Ind* **)**;

Parameters

- **StatementHandle** A statement handle.
- **ColumnNumber** The number of the column in the result set to bind.
- **TargetType** The output handle.
- **TargetValue** A pointer to the data buffer to bind to the column.
- **BufferLength** The length of the *TargetValue* buffer in bytes.
- **StrLen_or_Ind** A pointer to the length or indicator buffer to bind to the column.

Remarks

SQLGetData is typically used to retrieve variable-length data in parts.

SQLGetDiagRec function

Returns the current values of multiple fields of a diagnostic status record, for UltraLite ODBC.

Syntax

UL_FN_SPEC SQLRETURN UL_FN_MOD **SQLGetDiagRec (**
SQLSMALLINT *HandleType*,
SQLHANDLE *Handle*,
SQLSMALLINT *RecNumber*,
SQLTCHAR * *Sqlstate*,
SQLINTEGER * *NativeError*,
SQLTCHAR * *MessageText*,
SQLSMALLINT *BufferLength*,
SQLSMALLINT * *TextLength* **)**;

Parameters

- **HandleType** The type of handle to be allocated. UltraLite supports the following handle types:
  - SQL_HANDLE_ENV
  - SQL_HANDLE_DBC
  - SQL_HANDLE_STMT
- **Handle** The input handle
- **RecNumber** The output handle.
- **Sqlstate** The ANSI/ISO SQLSTATE value of the error. For a listing, see "SQL Anywhere error messages sorted by SQLSTATE" [*Error Messages*].
- **NativeError** The SQLCODE value of the error. For a listing, see "SQL Anywhere error messages sorted by SQLCODE" [*Error Messages*].
- **MessageText** The text of the error or status message.
- **BufferLength** The length of the **MessageText* buffer in bytes.
- **TextLength** A pointer to a buffer in which to return the total number of bytes (excluding the null-termination byte) available to return in **MessageText*.

SQLGetInfo function

Returns general information about the current ODBC driver and data source, for UltraLite ODBC.

Syntax

UL_FN_SPEC SQLRETURN UL_FN_MOD **SQLGetInfo (**
SQLHDBC *ConnectionHandle*,
SQLUSMALLINT *InfoType*,
SQLPOINTER * *InfoValue*,
SQLSMALLINT *BufferLength*,
SQLSMALLINT ODBCFAR * *StringLength* **)**;

Parameters

- **ConnectionHandle** A connection handle.
- **InfoType** The type of information returned. The only type supported is SQL_DBMS_VER. The information returned is a character string identifying the current release of the software.
- **InfoValue** A pointer to a buffer in which to return the information.
- **BufferLength** The length of the **InfoValue* buffer in bytes.
- **StringLength** A pointer to a buffer in which to return the total number of bytes (excluding the null-termination character for character data) available to return in **InfoValue*.

SQLNumResultCols function

Returns the number of columns in a result set, for UltraLite ODBC.

Syntax

UL_FN_SPEC SQLRETURN UL_FN_MOD **SQLNumResultCols (**
SQLHSTMT *StatementHandle*,
SQLSMALLINT * *ColumnCount* **)**;

Parameters

- **StatementHandle** A statement handle.
- **ColumnCount** A pointer to a buffer in which to return the total number of columns in the result set.

SQLPrepare function

Prepares a SQL statement for execution, for UltraLite ODBC.

Syntax

UL_FN_SPEC SQLRETURN UL_FN_MOD **SQLPrepare (**
SQLHSTMT *StatementHandle*,
SQLTCHAR * *StatementText*,
SQLINTEGER *TextLength* **)**;

Parameters

- **StatementHandle** A statement handle.
- **StatementText** A pointer to a buffer that holds the SQL statement text.
- **TextLength** The length of **StatementText*.

See also

- "SQLExecute function" on page 304

SQLRowCount function

Returns the number of rows affected by an INSERT, UPDATE, or DELETE operation, for UltraLite ODBC.

Syntax

UL_FN_SPEC SQLRETURN UL_FN_MOD **SQLRowCount (**
SQLHSTMT *StatementHandle*,
SQLLEN * *RowCount* **)**;

Parameters

- **StatementHandle** A statement handle.
- **RowCount** A pointer to a buffer in which the number of rows is returned.

SQLSetConnectionName function

Sets a connection name for the suspend and restore operation, for UltraLite ODBC. This function is specific to UltraLite and is not part of the ODBC standard.

Syntax

UL_FN_SPEC SQLRETURN UL_FN_MOD **SQLSetConnectionName (**
SQLHSTMT *StatementHandle*,
SQLTCHAR * *ConnectionName*,
SQLSMALLINT *NameLength* **)**;

Parameters

- **StatementHandle** A statement handle.
- **ConnectionName** A pointer to a buffer holding the connection name.
- **NameLength** The length of **ConnectionName*

Remarks

SQLSetConnectionName is used to provide a connection name for use in the suspend and restore operation, together with SQLSetSuspend. Set the connection name before opening a connection to restore application state.

See also

- "Maintaining state in UltraLite Palm applications (deprecated)" on page 74
- "SQLSetSuspend function (deprecated)" on page 317

SQLSetCursorName function

Sets the name of a cursor associated with a SQL statement, for UltraLite ODBC.

Syntax

UL_FN_SPEC SQLRETURN UL_FN_MOD **SQLSetCursorName (**
SQLHSTMT *StatementHandle*,
SQLTCHAR * *CursorName*,
SQLSMALLINT *NameLength* **)**;

Parameters

- **StatementHandle** A statement handle.
- **CursorName** A pointer to a buffer holding the cursor name.
- **NameLength** The length of **CursorName*.

See also

- "SQLGetCursorName function" on page 308

SQLSetSuspend function (deprecated)

Indicates whether the state of open cursors should be saved on closing the application, for UltraLite ODBC. This function is specific to UltraLite and is not part of the ODBC standard.

Syntax

UL_FN_SPEC SQLRETURN UL_FN_MOD **SQLSetSuspend (**
SQLSMALLINT *HandleType*,
SQLHSTMT *StatementHandle*,
SQLSMALLINT *TrueFalse* **)**;

Parameters

- **HandleType** The type of handle to be allocated. UltraLite supports the following handle types:
  - SQL_HANDLE_ENV
  - SQL_HANDLE_DBC
  - SQL_HANDLE_STMT
- **StatementHandle** A statement handle.
- **TrueFalse** The output handle.

See also

- "Maintaining state in UltraLite Palm applications (deprecated)" on page 74

SQLSynchronize function

Synchronizes data in the database using MobiLink synchronization, for UltraLite ODBC. This function is specific to UltraLite and is not part of the ODBC standard.

Syntax

UL_FN_SPEC SQLRETURN UL_FN_MOD **SQLSynchronize (**
SQLHDBC *ConnectionHandle*,
ul_synch_info * *SynchInfo* **)**;

Parameters

- **ConnectionHandle** A handle.
- **SynchInfo** The structure holding synchronization information. See "UltraLite synchronization parameters and network protocol options" [*UltraLite - Database Management and Reference*].

Remarks

SQLSynchronize is an extension to ODBC. It initiates a MobiLink synchronization operation.

See also

- "UltraLite synchronization parameters and network protocol options" [*UltraLite - Database Management and Reference*]
- MobiLink - Server Administration

Tutorial: Build an application using the C++ API

Contents

This tutorial guides you through the process of building an UltraLite C++ application. The application is built for Windows desktop operating systems, and runs at a command prompt.

This tutorial is based on development using Microsoft Visual C++, although you can also use any C++ development environment.

The tutorial takes about 30 minutes if you copy and paste the code. The final section of this tutorial contains the full source code of the program described in this tutorial.

Competencies and experience

This tutorial assumes:

- You are familiar with the C++ programming language
- You have a C++ compiler installed on your computer
- You know how to create an UltraLite database with the **Create Database Wizard**.

  For more information, see "Create a database with the Create Database Wizard" [*UltraLite - Database Management and Reference*].

The goal for the tutorial is to gain competence with the process of developing an UltraLite C++ application.

Lesson 1: Create database and connect to database

In the first procedure, you create a local UltraLite database. You then write, compile, and run a C++ application that accesses the database you created.

To create an UltraLite database

1. Add *c:\vendor\visualstudio8\VC\atlmfc\src\atl* to your INCLUDE environment variable.
2. Create a directory to contain the files you will create in this tutorial.

   The remainder of this tutorial assumes that this directory is *C:\tutorial\cpp\*. If you create a directory with a different name, use that directory instead of *C:\tutorial\cpp\*.
3. Using UltraLite in Sybase Central, create a database named *ULCustomer.udb* in your new directory with the following characteristics.

   For more information about using UltraLite in Sybase Central, see "Create a database with the Create Database Wizard" [*UltraLite - Database Management and Reference*].
4. Add a table named **ULCustomer** to the database. Use the following specifications for the ULCustomer table:

Column name	Data type (size)	Column allows NULL values?	Default value	Primary Key
cust_id	integer	No	autoincrement	ascending
cust_name	varchar(30)	No	None	

5. Disconnect from the database in Sybase Central, otherwise your executable will not be able to connect.

Connecting to the UltraLite database

1. In Microsoft Visual C++, choose **File** » **New**.
2. On the **Files** tab, choose **C++ Source File**.
3. Save the file as *customer.cpp* in your tutorial directory.
4. Include the UltraLite libraries and use the UltraLite namespace.

   Copy the code below into *customer.cpp*:

```
#include <tchar.h>
#include <stdio.h>
#include "uliface.h"
using namespace UltraLite;
#define MAX_NAME_LEN 100
ULSqlca Tutca;
```

   This code fragment defines an UltraLite SQL communications area (ULSqlca) named Tutca.

   For more information about using the UltraLite namespace to make class declarations simpler, see "Using the UltraLite namespace" on page 10.

5. Define connection parameters to connect to the database.

   In this code fragment, the connection parameters are hard coded. In a real application, the locations might be specified at runtime.

   Copy the code below into *customer.cpp*.

   ```
   static ul_char const * ConnectionParms =
     UL_TEXT( "UID=DBA;PWD=sql" )
     UL_TEXT( ";DBF=C:\\tutorial\\cpp\\ULCustomer.udb" );
   ```

   For more information about connection parameters, see "UltraLite_Connection_iface class" on page 156.

Special Character handling
A backslash character that appears in the file name location string must be escaped by a preceding backslash character.

6. Define a method for handling database errors in the application.

   UltraLite provides a callback mechanism to notify the application of errors. In a development environment this function can be useful as a mechanism to handle errors that were not anticipated. A production application typically includes code to handle all common error situations. An application can check for errors after every call to an UltraLite function or can choose to use an error callback function.

   This is a sample callback function.

   ```
   ul_error_action UL_GENNED_FN_MOD MyErrorCallBack(
    SQLCA *   Tutca,
    ul_void * user_data,
    ul_char * message_param )
   {
       ul_error_action rc;

       (void) user_data;

       switch( Tutca->sqlcode ){
           // The following error is used for flow control - don't report it
   here
           case SQLE_NOTFOUND:
               rc = UL_ERROR_ACTION_CONTINUE;
           break;

           default:
               if (Tutca->sqlcode >= 0) { // warning or success
                   rc = UL_ERROR_ACTION_DEFAULT;
               } else { // negative is real error
                   _tprintf( _TEXT( "Error %ld: %s\n" ), Tutca->sqlcode,
   message_param );
                   rc = UL_ERROR_ACTION_CANCEL;
               }
           break;
       }
       return rc;
    }
   ```

   In UltraLite, the error SQLE_NOTFOUND is often used to control application flow. That error is signaled to mark the end of a loop over a result set. The generic error handler coded above does not output a message for this error condition.

For more information about error handling, see "Handling errors" on page 25.

7. Define a method to open a connection to a database.

   If the database file does not exist, an error message is displayed, otherwise a connection is established.

```
Connection * open_conn( DatabaseManager * dm ) {
  Connection * conn = dm->OpenConnection( Tutca, ConnectionParms );

  if( conn == UL_NULL ) {
      _tprintf( _TEXT("Unable to open existing database.\n") );
   }
  return conn;
}
```

8. Implement the main method to carry out the following tasks:

   - Instantiates a DatabaseManager object. All UltraLite objects are created from the DatabaseManager object.
   - Registers the error handling function.
   - Opens a connection to the database.
   - Closes the connection and shuts down the database manager.

```
int main() {
  ul_char buffer[ MAX_NAME_LEN ];

  Connection * conn;

  Tutca.Initialize();

  ULRegisterErrorCallback(
    Tutca.GetCA(), MyErrorCallBack,
    UL_NULL, buffer, sizeof (buffer));

  DatabaseManager * dm = ULInitDatabaseManager( Tutca );

  conn = open_conn( dm );

  if( conn == UL_NULL ){
   dm->Shutdown( Tutca );
      Tutca.Finalize();
      return 1;
  }
  // main processing code to be inserted here
  dm->Shutdown( Tutca );
  Tutca.Finalize();
  return 0;
}
```

9. Compile and link the source file.

   The method you use to compile the source file depends on your compiler. The following instructions are for the Microsoft Visual C++ command line compiler using a makefile:

   - Open a command prompt and change to your tutorial directory.
   - Create a makefile named *makefile*.
   - In the makefile, add directories to your include path.

```
IncludeFolders=/I"$(SQLANY11)\SDK\Include"
```

- In the makefile, add directories to your libraries path.

  ```
  LibraryFolders=/LIBPATH:"$(SQLANY11)\UltraLite\win32\386\Lib\vs8"
  ```

- In the makefile, add libraries to your linker options.

  ```
  Libraries=ulimp.lib
  ```

  The UltraLite runtime library is named *ulimp.lib*.

- In the makefile, set compiler options. You must enter these options on a single line.

  ```
  CompileOptions=/c /nologo /W3 /Od /Zi /DWIN32 /DUL_USE_DLL
  ```

- In the makefile, add an instruction for linking the application.

  ```
  customer.exe: customer.obj
     link /NOLOGO /DEBUG customer.obj $(LibraryFolders) $(Libraries)
  ```

- In the makefile, add an instruction for compiling the application.

  ```
  customer.obj: customer.cpp
     cl $(CompileOptions) $(IncludeFolders) customer.cpp
  ```

- Run the makefile.

  ```
  nmake
  ```

  This creates an executable named *customer.exe*.

10. Run the application.

    At the command prompt, enter **customer**.

Lesson 2: Insert data into the database

The following procedure demonstrates how to add data to a database.

Adding rows to your database

1. Add the procedure below to *customer.cpp*, immediately before the main method:

```
bool do_insert( Connection * conn ) {
  Table * table = conn->OpenTable( UL_TEXT("ULCustomer") );
  if( table == UL_NULL ) {
      _tprintf( _TEXT("Table not found: ULCustomer\n") );
      return  false;
  }
  if( table->GetRowCount() == 0 ) {
      _tprintf( _TEXT("Inserting one row.\n") );
      table->InsertBegin();
      table->Set( UL_TEXT("cust_name"), UL_TEXT("New Customer") );
      table->Insert();
      conn->Commit();
  } else {
      _tprintf( _TEXT("The table has %lu rows\n"),
      table->GetRowCount() );
  }
  table->Release();
  return true;
}
```

   This procedure carries out the following tasks.

   - Opens the table using the connection->OpenTable() method. You must open a Table object to carry out operations on the table.
   - If the table is empty, adds a row to the table. To insert a row, the code changes to insert mode using the InsertBegin method, sets values for each required column, and executes an insert to add the row to the database.

     The commit method is only required when autocommit is turned off. By default, autocommit is enabled, but it may be disabled for better performance, or for multi-operation transactions.
   - If the table is not empty, reports the number of rows in the table.
   - Closes the Table object to free resources associated with it.
   - Returns a boolean indicating whether the operation was successful.

2. Call the do_insert method you have created.

   Add the following line to the `main()` method, immediately after the call to open_conn.

```
do_insert(conn);
```

3. Compile your application by running nmake.
4. Run your application by typing **customer** at the command prompt.

Lesson 3: Select and list rows from the table

The following procedure retrieves rows from the table and prints them on the command line.

Listing rows in the table

1. Add the method below to *customer.cpp*. This method carries out the following tasks:
   - Opens the Table object.
   - Retrieves the column identifiers.
   - Sets the current position before the first row of the table.

     Any operations on a table are carried out at the current position. The position may be before the first row, on one of the rows of the table, or after the last row. By default, as in this case, the rows are ordered by their primary key value (cust_id). To order rows in a different way, you can add an index to an UltraLite database and open a table using that index.
   - For each row, the cust_id and cust_name values are written out. The loop carries on until the Next method returns false, which occurs after the final row.
   - Closes the Table object.

```
bool do_select( Connection * conn ) {
  Table * table = conn->OpenTable( _TEXT("ULCustomer") );
  if( table == UL_NULL ) {
    return false;
  }
  TableSchema * schema = table->GetSchema();
  if( schema == UL_NULL ) {
    table->Release();
    return false;
  }
  ul_column_num id_cid =
     schema->GetColumnID( UL_TEXT("cust_id") );
  ul_column_num cname_cid =
     schema->GetColumnID( UL_TEXT("cust_name") );

  schema->Release();

  _tprintf( _TEXT("\n\nTable 'ULCustomer' row contents:\n") );

  while( table->Next() ) {
    ul_char cname[ MAX_NAME_LEN ];

    table->Get( cname_cid ).GetString(
                cname, MAX_NAME_LEN );

    _tprintf( _TEXT("id=%d, name=%s \n"), (int)table->Get(id_cid), cname);
  }
  table->Release();
  return true;
}
```

2. Add the following line to the main method, immediately after the call to the insert method:

```
do_select(conn);
```

3. Compile your application by running *nmake*.

4. Run your application by typing *customer* at the command prompt.

Lesson 4: Add synchronization to your application

This lesson synchronizes your application with a consolidated database running on your computer.

The following procedures add synchronization code to your application, start the MobiLink server, and run your application to synchronize.

The UltraLite database you created in the previous lessons synchronizes with the UltraLite 11 Sample database. The UltraLite 11 Sample database has a ULCustomer table whose columns include those in the customer table of your local UltraLite database.

This lesson assumes that you are familiar with MobiLink synchronization.

Add synchronization to your application

1. Add the method below to *customer.cpp*. This method carries out the following tasks:

   - Sets the synchronization stream to TCP/IP by invoking ULEnableTcpipSynchronization. Synchronization can also be carried out over HTTP, HotSync, or HTTPS. See “UltraLite clients” [*UltraLite - Database Management and Reference*].
   - Sets the script version. MobiLink synchronization is controlled by scripts stored in the consolidated database. The script version identifies which set of scripts to use.
   - Sets the MobiLink user name. This value is used for authentication at the MobiLink server. It is distinct from the UltraLite database user ID, although in some applications you may want to give them the same value.
   - Sets the download_only parameter to true. By default, MobiLink synchronization is two-way. This application uses download-only synchronization so that the rows in your table do not get uploaded to the sample database.

```
bool do_sync( Connection * conn ) {
  ul_synch_info info;
  ul_stream_error * se = &info.stream_error;

  ULEnableTcpipSynchronization( Tutca.GetCA() );
  conn->InitSynchInfo( &info );
  info.stream = ULSocketStream();
  info.version = UL_TEXT( "custdb 11.0" );
  info.user_name = UL_TEXT( "50" );
  info.download_only = true;
  if( !conn->Synchronize( &info ) ) {
      _tprintf( _TEXT("Synchronization error \n" ));
   _tprintf( _TEXT("  stream_error_code is '%lu'\n"), se-
>stream_error_code );
   _tprintf( _TEXT("  system_error_code is '%ld'\n"), se-
>system_error_code );
   _tprintf( _TEXT("  error_string is '") );
   _tprintf( _TEXT("%s"), se->error_string );
   _tprintf( _TEXT("'\n") );
   return false;
  }
  return true;
}
```

2. Add the following line to the main method, immediately after the call to the insert method and before the call to the select method:

```
do_sync(conn);
```

3. Compile your application by running *nmake*.

Synchronize data

1. Start the MobiLink server.

   From a command prompt, run the following command:

   ```
   mlsrv11 -c "dsn=SQL Anywhere 11 CustDB" -v+ -zu+
   ```

   The -zu+ option provides automatic addition of users. The -v+ option turns on verbose logging for all messages.

   For more information about this option, see "MobiLink server options" [*MobiLink - Server Administration*].

2. Run your application by typing *customer* at the command prompt.

   The MobiLink server window displays status messages indicating the synchronization progress. If synchronization is successful, the final message displays `Synchronization complete`.

Code listing for tutorial

Following is the complete code for the tutorial program described in the preceding sections.

```
#include <tchar.h>
#include <stdio.h>
#include "uliface.h"
using namespace UltraLite;
#define MAX_NAME_LEN 100
ULSqlca Tutca;

static ul_char const * ConnectionParms =
  UL_TEXT( "UID=DBA;PWD=sql;" )
  UL_TEXT( "DBF=C:\\temp\\ULCustomer.udb" );

ul_error_action UL_GENNED_FN_MOD MyErrorCallBack(
 SQLCA *  Tutca,
 ul_void * user_data,
 ul_char * message_param )
{
    ul_error_action rc;

    (void) user_data;

    switch( Tutca->sqlcode ){
        // The following error is used for flow control - don't report it
here
        case SQLE_NOTFOUND:
            rc = UL_ERROR_ACTION_CONTINUE;
        break;

        default:
            if (Tutca->sqlcode >= 0) { // warning or success
                rc = UL_ERROR_ACTION_DEFAULT;
            } else { // negative is real error
                _tprintf( _TEXT( "Error %ld: %s\n" ), Tutca->sqlcode,
message_param );
                rc = UL_ERROR_ACTION_CANCEL;
            }
        break;
     }
    return rc;
 }
Connection * open_conn( DatabaseManager * dm ) {
  Connection * conn = dm->OpenConnection( Tutca, ConnectionParms );
  if( conn == UL_NULL ) {
      _tprintf( _TEXT("Unable to open existing database.\n") );
   }
  return conn;
}
// Open table, insert 1 row if table is currently empty

bool do_insert( Connection * conn ) {
  Table * table = conn->OpenTable( UL_TEXT("ULCustomer") );
  if( table == UL_NULL ) {
      _tprintf( _TEXT("Table not found: ULCustomer\n") );
      return  false;
  }
  if( table->GetRowCount() == 0 ) {
      _tprintf( _TEXT("Inserting one row.\n") );
      table->InsertBegin();
      table->Set( UL_TEXT("cust_name"), UL_TEXT("New Customer") );
      table->Insert();
```

```
        conn->Commit();

    } else {
        _tprintf( _TEXT("The table has %lu rows\n"),
        table->GetRowCount() );
    }
    table->Release();
    return true;
}// Open table, display data from all rows

bool do_select( Connection * conn ) {
    Table * table = conn->OpenTable( _TEXT("ULCustomer") );
    if( table == UL_NULL ) {
    return false;
    }
    TableSchema * schema = table->GetSchema();
    if( schema == UL_NULL ) {
      table->Release();
      return false;
    }
    ul_column_num id_cid = schema->GetColumnID( UL_TEXT("cust_id") );
    ul_column_num cname_cid = schema->GetColumnID( UL_TEXT("cust_name") );

    schema->Release();

    _tprintf( _TEXT("\n\nTable 'ULCustomer' row contents:\n") );

    while( table->Next() ) {
      ul_char cname[ MAX_NAME_LEN ];

      table->Get( cname_cid ).GetString( cname, MAX_NAME_LEN );

      _tprintf( _TEXT("id=%d, name=%s \n"), (int)table->Get( id_cid ), cname );
    }
    table->Release();
    return true;
}
// sync database with MobiLink connection to reference database

bool do_sync( Connection * conn ) {
    ul_synch_info info;
    ul_stream_error * se = &info.stream_error;

    ULEnableTcpipSynchronization( Tutca.GetCA() );
    conn->InitSynchInfo( &info );
    info.stream = ULSocketStream();
    info.version = UL_TEXT( "custdb 11.0" );
    info.user_name = UL_TEXT( "50" );
    info.download_only = true;
    if( !conn->Synchronize( &info ) ) {
        _tprintf( _TEXT("Synchronization error \n" ));
     _tprintf( _TEXT("  stream_error_code is '%lu'\n"), se-
>stream_error_code );
     _tprintf( _TEXT("  system_error_code is '%ld'\n"), se-
>system_error_code );
     _tprintf( _TEXT("  error_string is '") );
     _tprintf( _TEXT("%s"), se->error_string );
     _tprintf( _TEXT("'\n") );
     return false;
    }
    return true;
}
```

```
int main() {
  ul_char buffer[ MAX_NAME_LEN ];

  Connection * conn;

  Tutca.Initialize();

  ULRegisterErrorCallback(
    Tutca.GetCA(), MyErrorCallBack,
    UL_NULL, buffer, sizeof (buffer));

  DatabaseManager * dm = ULInitDatabaseManager( Tutca );

  if( dm == UL_NULL ){
    //  You  may  have  mismatched  UNICODE  vs.  ANSI  runtimes.
 Tutca.Finalize();
    return 1;
  }

  conn = open_conn( dm );

  if( conn == UL_NULL ){
   dm->Shutdown( Tutca );
      Tutca.Finalize();
      return 1;
  }

  do_insert (conn);
  do_sync (conn);
  do_select (conn);


  dm->Shutdown( Tutca );
  Tutca.Finalize();
  return 0;
}
```

Glossary

Glossary

Adaptive Server Anywhere (ASA)

The relational database server component of SQL Anywhere Studio, intended for use in mobile and embedded environments or as a server for small and medium-sized businesses. In version 10.0.0, Adaptive Server Anywhere was renamed SQL Anywhere Server, and SQL Anywhere Studio was renamed SQL Anywhere.

See also: “SQL Anywhere” on page 359.

agent ID

See also: “client message store ID” on page 337.

article

In MobiLink or SQL Remote, an article is a database object that represents a whole table, or a subset of the columns and rows in a table. Articles are grouped together in a publication.

See also:

- “replication” on page 357
- “publication” on page 354

atomic transaction

A transaction that is guaranteed to complete successfully or not at all. If an error prevents part of an atomic transaction from completing, the transaction is rolled back to prevent the database from being left in an inconsistent state.

base table

Permanent tables for data. Tables are sometimes called **base tables** to distinguish them from temporary tables and views.

See also:

- “temporary table” on page 361
- “view” on page 363

bit array

A bit array is a type of array data structure that is used for efficient storage of a sequence of bits. A bit array is similar to a character string, except that the individual pieces are 0s (zeros) and 1s (ones) instead of characters. Bit arrays are typically used to hold a string of Boolean values.

business rule

A guideline based on real-world requirements. Business rules are typically implemented through check constraints, user-defined data types, and the appropriate use of transactions.

See also:

- “constraint” on page 339
- “user-defined data type” on page 363

carrier

A MobiLink object, stored in MobiLink system tables or a Notifier properties file, that contains information about a public carrier for use by server-initiated synchronization.

See also: “server-initiated synchronization” on page 358.

character set

A character set is a set of symbols, including letters, digits, spaces, and other symbols. An example of a character set is ISO-8859-1, also known as Latin1.

See also:

- “code page” on page 337
- “encoding” on page 343
- “collation” on page 337

check constraint

A restriction that enforces specified conditions on a column or set of columns.

See also:

- “constraint” on page 339
- “foreign key constraint” on page 344
- “primary key constraint” on page 354
- “unique constraint” on page 362

checkpoint

The point at which all changes to the database are saved to the database file. At other times, committed changes are saved only to the transaction log.

checksum

The calculated number of bits of a database page that is recorded with the database page itself. The checksum allows the database management system to validate the integrity of the page by ensuring that the numbers match as the page is being written to disk. If the counts match, it's assumed that page was successfully written.

client message store

In QAnywhere, a SQL Anywhere database on the remote device that stores messages.

client message store ID

In QAnywhere, a MobiLink remote ID that uniquely identifies a client message store.

client/server

A software architecture where one application (the client) obtains information from and sends information to another application (the server). The two applications often reside on different computers connected by a network.

code page

A code page is an encoding that maps characters of a character set to numeric representations, typically an integer between 0 and 255. An example of a code page is Windows code page 1252. For the purposes of this documentation, code page and encoding are interchangeable terms.

See also:

- "character set" on page 336
- "encoding" on page 343
- "collation" on page 337

collation

A combination of a character set and a sort order that defines the properties of text in the database. For SQL Anywhere databases, the default collation is determined by the operating system and language on which the server is running; for example, the default collation on English Windows systems is 1252LATIN1. A collation, also called a collating sequence, is used for comparing and sorting strings.

See also:

- "character set" on page 336
- "code page" on page 337
- "encoding" on page 343

command file

A text file containing SQL statements. Command files can be built manually, or they can be built automatically by database utilities. The dbunload utility, for example, creates a command file consisting of the SQL statements necessary to recreate a given database.

communication stream

In MobiLink, the network protocol used for communication between the MobiLink client and the MobiLink server.

concurrency

The simultaneous execution of two or more independent, and possibly competing, processes. SQL Anywhere automatically uses locking to isolate transactions and ensure that each concurrent application sees a consistent set of data.

See also:

- "transaction" on page 361
- "isolation level" on page 347

conflict resolution

In MobiLink, conflict resolution is logic that specifies what to do when two users modify the same row on different remote databases.

connection ID

A unique number that identifies a given connection between a client application and the database. You can determine the current connection ID using the following SQL statement:

```
SELECT CONNECTION_PROPERTY( 'Number' );
```

connection-initiated synchronization

A form of MobiLink server-initiated synchronization in which synchronization is initiated when there are changes to connectivity.

See also: "server-initiated synchronization" on page 358.

connection profile

A set of parameters that are required to connect to a database, such as user name, password, and server name, that is stored and used as a convenience.

consolidated database

In distributed database environments, a database that stores the master copy of the data. In case of conflict or discrepancy, the consolidated database is considered to have the primary copy of the data.

See also:

- "synchronization" on page 361
- "replication" on page 357

constraint

A restriction on the values contained in a particular database object, such as a table or column. For example, a column may have a uniqueness constraint, which requires that all values in the column be different. A table may have a foreign key constraint, which specifies how the information in the table relates to data in some other table.

See also:

- "check constraint" on page 336
- "foreign key constraint" on page 344
- "primary key constraint" on page 354
- "unique constraint" on page 362

contention

The act of competing for resources. For example, in database terms, two or more users trying to edit the same row of a database contend for the rights to edit that row.

correlation name

The name of a table or view that is used in the FROM clause of a query—either its original name, or an alternate name, that is defined in the FROM clause.

creator ID

In UltraLite Palm OS applications, an ID that is assigned when the application is created.

cursor

A named linkage to a result set, used to access and update rows from a programming interface. In SQL Anywhere, cursors support forward and backward movement through the query results. Cursors consist of two parts: the cursor result set, typically defined by a SELECT statement; and the cursor position.

See also:

- "cursor result set" on page 339
- "cursor position" on page 339

cursor position

A pointer to one row within the cursor result set.

See also:

- "cursor" on page 339
- "cursor result set" on page 339

cursor result set

The set of rows resulting from a query that is associated with a cursor.

See also:

- “cursor” on page 339
- “cursor position” on page 339

data cube

A multi-dimensional result set with each dimension reflecting a different way to group and sort the same results. Data cubes provide complex information about data that would otherwise require self-join queries and correlated subqueries. Data cubes are a part of OLAP functionality.

data definition language (DDL)

The subset of SQL statements for defining the structure of data in the database. DDL statements create, modify, and remove database objects, such as tables and users.

data manipulation language (DML)

The subset of SQL statements for manipulating data in the database. DML statements retrieve, insert, update, and delete data in the database.

data type

The format of data, such as CHAR or NUMERIC. In the ANSI SQL standard, data types can also include a restriction on size, character set, and collation.

See also: “domain” on page 342.

database

A collection of tables that are related by primary and foreign keys. The tables hold the information in the database. The tables and keys together define the structure of the database. A database management system accesses this information.

See also:

- “foreign key” on page 344
- “primary key” on page 354
- “database management system (DBMS)” on page 341
- “relational database management system (RDBMS)” on page 356

database administrator (DBA)

The user with the permissions required to maintain the database. The DBA is generally responsible for all changes to a database schema, and for managing users and groups. The role of database administrator is automatically built into databases as user ID DBA with password sql.

database connection

A communication channel between a client application and the database. A valid user ID and password are required to establish a connection. The privileges granted to the user ID determine the actions that can be carried out during the connection.

database file

A database is held in one or more database files. There is an initial file, and subsequent files are called dbspaces. Each table, including its indexes, must be contained within a single database file.

See also: “dbspace” on page 342.

database management system (DBMS)

A collection of programs that allow you to create and use databases.

See also: “relational database management system (RDBMS)” on page 356.

database name

The name given to a database when it is loaded by a server. The default database name is the root of the initial database file.

See also: “database file” on page 341.

database object

A component of a database that contains or receives information. Tables, indexes, views, procedures, and triggers are database objects.

database owner (dbo)

A special user that owns the system objects not owned by SYS.

See also:

- “database administrator (DBA)” on page 340
- “SYS” on page 361

database server

A computer program that regulates all access to information in a database. SQL Anywhere provides two types of servers: network servers and personal servers.

DBA authority

The level of permission that enables a user to do administrative activity in the database. The DBA user has DBA authority by default.

See also: “database administrator (DBA)” on page 340.

dbspace

An additional database file that creates more space for data. A database can be held in up to 13 separate files (an initial file and 12 dbspaces). Each table, together with its indexes, must be contained in a single database file. The SQL command CREATE DBSPACE adds a new file to the database.

See also: "database file" on page 341.

deadlock

A state where a set of transactions arrives at a place where none can proceed.

device tracking

In MobiLink server-initiated synchronization, functionality that allows you to address messages using the MobiLink user name that identifies a device.

See also: "server-initiated synchronization" on page 358.

direct row handling

In MobiLink, a way to synchronize table data to sources other than the MobiLink-supported consolidated databases. You can implement both uploads and downloads with direct row handling.

See also:

- "consolidated database" on page 338
- "SQL-based synchronization" on page 359

domain

Aliases for built-in data types, including precision and scale values where applicable, and optionally including DEFAULT values and CHECK conditions. Some domains, such as the monetary data types, are pre-defined in SQL Anywhere. Also called user-defined data type.

See also: "data type" on page 340.

download

The stage in synchronization where data is transferred from the consolidated database to a remote database.

dynamic SQL

SQL that is generated programmatically by your program before it is executed. UltraLite dynamic SQL is a variant designed for small-footprint devices.

EBF

Express Bug Fix. An express bug fix is a subset of the software with one or more bug fixes. The bug fixes are listed in the release notes for the update. Bug fix updates may only be applied to installed software with the same version number. Some testing has been performed on the software, but the software has not undergone full testing. You should not distribute these files with your application unless you have verified the suitability of the software yourself.

embedded SQL

A programming interface for C programs. SQL Anywhere embedded SQL is an implementation of the ANSI and IBM standard.

encoding

Also known as character encoding, an encoding is a method by which each character in a character set is mapped onto one or more bytes of information, typically represented as a hexadecimal number. An example of an encoding is UTF-8.

See also:

- "character set" on page 336
- "code page" on page 337
- "collation" on page 337

event model

In MobiLink, the sequence of events that make up a synchronization, such as begin_synchronization and download_cursor. Events are invoked if a script is created for them.

external login

An alternate login name and password used when communicating with a remote server. By default, SQL Anywhere uses the names and passwords of its clients whenever it connects to a remote server on behalf of those clients. However, this default can be overridden by creating external logins. External logins are alternate login names and passwords used when communicating with a remote server.

extraction

In SQL Remote replication, the act of unloading the appropriate structure and data from the consolidated database. This information is used to initialize the remote database.

See also: "replication" on page 357.

failover

Switching to a redundant or standby server, system, or network on failure or unplanned termination of the active server, system, or network. Failover happens automatically.

FILE

In SQL Remote replication, a message system that uses shared files for exchanging replication messages. This is useful for testing and for installations without an explicit message-transport system.

See also:"replication" on page 357.

file-based download

In MobiLink, a way to synchronize data in which downloads are distributed as files, allowing offline distribution of synchronization changes.

file-definition database

In MobiLink, a SQL Anywhere database that is used for creating download files.

See also: “file-based download” on page 344.

foreign key

One or more columns in a table that duplicate the primary key values in another table. Foreign keys establish relationships between tables.

See also:

- “primary key” on page 354
- “foreign table” on page 344

foreign key constraint

A restriction on a column or set of columns that specifies how the data in the table relates to the data in some other table. Imposing a foreign key constraint on a set of columns makes those columns the foreign key.

See also:

- “constraint” on page 339
- “check constraint” on page 336
- “primary key constraint” on page 354
- “unique constraint” on page 362

foreign table

The table containing the foreign key.

See also: “foreign key” on page 344.

full backup

A backup of the entire database, and optionally, the transaction log. A full backup contains all the information in the database and provides protection in the event of a system or media failure.

See also: “incremental backup” on page 346.

gateway

A MobiLink object, stored in MobiLink system tables or a Notifier properties file, that contains information about how to send messages for server-initiated synchronization.

See also: “server-initiated synchronization” on page 358.

generated join condition

A restriction on join results that is automatically generated. There are two types: key and natural. Key joins are generated when you specify KEY JOIN or when you specify the keyword JOIN but do not use the keywords CROSS, NATURAL, or ON. For a key join, the generated join condition is based on foreign key relationships between tables. Natural joins are generated when you specify NATURAL JOIN; the generated join condition is based on common column names in the two tables.

See also:

- "join" on page 348
- "join condition" on page 348

generation number

In MobiLink, a mechanism for forcing remote databases to upload data before applying any more download files.

See also: "file-based download" on page 344.

global temporary table

A type of temporary table for which data definitions are visible to all users until explicitly dropped. Global temporary tables let each user open their own identical instance of a table. By default, rows are deleted on commit, and rows are always deleted when the connection is ended.

See also:

- "temporary table" on page 361
- "local temporary table" on page 348

grant option

The level of permission that allows a user to grant permissions to other users.

hash

A hash is an index optimization that transforms index entries into keys. An index hash aims to avoid the expensive operation of finding, loading, and then unpacking the rows to determine the indexed value, by including enough of the actual row data with its row ID.

histogram

The most important component of column statistics, histograms are a representation of data distribution. SQL Anywhere maintains histograms to provide the optimizer with statistical information about the distribution of values in columns.

iAnywhere JDBC driver

The iAnywhere JDBC driver provides a JDBC driver that has some performance benefits and feature benefits compared to the pure Java jConnect JDBC driver, but which is not a pure-Java solution. The iAnywhere JDBC driver is recommended in most cases.

See also:

- "JDBC" on page 347
- "jConnect" on page 347

identifier

A string of characters used to reference a database object, such as a table or column. An identifier may contain any character from A through Z, a through z, 0 through 9, underscore (_), at sign (@), number sign (#), or dollar sign ($).

incremental backup

A backup of the transaction log only, typically used between full backups.

See also: "transaction log" on page 361.

index

A sorted set of keys and pointers associated with one or more columns in a base table. An index on one or more columns of a table can improve performance.

InfoMaker

A reporting and data maintenance tool that lets you create sophisticated forms, reports, graphs, cross-tabs, and tables, and applications that use these reports as building blocks.

inner join

A join in which rows appear in the result set only if both tables satisfy the join condition. Inner joins are the default.

See also:

- "join" on page 348
- "outer join" on page 352

integrated login

A login feature that allows the same single user ID and password to be used for operating system logins, network logins, and database connections.

integrity

Adherence to rules that ensure that data is correct and accurate, and that the relational structure of the database is intact.

See also: “referential integrity” on page 356.

Interactive SQL

A SQL Anywhere application that allows you to query and alter data in your database, and modify the structure of your database. Interactive SQL provides a pane for you to enter SQL statements, and panes that return information about how the query was processed and the result set.

isolation level

The degree to which operations in one transaction are visible to operations in other concurrent transactions. There are four isolation levels, numbered 0 through 3. Level 3 provides the highest level of isolation. Level 0 is the default setting. SQL Anywhere also supports three snapshot isolation levels: snapshot, statement-snapshot, and readonly-statement-snapshot.

See also: “snapshot isolation” on page 359.

JAR file

Java archive file. A compressed file format consisting of a collection of one or more packages used for Java applications. It includes all the resources necessary to install and run a Java program in a single compressed file.

Java class

The main structural unit of code in Java. It is a collection of procedures and variables grouped together because they all relate to a specific, identifiable category.

jConnect

A Java implementation of the JavaSoft JDBC standard. It provides Java developers with native database access in multi-tier and heterogeneous environments. However, the iAnywhere JDBC driver is the preferred JDBC driver for most cases.

See also:

- “JDBC” on page 347
- “iAnywhere JDBC driver” on page 346

JDBC

Java Database Connectivity. A SQL-language programming interface that allows Java applications to access relational data. The preferred JDBC driver is the iAnywhere JDBC driver.

See also:

- "jConnect" on page 347
- "iAnywhere JDBC driver" on page 346

join

A basic operation in a relational system that links the rows in two or more tables by comparing the values in specified columns.

join condition

A restriction that affects join results. You specify a join condition by inserting an ON clause or WHERE clause immediately after the join. In the case of natural and key joins, SQL Anywhere generates a join condition.

See also:

- "join" on page 348
- "generated join condition" on page 345

join type

SQL Anywhere provides four types of joins: cross join, key join, natural join, and joins using an ON clause.

See also: "join" on page 348.

light weight poller

In MobiLink server-initiated synchronization, a device application that polls for push notifications from a MobiLink server.

See also: "server-initiated synchronization" on page 358.

Listener

A program, dblsn, that is used for MobiLink server-initiated synchronization. Listeners are installed on remote devices and configured to initiate actions on the device when they receive push notifications.

See also: "server-initiated synchronization" on page 358.

local temporary table

A type of temporary table that exists only for the duration of a compound statement or until the end of the connection. Local temporary tables are useful when you need to load a set of data only once. By default, rows are deleted on commit.

See also:

- "temporary table" on page 361
- "global temporary table" on page 345

lock

A concurrency control mechanism that protects the integrity of data during the simultaneous execution of multiple transactions. SQL Anywhere automatically applies locks to prevent two connections from changing the same data at the same time, and to prevent other connections from reading data that is in the process of being changed.

You control locking by setting the isolation level.

See also:

- "isolation level" on page 347
- "concurrency" on page 338
- "integrity" on page 347

log file

A log of transactions maintained by SQL Anywhere. The log file is used to ensure that the database is recoverable in the event of a system or media failure, to improve database performance, and to allow data replication using SQL Remote.

See also:

- "transaction log" on page 361
- "transaction log mirror" on page 362
- "full backup" on page 344

logical index

A reference (pointer) to a physical index. There is no indexing structure stored on disk for a logical index.

LTM

Log Transfer Manager (LTM) also called Replication Agent. Used with Replication Server, the LTM is the program that reads a database transaction log and sends committed changes to Sybase Replication Server.

See: "Replication Server" on page 357.

maintenance release

A maintenance release is a complete set of software that upgrades installed software from an older version with the same major version number (version number format is *major.minor.patch.build*). Bug fixes and other changes are listed in the release notes for the upgrade.

materialized view

A materialized view is a view that has been computed and stored on disk. Materialized views have characteristics of both views (they are defined using a query specification), and of tables (they allow most table operations to be performed on them).

See also:

- “base table” on page 335
- “view” on page 363

message log

A log where messages from an application such as a database server or MobiLink server can be stored. This information can also appear in a messages window or be logged to a file. The message log includes informational messages, errors, warnings, and messages from the MESSAGE statement.

message store

In QAnywhere, databases on the client and server device that store messages.

See also:

- “client message store” on page 337
- “server message store” on page 359

message system

In SQL Remote replication, a protocol for exchanging messages between the consolidated database and a remote database. SQL Anywhere includes support for the following message systems: FILE, FTP, and SMTP.

See also:

- “replication” on page 357
- “FILE” on page 343

message type

In SQL Remote replication, a database object that specifies how remote users communicate with the publisher of a consolidated database. A consolidated database may have several message types defined for it; this allows different remote users to communicate with it using different message systems.

See also:

- “replication” on page 357
- “consolidated database” on page 338

metadata

Data about data. Metadata describes the nature and content of other data.

See also: “schema” on page 358.

mirror log

See also: “transaction log mirror” on page 362.

MobiLink

A session-based synchronization technology designed to synchronize UltraLite and SQL Anywhere remote databases with a consolidated database.

See also:

- "consolidated database" on page 338
- "synchronization" on page 361
- "UltraLite" on page 362

MobiLink client

There are two kinds of MobiLink clients. For SQL Anywhere remote databases, the MobiLink client is the dbmlsync command line utility. For UltraLite remote databases, the MobiLink client is built in to the UltraLite runtime library.

MobiLink Monitor

A graphical tool for monitoring MobiLink synchronizations.

MobiLink server

The computer program that runs MobiLink synchronization, mlsrv11.

MobiLink system table

System tables that are required by MobiLink synchronization. They are installed by MobiLink setup scripts into the MobiLink consolidated database.

MobiLink user

A MobiLink user is used to connect to the MobiLink server. You create the MobiLink user on the remote database and register it in the consolidated database. MobiLink user names are entirely independent of database user names.

network protocol

The type of communication, such as TCP/IP or HTTP.

network server

A database server that accepts connections from computers sharing a common network.

See also: "personal server" on page 353.

normalization

The refinement of a database schema to eliminate redundancy and improve organization according to rules based on relational database theory.

Notifier

A program that is used by MobiLink server-initiated synchronization. Notifiers are integrated into the MobiLink server. They check the consolidated database for push requests, and send push notifications.

See also:

- "server-initiated synchronization" on page 358
- "Listener" on page 348

object tree

In Sybase Central, the hierarchy of database objects. The top level of the object tree shows all products that your version of Sybase Central supports. Each product expands to reveal its own sub-tree of objects.

See also: "Sybase Central" on page 360.

ODBC

Open Database Connectivity. A standard Windows interface to database management systems. ODBC is one of several interfaces supported by SQL Anywhere.

ODBC Administrator

A Microsoft program included with Windows operating systems for setting up ODBC data sources.

ODBC data source

A specification of the data a user wants to access via ODBC, and the information needed to get to that data.

outer join

A join that preserves all the rows in a table. SQL Anywhere supports left, right, and full outer joins. A left outer join preserves the rows in the table to the left of the join operator, and returns a null when a row in the right table does not satisfy the join condition. A full outer join preserves all the rows from both tables.

See also:

- "join" on page 348
- "inner join" on page 346

package

In Java, a collection of related classes.

parse tree

An algebraic representation of a query.

PDB

A Palm database file.

performance statistic

A value reflecting the performance of the database system. The CURRREAD statistic, for example, represents the number of file reads issued by the database server that have not yet completed.

personal server

A database server that runs on the same computer as the client application. A personal database server is typically used by a single user on a single computer, but it can support several concurrent connections from that user.

physical index

The actual indexing structure of an index, as it is stored on disk.

plug-in module

In Sybase Central, a way to access and administer a product. Plug-ins are usually installed and registered automatically with Sybase Central when you install the respective product. Typically, a plug-in appears as a top-level container, in the Sybase Central main window, using the name of the product itself; for example, SQL Anywhere.

See also: "Sybase Central" on page 360.

policy

In QAnywhere, the way you specify when message transmission should occur.

polling

In MobiLink server-initiated synchronization, the way a light weight poller, such as the MobiLink Listener, requests push notifications from a Notifier.

See also: "server-initiated synchronization" on page 358.

PowerDesigner

A database modeling application. PowerDesigner provides a structured approach to designing a database or data warehouse. SQL Anywhere includes the Physical Data Model component of PowerDesigner.

PowerJ

A Sybase product for developing Java applications.

predicate

A conditional expression that is optionally combined with the logical operators AND and OR to make up the set of conditions in a WHERE or HAVING clause. In SQL, a predicate that evaluates to UNKNOWN is interpreted as FALSE.

primary key

A column or list of columns whose values uniquely identify every row in the table.

See also: “foreign key” on page 344.

primary key constraint

A uniqueness constraint on the primary key columns. A table can have only one primary key constraint.

See also:

- “constraint” on page 339
- “check constraint” on page 336
- “foreign key constraint” on page 344
- “unique constraint” on page 362
- “integrity” on page 347

primary table

The table containing the primary key in a foreign key relationship.

proxy table

A local table containing metadata used to access a table on a remote database server as if it were a local table.

See also: “metadata” on page 350.

publication

In MobiLink or SQL Remote, a database object that identifies data that is to be synchronized. In MobiLink, publications exist only on the clients. A publication consists of articles. SQL Remote users can receive a publication by subscribing to it. MobiLink users can synchronize a publication by creating a synchronization subscription to it.

See also:

- “replication” on page 357
- “article” on page 335
- “publication update” on page 354

publication update

In SQL Remote replication, a list of changes made to one or more publications in one database. A publication update is sent periodically as part of a replication message to the remote database(s).

See also:

- “replication” on page 357
- “publication” on page 354

publisher

In SQL Remote replication, the single user in a database who can exchange replication messages with other replicating databases.

See also: "replication" on page 357.

push notification

In QAnywhere, a special message delivered from the server to a QAnywhere client that prompts the client to initiate a message transmission. In MobiLink server-initiated synchronization, a special message delivered from a Notifer to a device that contains push request data and internal information.

See also:

- "QAnywhere" on page 355
- "server-initiated synchronization" on page 358

push request

In MobiLink server-initiated synchronization, a row of values in a result set that a Notifier checks to determine if push notifications need to be sent to a device.

See also: "server-initiated synchronization" on page 358.

QAnywhere

Application-to-application messaging, including mobile device to mobile device and mobile device to and from the enterprise, that permits communication between custom programs running on mobile or wireless devices and a centrally located server application.

QAnywhere agent

In QAnywhere, a process running on the client device that monitors the client message store and determines when message transmission should occur.

query

A SQL statement or group of SQL statements that access and/or manipulate data in a database.

See also: "SQL" on page 359.

Redirector

A web server plug-in that routes requests and responses between a client and the MobiLink server. This plug-in also implements load-balancing and failover mechanisms.

reference database

In MobiLink, a SQL Anywhere database used in the development of UltraLite clients. You can use a single SQL Anywhere database as both reference and consolidated database during development. Databases made with other products cannot be used as reference databases.

referencing object

An object, such as a view, whose definition directly references another object in the database, such as a table.

See also: "foreign key" on page 344.

referenced object

An object, such as a table, that is directly referenced in the definition of another object, such as a view.

See also: "primary key" on page 354.

referential integrity

Adherence to rules governing data consistency, specifically the relationships between the primary and foreign key values in different tables. To have referential integrity, the values in each foreign key must correspond to the primary key values of a row in the referenced table.

See also:

- "primary key" on page 354
- "foreign key" on page 344

regular expression

A regular expression is a sequence of characters, wildcards, and operators that defines a pattern to search for within a string.

relational database management system (RDBMS)

A type of database management system that stores data in the form of related tables.

See also: "database management system (DBMS)" on page 341.

remote database

In MobiLink or SQL Remote, a database that exchanges data with a consolidated database. Remote databases may share all or some of the data in the consolidated database.

See also:

- "synchronization" on page 361
- "consolidated database" on page 338

REMOTE DBA authority

In SQL Remote, a level of permission required by the Message Agent (dbremote). In MobiLink, a level of permission required by the SQL Anywhere synchronization client (dbmlsync). When the Message Agent (dbremote) or synchronization client connects as a user who has this authority, it has full DBA access. The user ID has no additional permissions when not connected through the Message Agent (dbremote) or synchronization client (dbmlsync).

See also: "DBA authority" on page 341.

remote ID

A unique identifier in SQL Anywhere and UltraLite databases that is used by MobiLink. The remote ID is initially set to NULL and is set to a GUID during a database's first synchronization.

replication

The sharing of data among physically distinct databases. Sybase has three replication technologies: MobiLink, SQL Remote, and Replication Server.

Replication Agent

See: "LTM" on page 349.

replication frequency

In SQL Remote replication, a setting for each remote user that determines how often the publisher's message agent should send replication messages to that remote user.

See also: "replication" on page 357.

replication message

In SQL Remote or Replication Server, a communication sent between a publishing database and a subscribing database. Messages contain data, passthrough statements, and information required by the replication system.

See also:

- "replication" on page 357
- "publication update" on page 354

Replication Server

A Sybase connection-based replication technology that works with SQL Anywhere and Adaptive Server Enterprise. It is intended for near-real time replication between a few databases.

See also: "LTM" on page 349.

role

In conceptual database modeling, a verb or phrase that describes a relationship from one point of view. You can describe each relationship with two roles. Examples of roles are "contains" and "is a member of."

role name

The name of a foreign key. This is called a role name because it names the relationship between the foreign table and primary table. By default, the role name is the table name, unless another foreign key is already using that name, in which case the default role name is the table name followed by a three-digit unique number. You can also create the role name yourself.

See also: "foreign key" on page 344.

rollback log

A record of the changes made during each uncommitted transaction. In the event of a ROLLBACK request or a system failure, uncommitted transactions are reversed out of the database, returning the database to its former state. Each transaction has a separate rollback log, which is deleted when the transaction is complete.

See also: “transaction” on page 361.

row-level trigger

A trigger that executes once for each row that is changed.

See also:

- “trigger” on page 362
- “statement-level trigger” on page 360

schema

The structure of a database, including tables, columns, and indexes, and the relationships between them.

script

In MobiLink, code written to handle MobiLink events. Scripts programmatically control data exchange to meet business needs.

See also: “event model” on page 343.

script-based upload

In MobiLink, a way to customize the upload process as an alternative to using the log file.

script version

In MobiLink, a set of synchronization scripts that are applied together to create a synchronization.

secured feature

A feature specified by the -sf option when a database server is started, so it is not available for any database running on that database server.

server-initiated synchronization

A way to initiate MobiLink synchronization from the MobiLink server.

server management request

A QAnywhere message that is formatted as XML and sent to the QAnywhere system queue as a way to administer the server message store or monitor QAnywhere applications.

server message store

In QAnywhere, a relational database on the server that temporarily stores messages until they are transmitted to a client message store or JMS system. Messages are exchanged between clients via the server message store.

service

In Windows operating systems, a way of running applications when the user ID running the application is not logged on.

session-based synchronization

A type of synchronization where synchronization results in consistent data representation across both the consolidated and remote databases. MobiLink is session-based.

snapshot isolation

A type of isolation level that returns a committed version of the data for transactions that issue read requests. SQL Anywhere provides three snapshot isolation levels: snapshot, statement-snapshot, and readonly-statement-snapshot. When using snapshot isolation, read operations do not block write operations.

See also: “isolation level” on page 347.

SQL

The language used to communicate with relational databases. ANSI has defined standards for SQL, the latest of which is SQL-2003. SQL stands, unofficially, for Structured Query Language.

SQL Anywhere

The relational database server component of SQL Anywhere that is intended for use in mobile and embedded environments or as a server for small and medium-sized businesses. SQL Anywhere is also the name of the package that contains the SQL Anywhere RDBMS, the UltraLite RDBMS, MobiLink synchronization software, and other components.

SQL-based synchronization

In MobiLink, a way to synchronize table data to MobiLink-supported consolidated databases using MobiLink events. For SQL-based synchronization, you can use SQL directly or you can return SQL using the MobiLink server APIs for Java and .NET.

SQL Remote

A message-based data replication technology for two-way replication between consolidated and remote databases. The consolidated and remote databases must be SQL Anywhere.

SQL statement

A string containing SQL keywords designed for passing instructions to a DBMS.

See also:

- “schema” on page 358
- “SQL” on page 359
- “database management system (DBMS)” on page 341

statement-level trigger

A trigger that executes after the entire triggering statement is completed.

See also:

- “trigger” on page 362
- “row-level trigger” on page 358

stored procedure

A stored procedure is a group of SQL instructions stored in the database and used to execute a set of operations or queries on a database server

string literal

A string literal is a sequence of characters enclosed in single quotes.

subquery

A SELECT statement that is nested inside another SELECT, INSERT, UPDATE, or DELETE statement, or another subquery.

There are two types of subquery: correlated and nested.

subscription

In MobiLink synchronization, a link in a client database between a publication and a MobiLink user, allowing the data described by the publication to be synchronized.

In SQL Remote replication, a link between a publication and a remote user, allowing the user to exchange updates on that publication with the consolidated database.

See also:

- “publication” on page 354
- “MobiLink user” on page 351

Sybase Central

A database management tool that provides SQL Anywhere database settings, properties, and utilities in a graphical user interface. Sybase Central can also be used for managing other Sybase products, including MobiLink.

synchronization

The process of replicating data between databases using MobiLink technology.

In SQL Remote, synchronization is used exclusively to denote the process of initializing a remote database with an initial set of data.

See also:

- "MobiLink" on page 351
- "SQL Remote" on page 359

SYS

A special user that owns most of the system objects. You cannot log in as SYS.

system object

Database objects owned by SYS or dbo.

system table

A table, owned by SYS or dbo, that holds metadata. System tables, also known as data dictionary tables, are created and maintained by the database server.

system view

A type of view, included in every database, that presents the information held in the system tables in an easily understood format.

temporary table

A table that is created for the temporary storage of data. There are two types: global and local.

See also:

- "local temporary table" on page 348
- "global temporary table" on page 345

transaction

A sequence of SQL statements that comprise a logical unit of work. A transaction is processed in its entirety or not at all. SQL Anywhere supports transaction processing, with locking features built in to allow concurrent transactions to access the database without corrupting the data. Transactions end either with a COMMIT statement, which makes the changes to the data permanent, or a ROLLBACK statement, which undoes all the changes made during the transaction.

transaction log

A file storing all changes made to a database, in the order in which they are made. It improves performance and allows data recovery in the event the database file is damaged.

transaction log mirror

An optional identical copy of the transaction log file, maintained simultaneously. Every time a database change is written to the transaction log file, it is also written to the transaction log mirror file.

A mirror file should be kept on a separate device from the transaction log, so that if either device fails, the other copy of the log keeps the data safe for recovery.

See also: “transaction log” on page 361.

transactional integrity

In MobiLink, the guaranteed maintenance of transactions across the synchronization system. Either a complete transaction is synchronized, or no part of the transaction is synchronized.

transmission rule

In QAnywhere, logic that determines when message transmission is to occur, which messages to transmit, and when messages should be deleted.

trigger

A special form of stored procedure that is executed automatically when a user runs a query that modifies the data.

See also:

- “row-level trigger” on page 358
- “statement-level trigger” on page 360
- “integrity” on page 347

UltraLite

A database optimized for small, mobile, and embedded devices. Intended platforms include cell phones, pagers, and personal organizers.

UltraLite runtime

An in-process relational database management system that includes a built-in MobiLink synchronization client. The UltraLite runtime is included in the libraries used by each of the UltraLite programming interfaces, and in the UltraLite engine.

unique constraint

A restriction on a column or set of columns requiring that all non-null values are different. A table can have multiple unique constraints.

See also:

- “foreign key constraint” on page 344
- “primary key constraint” on page 354
- “constraint” on page 339

unload

Unloading a database exports the structure and/or data of the database to text files (SQL command files for the structure, and ASCII comma-separated files for the data). You unload a database with the Unload utility.

In addition, you can unload selected portions of your data using the UNLOAD statement.

upload

The stage in synchronization where data is transferred from a remote database to a consolidated database.

user-defined data type

See "domain" on page 342.

validate

To test for particular types of file corruption of a database, table, or index.

view

A SELECT statement that is stored in the database as an object. It allows users to see a subset of rows or columns from one or more tables. Each time a user uses a view of a particular table, or combination of tables, it is recomputed from the information stored in those tables. Views are useful for security purposes, and to tailor the appearance of database information to make data access straightforward.

window

The group of rows over which an analytic function is performed. A window may contain one, many, or all rows of data that has been partitioned according to the grouping specifications provided in the window definition. The window moves to include the number or range of rows needed to perform the calculations for the current row in the input. The main benefit of the window construct is that it allows additional opportunities for grouping and analysis of results, without having to perform additional queries.

Windows

The Microsoft Windows family of operating systems, such as Windows Vista, Windows XP, and Windows 200x.

Windows CE

See "Windows Mobile" on page 363.

Windows Mobile

A family of operating systems produced by Microsoft for mobile devices.

work table

An internal storage area for interim results during query optimization.

Index