UltraLite®
C and C++ Programming

Version 12.0.1

January 2012

Version 12.0.1
January 2012

Contents

About this book v

UltraLite C/C++ 1

- **System requirements and supported platforms 1**
- **UltraLite C/C++ API architecture 1**
- **Developing embedded SQL applications 2**

Application development 5

- **UltraLite C++ application development 5**
- **UltraLite C++ application development using embedded SQL 27**
- **UltraLite application development for Windows Mobile 56**

Tutorials 65

- **Tutorial: Building a Windows application using the C++ API 65**
- **Tutorial: Building an iPhone application using the C++ API 75**

API reference 101

- **UltraLite C/C++ common API reference 101**
- **UltraLite C/C++ API reference 119**
- **UltraLite Embedded SQL API reference 237**

Index 285

About this book

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

UltraLite C/C++

The C/C++ interfaces provide the following benefits for UltraLite developers:

- A small, high-performance database store with native synchronization.
- The power, efficiency, and flexibility of the C or C++ language.
- The ability to deploy applications on Windows Mobile, Windows desktop platforms, Linux desktop, embedded Linux, iPhone and iPad.

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

See also

- "UltraLite database creation" [*UltraLite - Database Management and Reference*]

System requirements and supported platforms

Development platforms

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

- A Microsoft Windows, Linux, or Mac desktop as a development platform.
- A supported Microsoft or GNU C/C++ compiler.

Target platforms

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

- Windows Mobile 5.0 or later
- Windows XP or later
- Linux
- Embedded Linux
- iOS 3 and later (iPhone and iPad)
- Mac

UltraLite C/C++ API architecture

The UltraLite C/C++ API architecture is defined in the *ulcpp.h* header file. The following list describes some of the commonly used objects:

- **ULDatabaseManager** Provides methods for managing database connections, such as CreateDatabase and OpenConnection.
- **ULConnection** Represents a connection to an UltraLite database. You can create one or more ULConnection objects.
- **ULTable** Provides direct access to tables in the database.
- **ULPreparedStatement, ULResultSet, and ULResultSetSchema** Create Dynamic SQL statements, make queries and execute INSERT, UPDATE, and DELETE statements, and attain programmatic control over database result sets.

See also

- "UltraLite C/C++ API reference" on page 119

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. Design 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 BEGIN DECLARE SECTION
   int cost
   char pname[31];
EXEC SQL END DECLARE SECTION

EXEC SQL SELECT price, prod_name
   INTO :cost, :pname
   FROM ULProduct
   WHERE prod_id= :pid;
```

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.

See also

- “Embedded SQL application building” on page 54
- “UltraLite C++ application development using embedded SQL” on page 27

Application development

This section provides development notes for the UltraLite C/C++ API.

UltraLite C++ application development

Quick start to UltraLite C++ application development

The following procedure is generally used when creating an application using the UltraLite C++ API:

1. Initialize a ULDatabaseManager object.
2. (Optional) Enable features in the UltraLite runtime library.
3. Use an UltraLite database. You can open a connection to an existing database, create a new one, drop an existing database, or validate that an existing database has no file corruption.
4. Finalize the ULDatabaseManager object.

The ULDatabaseManager object should only be initialized once in your application and then finalized when your application is terminating. All methods on the ULDatabaseManager class are static. Use the ULError class to get error information throughout your UltraLite application.

See also

- "ULDatabaseManager class [UltraLite C++]" on page 145
- "Build and deploy UltraLite C++ applications" on page 23

iPhone and Mac OS X considerations

Development environment

The development environment for iPhone and Mac OS X is Xcode.

Build settings

To reference the UltraLite header files and library it is convenient to create a user-defined build setting set to the location of the SQL Anywhere installation directory. For example, set *SQLANY_ROOT* to */Applications/SQLAnywhere12*. To create this setting, open the project editor's **Build** pane and click **Add User-Defined Setting** and enter the name and value.

Include files

To find the UltraLite include files, add *$(SQLANY_ROOT)/sdk/include* to the **User Header Search Paths** *(USER_HEADER_SEARCH_PATHS*) build setting.

Unsupported MobiLink client network protocol options

UltraLite for iPhone and/or Mac OS X does not support the following MobiLink client network protocol options:

- certificate_company
- certificate_unit
- client_port
- identity
- identity_password
- network_leave_open
- network_name

Encryption

To use end-to-end encryption when synchronizing Mac OS X and iPhone UltraLite clients with a MobiLink server, you must encapsulate your public keys in a PEM encoded X509 certificate (as opposed to a PEM public key file) and supply an E2EE private key. To create a PEM encoded X509 certificate with an E2EE private key, it is recommended that you use the certificate creation utility, createcert. After you obtain an E2EE private key, specify the -x option when you start the MobiLink server and assign the key to the e2ee_private_key option. To synchronize the UltraLite client database with the MobiLink server, run the UltraLite synchronization utility, ulsync, and assign the E2EE public key to the e2ee_public_key connection option. Extracting the public key from the certificate is necessary when using both iPhone and non-iPhone clients together. When developing for iPhone UltraLite clients, the UltraLite Synchronization utility searches for the certificate file in the Main Resource Bundle (mainBundle) of the iPhone development package if the trusted_certificate or e2ee_public_key options are assigned. You must include the certificate in the Resources folder in your Xcode project.

The following encryption standards are not supported:

- ECC encryption (only RSA)
- FIPS-certified encryption

Debugging iPhone applications

The Xcode debugger (GDB) has support for stepping through and breaking on *longjmp()* calls. Applications typically do not use *longjmp*, but the UltraLite runtime library does internally (sometimes, when an error is signaled, for instance). This may cause problems when tracing through application code and stepping over UltraLite calls. If you step over an UltraLite call and get an error from the debugger: Restart the program, set a breakpoint after the problematic line and, instead of stepping over the problematic line, use the **Continue** command - this will have the same effect because the debugger will stop at the following breakpoint, but should avoid problems related to *longjmp* calls. The most likely place to encounter this is when using **OpenConnection()** to open an existing database or determine that the database doesn't exist (an error is signaled when the database doesn't exist).

See also

- "e2ee_public_key" [*MobiLink - Client Administration*]
- "Tutorial: Building an iPhone application using the C++ API" on page 75
- "MobiLink client network protocol options" [*MobiLink - Client Administration*]
- "UltraLite database security" [*UltraLite - Database Management and Reference*]
- "Certificate Creation utility (createcert)" [*SQL Anywhere Server - Database Administration*]
- "-x mlsrv12 option" [*MobiLink - Server Administration*]
- "UltraLite Synchronization utility (ulsync)" [*UltraLite - Database Management and Reference*]

Connecting to an UltraLite database

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

The ULDatabaseManager class is used to open a connection to a database. The ULDatabaseManager class returns a non-null ULConnection object when a connection is established. Use the ULConnection object to perform the following tasks:

- Commit or roll back transactions.
- Synchronize data with a MobiLink server.
- Access tables in the database.
- Work with SQL statements.
- Handle errors in your application.

Ensure you specify a writable path for the database file. Use the *NSSearchPathForDirectoriesInDomains* function to query the *NSDocumentDirectory*, for example.

Note
You can find sample code in the *%SQLANYSAMP12%\UltraLite\CustDB* directory.

Connect to an UltraLite database

1. Initialize the ULDatabaseManager object and enable features in UltraLite using the following code:

```
if( !ULDatabaseManager::Init() ) {
    return 0;
}
ULDatabaseManager::EnableAesDBEncryption();

// Use ULDatabaseManager.Fini() when terminating the app.
```

2. Open a connection to an existing database or create a new database if the specified database file does not exist using the following code:

```
ULConnection * conn;
ULError ulerr;
```

```
conn =
ULDatabaseManager::OpenConnection( "dbf=sample.udb;dbkey=aBcD1234",
&ulerr );
if( conn == NULL ) {
    if( ulerr.GetSQLCode() == SQLE_ULTRALITE_DATABASE_NOT_FOUND ) {
        conn =
ULDatabaseManager::CreateDatabase( "dbf=sample.udb;dbkey=aBcD1234",
&ulerr );
        if( conn == NULL ) {
            // write code that uses ulerr to determine what happened
            return 0;
        }
        // add code to create the schema for your database
    } else {
        // write code that uses ulerr to determine what happened
        return 0;
    }
}
assert( conn != NULL );
```

In this step, you declare a ULError object that contains error information in case the connection is not successful.

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.

See also

- "ULConnection class [UltraLite C++]" on page 119
- "ULDatabaseManager class [UltraLite C++]" on page 145
- "ULError class [UltraLite C++]" on page 159

Data creation and modification using SQL statements

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

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.

See also

- "UltraLite SQL statements" [*UltraLite - Database Management and Reference*]
- "Data creation and modification using the ULTable class" on page 15

Modifying data using INSERT, UPDATE and DELETE

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

Insert a row

Note
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.

1. Declare a ULPreparedStatement using the following code:

   ```
   ULPreparedStatement * prepStmt;
   ```

2. Prepare a SQL statement for execution.

   The following code prepares an INSERT statement for execution:

   ```
   prepStmt = conn->PrepareStatement("INSERT INTO MyTable(MyColumn1) VALUES
   (?)");
   ```

3. Check for errors when preparing the statement.

   For example, the following code is useful when checking for SQL syntax errors:

   ```
   if( prepStmt == NULL ) {
       const ULError * ulerr;
       ulerr = conn->GetLastError();
       // write code to handle the error
       return;
   }
   ```

4. Set values to replace ? characters in the prepared statement.

   The following code sets ? characters to "some value" while error checking. For example, an error is caught when the parameter ordinal is out of range for the number of parameters in the prepared statement.

   ```
   if( !prepStmt->SetParameterString(1, "some value") ) {
       const ULError * ulerr;
       ulerr = conn->GetLastError();
       // write code to handle the error
       return;
   }
   ```

5. Execute the prepared statement, inserting the data into the database.

   The following code checks for errors that could occur after executing the statement. For example, an error is returned if a duplicate index value is found in a unique index.

   ```
   bool success;
   success = prepStmt->ExecuteStatement();
   ```

```
if( !success ) {
    const ULError * ulerr;
    ulerr = conn->GetLastError();
    // write code to handle the error
} else {
    // Use the following line if you are interested in the number of rows
inserted ...
    ul_u_long rowsInserted = prepStmt->GetRowsAffectedCount();
}
```

6. Clean up the prepared statement resources.

   The following code releases the resources used by the prepared statement object. This object should no longer be accessed after the Close method is called.

   ```
   prepStmt->Close();
   ```

7. Commit the data to the database.

   The following code saves the data to the database and prevents data loss. The data from step 5 is lost if the device application terminates unexpectedly before the application completes a commit call.

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

Delete a row

Note
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.

1. Declare a ULPreparedStatement using the following code:

   ```
   ULPreparedStatement * prepStmt;
   ```

2. Prepare a SQL statement for execution.

   The following code prepares a DELETE statement for execution:

   ```
   prepStmt = conn->PrepareStatement("DELETE FROM MyTable(MyColumn1) VALUES
   (?)");
   ```

3. Check for errors when preparing the statement.

   For example, the following code is useful when checking for SQL syntax errors:

   ```
   if( prepStmt == NULL ) {
       const ULError * ulerr;
       ulerr = conn->GetLastError();
       // write code to handle the error
       return;
   }
   ```

4. Set values to replace ? characters in the prepared statement.

The following code sets ? characters to 7 while error checking. For example, an error is caught when the parameter ordinal is out of range for the number of parameters in the prepared statement.

```
ul_s_long value_to_delete = 7;
if( !prepStmt->SetParameterInt(1, value_to_delete) ) {
    const ULError * ulerr;
    ulerr = conn->GetLastError();
    // write code to handle the error.
    return;
}
```

5. Execute the prepared statement, deleting the data from the database.

   The following code checks for errors that could occur after executing the statement. For example, an error is returned if you try deleting a row that has a foreign key referenced to it.

   ```
   bool success;
   success = prepStmt->ExecuteStatement();
   if( !success ) {
       const ULError * ulerr;
       ulerr = conn->GetLastError();
       // write code to handle the error
   } else {
       // Use the following line if you are interested in the number of rows
   deleted ...
       ul_u_long rowsDeleted = prepStmt->GetRowsAffectedCount();
   }
   ```

6. Clean up the prepared statement resources.

   The following code releases the resources used by the prepared statement object. This object should no longer be accessed after the Close method is called.

   ```
   prepStmt->Close();
   ```

7. Commit the data to the database.

   The following code saves the data to the database and prevents data loss. The data from step 5 is lost if the device application terminates unexpectedly before the application completes a commit call.

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

Update a row

Note
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.

1. Declare a ULPreparedStatement using the following code:

   ```
   ULPreparedStatement * prepStmt;
   ```

2. Prepare a SQL statement for execution.

   The following code prepares an UPDATE statement for execution:

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

3. Check for errors when preparing the statement.

   For example, the following code is useful when checking for SQL syntax errors:

   ```
   if( prepStmt == NULL ) {
       const ULError * ulerr;
       ulerr = conn->GetLastError();
       // write code to handle the error
       return;
   }
   ```

4. Set values to replace ? characters in the prepared statement.

   The following code sets ? characters to integer values while error checking. For example, an error is caught when the parameter ordinal is out of range for the number of parameters in the prepared statement.

   ```
   bool success;
   success = prepStmt->SetParameterInt( 1, 25 );
   if( success ) {
       success = prepStmt->SetParameterInt( 2, -1 );
   }
   if( !success ) {
       const ULError * ulerr;
       ulerr = conn->GetLastError();
       // write code to handle the error
       return;
   }
   ```

5. Execute the prepared statement, updating the data in the database.

   The following code checks for errors that could occur after executing the statement. For example, an error is returned if a duplicate index value is found in a unique index.

   ```
   success = prepStmt->ExecuteStatement();
   if( !success ) {
       const ULError * ulerr;
       ulerr = conn->GetLastError();
       // write code to handle the error
   } else {
       // if you are interested in the number of rows updated ...
       ul_u_long rowsUpdated = prepStmt->GetRowsAffectedCount();
   }
   ```

6. Clean up the prepared statement resources.

   The following code releases the resources used by the prepared statement object. This object should no longer be accessed after the Close method is called.

   ```
   prepStmt->Close();
   ```

7. Commit the data to the database.

   The following code saves the data to the database and prevents data loss. The data from step 5 is lost if the device application terminates unexpectedly before the application completes a commit call.


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

See also

- "ULPreparedStatement class [UltraLite C++]" on page 168

Retrieving data using 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.

Execute a SELECT statement

1. Declare the required variables using the following code:

   ```
   ULPreparedStatement * prepStmt;
   ULResultSet * resultSet;
   ```

2. Prepare a SQL statement for execution.

   The following code prepares a SELECT statement for execution:

   ```
   prepStmt = conn->PrepareStatement("SELECT MyColumn1 FROM MyTable");
   ```

3. Check for errors when preparing the statement.

   For example, the following code is useful when checking for SQL syntax errors:

   ```
   if( prepStmt == NULL ) {
       const ULError * ulerr;
       ulerr = conn->GetLastError();
       // write code to handle the error
       return;
   }
   ```

4. Execute the SQL and return a result set object that can be used to move the results of the query.

   ```
   resultSet = prepStmt->ExecuteQuery();
   if( resultSet == NULL ) {
       const ULError * ulerr;
       ulerr = conn->GetLastError();
       // write code to handle the error
       prepStmt->Close();
       return;
   }
   ```

5. Traverse the rows by calling the Next method. Store the result as a string and store them in a buffer.

   The Next method moves to the next row of the result set. The ULResultSet object is positioned on a row if the call returns true; otherwise, if the call returns false, all the rows have been traversed.

   ```
   while( resultSet->Next() ) {
       char buffer[ 100 ];
       resultSet->GetString( 1, buffer, 100 );
       printf( "MyColumn = %s\n", buffer );
   }
   ```

6. Clean up the prepared statement and result set object resources.

   The prepared statement object should not accessed after the Close method is called.

   ```
   resultSet->Close();
   prepStmt->Close();
   ```

See also

- "ULPreparedStatement.ExecuteQuery method [UltraLite C++]" on page 170

Schema description creation and retrieval

The GetResultSetSchema 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 GetResultSetSchema method to display schema information in a command prompt:

```
const char * name;
int column_count;
const ULResultSetSchema & rss = prepStmt->GetResultSetSchema();
int column_count = rss.GetColumnCount();
for( int i = 1; i < column_count; i++ ) {
    name = rss.GetColumnName( i );
    printf( "id = %d, name = %s\n", i, name );
}
```

In this example, required variables are declared and the ULResultSetSchema object is assigned. It is possible to get a ULResultSetSchema object from the result set object itself, but this example demonstrates how the schema is available after the statement is prepared and before the query is executed. The number of rows in the result set are counted, and the name of each column is displayed.

See also

- "ULPreparedStatement.GetResultSetSchema method [UltraLite C++]" on page 173

SQL result set navigation

You can navigate through a result set using methods associated with the ULResultSet class.

The result set class 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 pointer position 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 also

- "ULResultSet class [UltraLite C++]" on page 177

Data creation and modification using the ULTable class

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

This section explains how to perform the following tasks using the ULTable class:

- 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.

See also

- "Data creation and modification using SQL statements" on page 8

Row navigation

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

The ULTable 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 pointer position 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 also

- "ULTable class [UltraLite C++]" on page 218

Example

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

```
char buffer[ 100 ];
ul_column_num column_id;
ULTable * tbl = conn->OpenTable( "MyTable" );
if( tbl == NULL ) {
    const ULError * ulerr;
    ulerr = conn->GetLastError();
    // write code to handle the error
    return;
}
column_id = tbl->GetTableSchema().GetColumnID( "MyColumn" );
if( column_id == 0 ) {
    // the column "MyColumn" likely does not exist.  Handle the error.
    tbl->Close();
    return;
}
while( tbl->Next() ) {
    tbl->GetString( column_id, buffer, 100 );
    printf( "%s\n", buffer );
}
tbl->Close();
```

You expose the rows of the table to the application when you open the ULTable 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 example moves to the first row of the MyTable table as ordered by the ix_col index:

```
ULTable * tbl = conn->OpenTable( "MyTable", "ix_col" );
```

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, UpdateBegin, FindBegin, and so on.

Row insertion

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.

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.

Example

The following code demonstrates new row insertion:

```
ULTable * tbl = conn->OpenTable("MyTable");
bool success;
tbl->InsertBegin();  // enter "Insert mode"
tbl->SetInt("id", 3);
tbl->SetString("lname", "Smith");
tbl->SetString("fname", "Mary");
success = tbl->Insert();
conn->Commit();
tbl->Close();
```

In this example, the tbl variable is set to open MyTable. The values for each column are set in the current row buffer; columns can be referenced name or ID. The Insert method causes the temporary row buffer values to be inserted into the database. The results are then committed and displayed. Resources are freed with the Close method.

Updating rows

The following procedure updates a row in a table.

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

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->UpdateBegin();
   ```

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->SetInt("id", 3);
   ```

4. Execute the Update.

   ```
   tbl->Update();
   ```

Caution
When using the Find and Update methods, your pointer may not be in the expected position after updating a column that is involved in the search criteria. In some instances, it is recommended that you use a SQL statement when updating several rows.

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 use the Commit method.

See also

- "Managing transactions" on page 21

Search rows with find and lookup modes

UltraLite has different modes of operation for working with data. You can use two of these modes, find and lookup, for searching. The ULTable 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 ULTable 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 ULTable object was opened.

Example

This example uses a table named MyTable that was created using the following SQL statements:

```
CREATE TABLE MyTable( id int primary key, lname char(100), fname char(100) )
CREATE INDEX ix_lname ON MyTable ( lname )
```

The following code displays all the fname column contents where lname column is "Smith":

```
ULTable * tbl = conn->OpenTable( "MyTable", "ix_lname" );
char buffer[ 100 ];
bool found;
tbl->FindBegin();  // enter "Find mode"
tbl->SetString( "lname", "Smith" );  // set pointer row buffer to "Smith"
found = tbl->FindFirst();
while( found ) {
    tbl->GetString( 3, buffer, 100 );
    printf( "%s\n", buffer );
    found = tbl->FindNext();
}
tbl->Close();
```

In this example, the tbl variable is set to open MyTable using the ix_lname index so that rows are returned in same order as the lname column. ULTable objects use the values in the row buffer when they execute a find. This buffer is specified as "Smith", as defined by the SetString method. The FindFirst method indicates that traversal should begin at the first row that has lname set to "Smith"; the pointer is positioned after the last row of the table if there are no rows where lname is set to "Smith". The fname is retrieved by the GetString method because The fname has a column ID of 3. The results are then displayed, and the resources are freed.

See also

- “ULTable class [UltraLite C++]” on page 218

Access values of the current row

A ULTable 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 ULTable 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 ULTable object provides a set of methods for retrieving column values. These methods take the column name or ID as the argument.

The following example demonstrates two ways to get an age value out of an open table, assuming that age is the first column of the table:

```
ul_s_long age1 = tbl->GetInt( 1 );
ul_s_long age2 = tbl->GetInt( "age" );
assert( age1 == age2 );
```

Using the column ID version of value retrieval has performance benefits when values are retrieved in a loop.

Modifying column values

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

For example demonstrates two ways to set a string value for a row with string columns of lname and fname, assuming that lname is the first column in the table.

```
tbl->SetString( 1, last_name );
tbl->SetString( "fname", first_name );
```

By setting column values, you do not directly alter the data in the database. You can assign values to the columns, 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();
tbl = tbl.GetInt( 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 also

- "Converting data types explicitly" [*UltraLite - Database Management and Reference*]

Deleting rows

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

The following procedure deletes a row.

Delete a row

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

   ```
   tbl->Delete();
   ```


Managing transactions

The UltraLite C++ API does not support AutoCommit mode. Transactions are started implicitly by the first statement to modify the database, and must be explicitly committed or rolled back.

Commit a transaction

- Execute a conn->Commit statement, where conn is a valid ULConnection pointer.

Roll back a transaction

- Execute a conn->Rollback statement, where conn is a valid ULConnection pointer.

See also

- "ULConnection.Commit method [UltraLite C++]" on page 123
- "ULConnection.Rollback method [UltraLite C++]" on page 138
- "UltraLite transaction processing" [*UltraLite - Database Management and Reference*]

Schema information access

You can programmatically retrieve result set or database structure descriptions. These descriptions are known as schema information, and this information is available through the UltraLite C API schema classes.

Note
You cannot modify the schema using the UltraLite C API. You can only retrieve the schema information.

You can access the following schema objects and information:

- **ULResultSetSchema** Describes a query or data in a table. It exposes the identifier, name, and type information of each column, and the number of columns in the table. ULResultSetSchema classes can be retrieved from the following classes:
  - ULPreparedStatement
  - ULResultSet
  - ULTable
- **ULDatabaseSchema** Exposes the number and names of the tables and publications in the database, and the global properties such as the format of dates and times. ULDatabaseSchema classes can be retrieved from ULConnection classes.
- **ULTableSchema** Exposes information about the column and index configurations. The column information in the ULTableSchema class complements the information available from the ULResultSetSchema class. For example, you can determine whether columns have default values or permit null values. ULTableSchema classes can be retrieved from ULTable classes.

- **ULIndexSchema** Returns information about the column in the index. ULIndexSchema classes can be retrieved from ULTableSchema classes.

The ULResultSetSchema class is returned as a constant reference unlike the ULDatabaseSchema, ULTableSchema and ULIndexSchema classes, which are returned as pointers. You cannot close a class that returns a constant reference but you must close classes that are returned as pointers.

The following code demonstrates proper and improper use of schema class closure:

```
// This code demonstrates proper use of the ULResultSetSchema class:
const ULResultSetSchema & rss = prepStmt->GetResultSetSchema();
c_count = prepStmt->GetSchema().GetColumnCount();

// This code demonstrates proper use of the ULDatabaseSchema class:
ULDatabaseSchema * dbs = conn->GetResultSetSchema();
t_count = dbs->GetTableCount();
dbs->Close();  // This line is required.

// This code demonstrates improper use of the ULDatabaseSchema class
// because the object needs to be closed using the Close method:
t_count = conn->GetResultSetSchema()->GetTableCount();
```

See also

- "ULPreparedStatement class [UltraLite C++]" on page 168
- "ULResultSet class [UltraLite C++]" on page 177
- "ULTable class [UltraLite C++]" on page 218
- "ULConnection class [UltraLite C++]" on page 119

Error handling

The UltraLite C++ API includes a ULError object that should be used to retrieve error information. Several methods in the API return a boolean value, indicating whether the method call was successful. In some instances, null is returned when an error occurs. The ULConnection object contains a GetLastError method, which returns a ULError object.

Use SQLCode to diagnose an error. In addition to the SQLCode, you can use the GetParameterCount and GetParameter methods to determine whether additional parameters exist to provide additional information about the error.

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.

See also

- "Tutorial: Building a Windows application using the C++ API" on page 65
- "ULSetErrorCallback method [UltraLite Embedded SQL]" on page 268
- "SQL Anywhere error messages sorted by Sybase error code" [*Error Messages*]

MobiLink data synchronization

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. The methods and properties of the connection object can be used to control synchronization.

See also

- "UltraLite clients" [*UltraLite - Database Management and Reference*]
- "ul_sync_info structure [UltraLite C and Embedded SQL datatypes]" on page 110
- "Synchronization parameters for UltraLite" [*UltraLite - Database Management and Reference*]

Closing the UltraLite database connection

It is important to release resources when they are no longer being used; otherwise, an UltraLite database file remains in use as long as an application has a connection to the database.

Close an UltraLite database connection

1. Call the Close method to release resources.

   Use the following code when the application no longer requires a connection to the database:

   ```
   if( conn != NULL ) {
       conn->Close( &ulerr );
   }
   ```

2. Call the Fini method to finalize the ULDatabaseManager object.

   Use the following code when closing the application.

   ```
   ULDatabaseManager.Fini();
   ```

See also

- "ULConnection.Close method [UltraLite C++]" on page 123
- "ULDatabaseManager.Fini method [UltraLite C++]" on page 153

Build and deploy UltraLite C++ applications

When building a C/C++ application that does not use the UltraLite engine, you can either link to a static UltraLite runtime library (this makes sure all the UltraLite code is linked into your application) or, on Windows and Windows Mobile, you can link to an import library and load the UltraLite runtime code dynamically when the application starts.

Deploy UltraLite for Windows and Windows Mobile devices when using static linkage

1. Specify the following connection and creation parameters:
   - When using obfuscation, set the creation parameter **obfuscate=1** when creating the database
   - When using AES, or FIPS 140-2 AES encryption, set the connection parameter **DBKEY=***encryption-key* when creating or connecting to the database
2. Follow the appropriate steps for the synchronization type used in your UltraLite application:

Synchronization Type	Parameter settings
TCP/IP	Set the **Stream** synchronization parameter to "tcpip".
HTTP	Set the **Stream** synchronization parameter to "http".
RSA TLS	Set the **Stream** synchronization parameter to "tls".
RSA HTTPS	Set the **Stream** synchronization parameter to "https".
ECC TLS	Set the **Stream** synchronization parameter to "tls". Set the protocol option **tls_type=ecc**. When using ECC E2EE encryption, set the protocol option **e2ee_type=ecc**.
ECC HTTPS	Set the **Stream** synchronization parameter to "https". Set the protocol option **tls_type=ecc**. When using ECC E2EE encryption, set the protocol option **e2ee_type=ecc**.
FIPS 140-2 RSA TLS	Set the **Stream** synchronization parameter to "tls". Set the protocol option **fips=yes**.
FIPS 140-2 RSA HTTPS	Set the **Stream** synchronization parameter to "https". Set the protocol option **fips=yes**.

3. When using RSA, ECC, or RSA FIPS 140-2 End-to-End encryption, set the protocol option **e2ee_public_key=***key-file*.
4. When using ZLIB compression, set the protocol option **compression=zlib**.
5. Link against the following files:
   - *ulrt.lib*

- *ulbase.lib*
- When using RSA TLS, or RSA HTTPS synchronization, *ulrsa.lib*
- When using ECC TLS, or ECC HTTPS synchronization, *ulecc.lib*

6. Call the following methods in your UltraLite application:

   - When using AES encryption, the ULDatabaseManager.EnableAesDBEncryption method
   - When using FIPS 140-2 AES encryption, the ULDatabaseManager.EnableAesFipsDBEncryption method

7. Ensure that the following methods are called for they synchronization type used in your UltraLite application:

   - **TCP/IP** Call the EnableTcpipSynchronization method.
   - **HTTP** Call the EnableHttpSynchronization method.
   - **TLS using RSA** Call the EnableTlsSynchronization and EnableRsaSyncEncryption methods.
   - **HTTPS using RSA** Call the EnableHttpsSynchronization and EnableRsaSyncEncryption methods.
   - **TLS using ECC** Call the EnableTlsSynchronization and EnableEccSyncEncryption methods.
   - **HTTPS using ECC** Call the EnableHttpsSynchronization and EnableEccSyncEncryption methods.
   - **TLS using FIPS 140-2 RSA** Call the EnableTlsSynchronization and EnableRsaFipsEncryption methods.
   - **HTTPS using FIPS 140-2 RSA** Call the EnableHttpsSynchronization and EnableRsaFipsSyncEncryption methods.

8. Deploy the following files:

   - When using FIPS 140-2 AES encryption, *ulfips12.dll* and *sbgse2.dll*.
   - When using RSA FIPS 140-2 TLS, or RSA FIPS 140-2 HTTPS synchronization, *sbgsse2.dll*, and *mlcrsafips12.dll*.

Linker/compiler options to build and link runtimes for Linux deployment

The linker/compiler options for *libulrt.a* are:

```
-L<$SQLANY12>/ultralite/linux/x86/586/lib -lulrt -|ulbase
```

and for the engine:

```
-L<$SQLANY12>/ultralite/linux/x86/586/lib -lulrtc -|ulbase
```

The headers command line switch is:

```
-I<$SQLANY12>/sdk/include
```

Build and link runtimes for iPhone deployment

UltraLite runtimes must be built after installation. Follow the instructions provided in *install-dir/ultralite/iphone/readme.txt*

To link to the UltraLite runtime library, either:

Control-click the **Frameworks** group and click **Add** » **Existing Files**, then navigate to the *install-dir/ultralite/iphone* directory and click *libulrt.a*.

OR

Add the following to the **Other Linker Flags** (*OTHER_LDFLAGS*) build setting:

```
-L$(SQLANY_ROOT)/ultralite/iphone
-lulrt
```

where *SQLANY_ROOT* is a custom build setting set to the SQL Anywhere installation directory.

In addition, the *CFNetwork.framework* and *Security.framework* frameworks are required. To add these, control-click the **Frameworks** group and click **Add** » **Existing Frameworks** and select from the list.

Build and link runtimes for Mac OS X deployment

To link to the UltraLite runtime library, Control-click the **Frameworks** group and click **Add** » **Existing Files**, then navigate to the */Applications/SQLAnywhere12/ultralite/macosx/x86_64* directory and click *libulrt.a* and also *libulbase.a*.

In addition, the *CoreFoundation.framework*, *CoreServices.framework*, and *Security.framework* frameworks are required. To add these, Control-click the **Frameworks** group and click **Add** » **Existing Frameworks** and select from the list.

See also

- "UltraLite application build and deployment specifications" [*UltraLite - Database Management and Reference*]

UltraLite C++ application deployment

There are two primary considerations for deploying your UltraLite C++ solution:

- Deploy the files that provide UltraLite functionality (the runtime files)
- Deploy the UltraLite database file or files (used by the runtime files and containing application data)

Ensure that you have linked to the appropriate libraries.

Build and deploy an application using the UltraLite runtime DLL

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.

1. Preprocess your code, then compile the output with UL_USE_DLL.
2. Link the UltraLite import library to your application.
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, connect to the emulator using ActiveSync and copy the DLL to the device using Windows Explorer.

See also

- "Deploying the ActiveSync provider for UltraLite" [*UltraLite - Database Management and Reference*]
- "UltraLite C++ application development" on page 5
- "Configuring UltraLite clients to use transport-layer security" [*SQL Anywhere Server - Database Administration*]

UltraLite C++ application development using embedded SQL

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

See also

- "UltraLite C/C++" on page 1
- "UltraLite Embedded SQL API reference" on page 237
- "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.

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.

See also

- "db_init method" on page 238

Embedded SQL program structure

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.

Initialize 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 import library.

The SQLCA type is defined in the header file *%SQLANY12%\SDK\Include\sqlca.h*.

After declaring the SQLCA (`EXEC SQL INCLUDE SQLCA;`), but before your application can perform 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 *%SQLANY12%\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.
- **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.

See also

- "SQL Anywhere error messages" [*Error Messages*]

Connect to an UltraLite 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.

See also

- "UltraLite connection parameters" [*UltraLite - Database Management and Reference*]
- "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.

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**.

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.

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 also

- “db_init method” on page 238
- “SET SQLCA statement [ESQL]” [*SQL Anywhere Server - SQL Reference*]
- “SET CONNECTION statement [Interactive SQL] [ESQL]” [*SQL Anywhere Server - SQL Reference*]

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.

Host variable declaration

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

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.

-Note
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 {
   a_sql_ulen 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 {
  a_sql_ulen 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 you 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.

- **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.

- **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 *%SQLANY12%\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.

See also

- "Database options" [*SQL Anywhere Server - Database Administration*]

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.

Host variable scope

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. However, 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.

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

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.

Indicator variables

Indicator variables are C variables that hold 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 a_sql_len that is placed immediately following a regular host variable in a SQL statement. 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 );
```

On a fetch or execute where no rows are received from the database server (such as when an error or end of result set occurs), then indicator values are unchanged.

Note
To allow for the future use of 32 and 64-bit lengths and indicators, the use of short int for embedded SQL indicator variables is deprecated. Use a_sql_len instead.

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

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];
a_sql_len ind_phone;
EXEC SQL END DECLARE SECTION;
/* set values of employee number, name,
   initials, and phone number */
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.

See also

- "Initialize the SQL Communications Area" on page 29

Data fetching

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.

Single row fetching

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.

See also

- "Initialize the SQL Communications Area" on page 29

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];
   a_sql_len   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.

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:

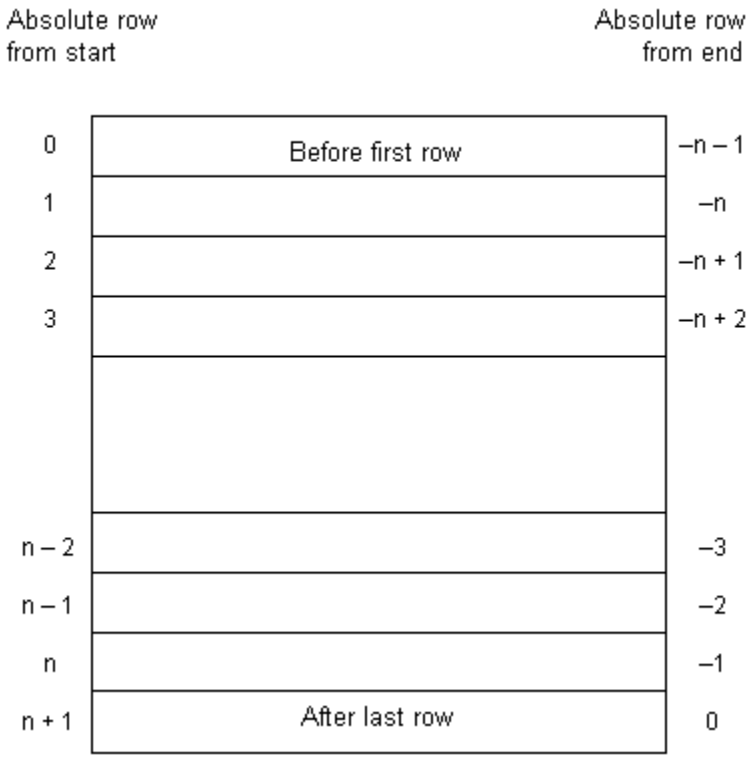
```
void print_employees( void )
{
   int status;
   EXEC SQL BEGIN DECLARE SECTION;
   char name[50];
   char sex;
   char birthdate[15];
   a_sql_len   ind_birthdate;
   EXEC SQL END DECLARE SECTION;
   /* 1. Declare the cursor. */
   EXEC SQL DECLARE C1 CURSOR FOR
      SELECT emp_fname || ' ' || 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;
}
```

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 can be difficult 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.

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.

See also

- "FETCH statement [ESQL] [SP]" [*SQL Anywhere Server - SQL Reference*]
- "Working with cursors" [*SQL Anywhere Server - Programming*]
- "Use work tables in query processing (use All-rows optimization goal)" [*SQL Anywhere Server - SQL Usage*]

User authentication

A complete sample can be found in the *%SQLANYSAMP12%\UltraLite\esqlauth* directory. The code below is taken from *%SQLANYSAMP12%\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 also

- “ULGrantConnectTo method [UltraLite Embedded SQL]” on page 262
- “ULRevokeConnectFrom method [UltraLite Embedded SQL]” on page 265

Encrypting data using UltraLite embedded SQL

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

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 easily be 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.

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

Change the encryption key on an UltraLite database

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.

- 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.

Obfuscation

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 also

- "ULChangeEncryptionKey method [UltraLite Embedded SQL]" on page 244
- "Specify UltraLite creation parameters" [*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.

Add synchronization to your application

1. Initialize the structure that holds the synchronization parameters.
2. Assign the parameter values for your application.
3. Call the synchronization function, supplying the structure or object as argument.

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

Synchronization parameters

The ul_sync_info structure is documented in the C/C++ component chapter; however, members of the structures are common to embedded SQL development as well.

See also

- "Initializing the synchronization parameters" on page 48
- "Network protocol options for UltraLite synchronization streams" [*UltraLite - Database Management and Reference*]
- "Invoking synchronization" on page 49
- "ul_sync_info structure [UltraLite C and Embedded SQL datatypes]" on page 110
- "Synchronization parameters for UltraLite" [*UltraLite - Database Management and Reference*]

Initializing the synchronization parameters

The synchronization parameters are stored in a structure.

Initialize the synchronization parameters (embedded SQL)

The members of the structure are 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 *%SQLANY12%\SDK\Include\ulglobal.h*.

- Call the ULInitSyncInfo function. For example:


```
ul_sync_info synch_info;
ULInitSyncInfo( &synch_info );
```

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_sync_info synch_info;
ULInitSyncInfo( &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 );
```

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. If the synchronization cannot be completed, add error handling code to your application.

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

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

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.

See also

- “Download Only synchronization parameter” [*UltraLite - Database Management and Reference*]

Add 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.

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.

See also

- "MobiLink development tips" [*MobiLink - Server Administration*]

Synchronization communications errors

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

```
if( psqlca->sqlcode == SQLE_MOBILINK_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_MOBILINK_COMMUNICATIONS_ERROR is the general error code for communications errors.

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

Synchronization monitoring and canceling

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:

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

Synchronization status information

The callback function that monitors synchronization takes a ul_sync_status structure as parameter.

The ul_sync_status structure has the following members:

```
struct ul_sync_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_sync_info  info;
    ul_sync_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_sync_info structure.
- **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_SYNC_STATE_STARTING** No synchronization actions have yet been taken.
  - **UL_SYNC_STATE_CONNECTING** The synchronization stream has been built, but not yet opened.
  - **UL_SYNC_STATE_SENDING_HEADER** The synchronization stream has been opened, and the header is about to be sent.
  - **UL_SYNC_STATE_SENDING_TABLE** A table is being sent.
  - **UL_SYNC_STATE_SENDING_DATA** Schema information or data is being sent.
  - **UL_SYNC_STATE_FINISHING_UPLOAD** The upload stage is completed and a commit is being carried out.
  - **UL_SYNC_STATE_RECEIVING_UPLOAD_ACK** An acknowledgement that the upload is complete is being received.
  - **UL_SYNC_STATE_RECEIVING_TABLE** A table is being received.
  - **UL_SYNC_STATE_RECEIVING_DATA** Schema information or data is being received.
  - **UL_SYNC_STATE_COMMITTING_DOWNLOAD** The download stage is completed and a commit is being carried out.
  - **UL_SYNC_STATE_SENDING_DOWNLOAD_ACK** An acknowledgement that download is complete is being sent.
  - **UL_SYNC_STATE_DISCONNECTING** The synchronization stream is about to be closed.

- **UL_SYNC_STATE_DONE** Synchronization has completed successfully.
- **UL_SYNC_STATE_ERROR** Synchronization has completed, but with an error.
- **UL_SYNC_STATE_ROLLING_BACK_DOWNLOAD** An error occurred during download and the download is being rolled back.

- **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 ULSetSynchronizationCallback 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_sync_status status )
{
    switch( status->state ) {
        case UL_SYNC_STATE_STARTING:
        printf( "Starting\n");
        break;
        case UL_SYNC_STATE_CONNECTING:
            printf( "Connecting\n"  );
            break;
        case UL_SYNC_STATE_SENDING_HEADER:
            printf( "Sending Header\n" );
            break;
        case UL_SYNC_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_SYNC_STATE_RECEIVING_TABLE:
            printf( "Receiving Table %d of %d\n",
                    status->tableIndex + 1,
                    status->tableCount );
            break;
        case UL_SYNC_STATE_SENDING_DOWNLOAD_ACK:
            printf( "Sending Download Ack\n" );
            break;
        case UL_SYNC_STATE_DISCONNECTING:
            printf( "Disconnecting\n" );
            break;
        case UL_SYNC_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 *%SQLANYSAMP12%\UltraLite\CustDB* directory. The observer function is contained in platform-specific subdirectories of the *CustDB* directory.

See also

- “ul_sync_info structure [UltraLite C and Embedded SQL datatypes]” on page 110
- “ul_sync_status structure [UltraLite C and Embedded SQL datatypes]” on page 113
- “The synchronization process” [*MobiLink - Getting Started*]

Embedded SQL application building

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

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

Understanding general build procedures

Sample code

You can find a makefile that uses this process in the *%SQLANYSAMP12%\UltraLite\ESQLSecurity* directory.

Note
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 12 - Introduction*].

Procedure

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.

   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.

See also

- "SQL preprocessor for UltraLite utility (sqlpp)" [*UltraLite - Database Management and Reference*]

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 (usually the object file) 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.

SQL preprocessing

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

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):

     ```
     "%SQLANY12%\Bin32\sqlpp.exe" -q -u $(InputPath) $(InputName).cpp
     ```

     where SQLANY12 is an environment variable that points to your SQL Anywhere installation directory.
   - 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 *$(SQLANY12)\SDK\Include* and any other include folders you require to your include path, as a comma-separated list.

See also

- "SQL preprocessor for UltraLite utility (sqlpp)" [*UltraLite - Database Management and Reference*]

UltraLite application development for Windows Mobile

Microsoft Visual Studio 2005 and later can be used to develop applications for the Windows Mobile environment.

Applications targeting Windows Mobile should use the default setting for wchar_t and link against the UltraLite runtime libraries in *\Program Files\SQLAny12\ultralite\ce\arm.50\lib*.

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

See also

- "Supported platforms" [*SQL Anywhere 12 - Introduction*]

Building the CustDB sample application

CustDB is a simple sales-status application. It is located in the *%SQLANYSAMP12%\UltraLite* directory of your SQL Anywhere installation.

The CustDB application is provided as a Visual Studio solution.

Note
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.

Build the CustDB sample application

1. Start Visual Studio.
2. Open the project file that corresponds to your version of Visual Studio.
   - *%SQLANYSAMP12%\UltraLite\CustDB\VS8* for Visual Studio 2005.
   - *%SQLANYSAMP12%\UltraLite\CustDB\VS9* for Visual Studio 2008 or later.
3. Click **Build** » **Set Configuration Manager** to set the target platform.
   - Set an active solution platform of your choice.
4. Build the application:
   - Click **Build** » **Deploy Solution** to build and deploy CustDB.

     When the application is built it will be uploaded automatically to the remote device.
5. Start the MobiLink server:
   - To start the MobiLink server, click **Start** » **Programs** » **SQL Anywhere 12** » **MobiLink** » **Synchronization Server Sample**.
6. 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 *%SQLANYSAMP12%\UltraLite\CustDB\custdb.udb* to the root of the device.

   On the device or simulator, execute *CustDB.exe*, which is located in the project folder under *\Program Files*.

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 also

- "The CustDB sample database application" [*SQL Anywhere 12 - Introduction*]
- "Embedded SQL application building" on page 54

Persistent data

The UltraLite database is stored in the Windows Mobile file system. The default file is *\UltraLiteDB \ul_store.udb*. 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"
```

Assigning class names for applications

Assign a window class name for MFC 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.

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 **MyAppDlg**.

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.

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. Add code to catch the synchronization message.

See also

- "Adding ActiveSync synchronization (MFC)" on page 61

Windows Mobile synchronization

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

- ActiveSync
- TCP/IP
- HTTP

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.

See also

- "Add ActiveSync synchronization to your application" on page 60
- "TCP/IP, HTTP, or HTTPS synchronization from Windows Mobile" on page 63
- "TCP/IP, HTTP, or HTTPS synchronization from Windows Mobile" on page 63
- "Synchronization parameters for UltraLite" [*UltraLite - Database Management and Reference*]

Add 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 **Synchronize** 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.

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.

See also

- "Deploying the ActiveSync provider for UltraLite" [*UltraLite - Database Management and Reference*]

Add 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 method that actually calls ULSynchronize.

See also

- "ULIsSynchronizeMessage method [UltraLite Embedded SQL]" on page 262

Adding ActiveSync synchronization (MFC)

Add ActiveSync synchronization in the main dialog class

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.

Your application must create and register a custom window class name for notification.

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.

Add ActiveSync synchronization in the Application class

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.

Your application must create and register a custom window class name for notification.

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.

See also

- “Assigning class names for applications” on page 58
- “ULIsSynchronizeMessage method [UltraLite Embedded SQL]” on page 262
- “Assigning class names for applications” on page 58
- “ULIsSynchronizeMessage method [UltraLite Embedded SQL]” on page 262

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.

Tutorials

This section provides tutorials for the UltraLite C/C++ API.

Tutorial: Building a Windows application using the C++ API

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**.

The goal for the tutorial is to gain competence with the process of developing an UltraLite C++ application.

See also

- "Creating an UltraLite database with the Create Database Wizard" [*UltraLite - Database Management and Reference*]

Lesson 1: Creating and connecting to a 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.

Create an UltraLite database

1. Set the **VCINSTALLDIR** environment variable to the root directory of your Visual C++ installation if the variable does not already exist.
2. Add *%VCINSTALLDIR%\VC\atlmfc\src\atl* to your **INCLUDE** environment variable.
3. 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*.

4. Using UltraLite in Sybase Central, create a database named *ULCustomer.udb* in your new directory with the default characteristics.

5. 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	

6. Disconnect from the database in Sybase Central, otherwise your executable will not be able to connect.

Connect to the UltraLite database

1. In Microsoft Visual C++, click **File** » **New**.

2. On the **Files** tab, click **C++ Source File**.

3. Save the file as *customer.cpp* in your tutorial directory.

4. Include the UltraLite libraries.

   Copy the code below into *customer.cpp*:

```
#include <tchar.h>
#include <stdio.h>
#include "ulcpp.h"
#define MAX_NAME_LEN 100
```

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 =
    "UID=DBA;PWD=sql;DBF=C:\\tutorial\\cpp\\ULCustomer.udb";
```

Note
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 method 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 method or can choose to use an error callback function.

The following code is a sample callback function:

```
ul_error_action UL_CALLBACK_FN MyErrorCallBack(
    const ULError *  error,
    ul_void *        user_data )
{
    ul_error_action rc;
    an_sql_code code = error->GetSQLCode();

    (void) user_data;

    switch( code ){
        // The following error is used for flow control - don't report it
here
        case SQLE_NOTFOUND:
            rc = UL_ERROR_ACTION_CONTINUE;
        break;

        default:
            if (code >= 0) { // warning or success
                rc = UL_ERROR_ACTION_DEFAULT;
            } else { // negative is real error
                ul_char etext[ MAX_NAME_LEN ];
                error->GetString( etext, MAX_NAME_LEN );
                _tprintf( "Error %ld: %s\n", code, etext );
                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.

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.

   ```
   static ULConnection * open_conn( void )
   {
       ULConnection * conn =
   ULDatabaseManager::OpenConnection( ConnectionParms );
       if( conn == UL_NULL ) {
         _tprintf("Unable to open existing database.\n");
      }
     return conn;
   }
   ```

8. Implement the main method to perform the following tasks:

   - Registers the error handling method.

- Opens a connection to the database.
- Closes the connection and finalizes the database manager.

```
int main() {
    ULConnection *    conn;

    ULDatabaseManager::Init();
    ULDatabaseManager::SetErrorCallback( MyErrorCallBack, NULL );

    conn = open_conn();
    if ( conn == UL_NULL ) {
        ULDatabaseManager::Fini();
        return 1;
    }

    // Main processing code goes here ...
    do_insert( conn );
    do_select( conn );
    do_sync( conn );

    conn->Close();
    ULDatabaseManager::Fini();
    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"$(SQLANY12)\SDK\Include"
     ```

   - In the makefile, add directories to your libraries path.

     ```
     LibraryFolders=/LIBPATH:"$(SQLANY12)\UltraLite\Windows\x86\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 *vsvars32.bat*.

  ```
  %VCINSTALLDIR%\Tools\vsvars32.bat
  ```

- Run the makefile.

  ```
  nmake
  ```

  This creates an executable named *customer.exe*.

10. Run the application.

    At a command prompt, enter **customer**.

    The application connects to the database and then disconnects. The application runs successfully if you do not see any error messages.

See also

- "Creating an UltraLite database with the Create Database Wizard" [*UltraLite - Database Management and Reference*]
- "UltraLite connection parameters" [*UltraLite - Database Management and Reference*]
- "Error handling" on page 22

Lesson 2: Inserting data into the database

The following procedure demonstrates how to add data to a database.

Add rows to your database

1. Add the method below to *customer.cpp* immediately before the main method:

```
static bool do_insert( ULConnection * conn )
{
    ULTable * table = conn->OpenTable( "ULCustomer" );
    if( table == UL_NULL ) {
        _tprintf( "Table not found: ULCustomer\n" );
        return false;
    }
    if( table->GetRowCount() == 0 ) {
        _tprintf( "Inserting one row.\n" );
        table->InsertBegin();
        table->SetString( "cust_name", "New Customer" );
        table->Insert();
        conn->Commit();
    } else {
        _tprintf( "The table has %lu rows\n", table->GetRowCount() );
    }
    table->Close();
    return true;
}
```

This method performs the following tasks.

- Opens the table using the connection->OpenTable() method. You must open a Table object to perform 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.
- 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 before the call to conn->Close.

   ```
   do_insert(conn);
   ```

3. Compile your application by running nmake.

4. Run your application by typing **customer** at a command prompt.

Lesson 3: Selecting and listing rows from the table

The following procedure retrieves rows from the table and prints them on the command line.

List rows in the table

1. Add the method below to *customer.cpp* immediately after the do_insert method. 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.

   ```
   static bool do_select( ULConnection * conn )
   {
       ULTable * table = conn->OpenTable( "ULCustomer" );
       if( table == UL_NULL ) {
           return false;
       }
       ULTableSchema * schema = table->GetTableSchema();
       if( schema == UL_NULL ) {
           table->Close();
           return false;
   ```



```
    }
    ul_column_num id_cid =
        schema->GetColumnID( "cust_id" );
    ul_column_num cname_cid =
        schema->GetColumnID( "cust_name" );
    schema->Close();

    _tprintf( "\n\nTable 'ULCustomer' row contents:\n" );
    while( table->Next() ) {
        ul_char cname[ MAX_NAME_LEN ];
        table->GetString( cname_cid, cname, MAX_NAME_LEN );
        _tprintf( "id=%d, name=%s \n", (int)table->GetInt(id_cid),
cname );
    }
    table->Close();
    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 a command prompt.

Lesson 4: Adding 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 12 Sample database. The UltraLite 12 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:

   - Enables TCP/IP communications by invoking EnableTcpipSynchronization. Synchronization can also be carried out over HTTP, HTTPS, and TLS.
   - 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.

```
static bool do_sync( ULConnection * conn )
{
    ul_sync_info info;
    ul_stream_error * se = &info.stream_error;

    ULDatabaseManager::EnableTcpipSynchronization();
    conn->InitSyncInfo( &info );
    info.stream = "TCPIP";
    info.version = "custdb 12.0";
    info.user_name = "50";
    info.download_only = true;
    if( !conn->Synchronize( &info ) ) {
        _tprintf( "Synchronization error \n" );
        _tprintf( "  stream_error_code is '%lu'\n", se-
>stream_error_code );
        _tprintf( "  system_error_code is '%ld'\n", se-
>system_error_code );
        _tprintf( "  error_string is '" );
        _tprintf( "%s", se->error_string );
        _tprintf( "'\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.

   At a command prompt, run the following command:

```
mlsrv12 -c "dsn=SQL Anywhere 12 CustDB;uid=ml_server;pwd=sql" -v -vr -vs -
zu+ -o custdbASA.log
```

   The -zu+ option provides automatic addition of users. The -v+ option turns on verbose logging for all messages.

2. Run your application by typing *customer* at a command prompt.

   The MobiLink server messages window displays status messages indicating the synchronization progress. If synchronization is successful, the final message displays `Synchronization complete`.

See also

- “UltraLite clients” [*UltraLite - Database Management and Reference*]
- “MobiLink server options” [*MobiLink - Server Administration*]

Code listing for tutorial

The following is the complete code for the tutorial program described in the preceding sections.

```
#include <tchar.h>
#include <stdio.h>

#include "ulcpp.h"

#define MAX_NAME_LEN 100

static ul_char const * ConnectionParms =
    "UID=DBA;PWD=sql;DBF=c:\\tutorial\\cpp\\ULCustomer.udb";

ul_error_action UL_CALLBACK_FN MyErrorCallBack(
    const ULError *  error,
    ul_void *        user_data )
{
    ul_error_action rc;
    an_sql_code code = error->GetSQLCode();

    (void) user_data;

    switch( code ){
        // The following error is used for flow control - don't report it
here
        case SQLE_NOTFOUND:
            rc = UL_ERROR_ACTION_CONTINUE;
        break;

        default:
            if (code >= 0) { // warning or success
                rc = UL_ERROR_ACTION_DEFAULT;
            } else { // negative is real error
                _tprintf( "Error %ld: %s\n", code, error->GetString() );
                rc = UL_ERROR_ACTION_CANCEL;
            }
        break;
     }
    return rc;
}

static ULConnection * open_conn( void ) {
    ULConnection * conn =
ULDatabaseManager::OpenConnection( ConnectionParms );
    if( conn == UL_NULL ) {
        _tprintf( "Unable to open existing database.\n" );
    }
    return conn;
}

static bool do_insert( ULConnection * conn ) {
    ULTable * table = conn->OpenTable( "ULCustomer" );
    if( table == UL_NULL ) {
        _tprintf( "Table not found: ULCustomer\n" );
        return false;
    }
    if( table->GetRowCount() == 0 ) {
        _tprintf( "Inserting one row.\n" );
        table->InsertBegin();
        table->SetString( "cust_name", "New Customer" );
        table->Insert();
        conn->Commit();
```

```
    } else {
        _tprintf( "The table has %lu rows\n",
        table->GetRowCount() );
    }
    table->Close();
    return true;
}

static bool do_select( ULConnection * conn )
{
    ULTable * table = conn->OpenTable( "ULCustomer" );
    if( table == UL_NULL ) {
        return false;
    }
    ULTableSchema * schema = table->GetTableSchema();
    if( schema == UL_NULL ) {
        table->Close();
        return false;
    }
    ul_column_num id_cid =
        schema->GetColumnID( "cust_id" );
    ul_column_num cname_cid =
        schema->GetColumnID( "cust_name" );

    schema->Close();

    _tprintf( "\n\nTable 'ULCustomer' row contents:\n" );

    while( table->Next() ) {
        ul_char cname[ MAX_NAME_LEN ];

        table->GetString( cname_cid, cname, MAX_NAME_LEN );

        _tprintf( "id=%d, name=%s \n", (int)table->GetInt(id_cid), cname );
    }
    table->Close();
    return true;
}

static bool do_sync( ULConnection * conn )
{
    ul_sync_info info;
    ul_stream_error * se = &info.stream_error;

    ULDatabaseManager::EnableTcpipSynchronization();
    conn->InitSyncInfo( &info );
    info.stream = "TCPIP";
    info.version = "custdb 12.0";
    info.user_name = "50";
    info.download_only = true;
    if( !conn->Synchronize( &info ) ) {
        _tprintf( "Synchronization error \n" );
        _tprintf( "  stream_error_code is '%lu'\n", se->stream_error_code );
        _tprintf( "  system_error_code is '%ld'\n", se->system_error_code );
        _tprintf( "  error_string is '" );
        _tprintf( "%s", se->error_string );
        _tprintf( "'\n" );
        return false;
    }
    return true;
}

int main()
{
```



```
    ULConnection *    conn;

    ULDatabaseManager::Init();
    ULDatabaseManager::SetErrorCallback( MyErrorCallBack, NULL );

    conn = open_conn();
    if( conn == UL_NULL ){
        ULDatabaseManager::Fini();
        return 1;
    }

    // Main processing code goes here ...
    do_insert( conn );
    do_select( conn );
    do_sync( conn );

    conn->Close();
    ULDatabaseManager::Fini();
    return 0;
}
```

Tutorial: Building an iPhone application using the C ++ API

This tutorial guides you through the process of building an UltraLite C++ application for the iPhone using the Apple development tools. Wherever possible, this tutorial provides links to more detailed information.

Required software

- Xcode 3.2

> **Note**
> The **Names** sample used in this tutorial is compatible with Xcode 4.2 but there are several graphical user interface differences between Xcode 3.2 and Xcode 4.2. For a tutorial about developing iPhone applications with the latest version of Xcode, see http://developer.apple.com/library/ios/#documentation/iPhone/Conceptual/iPhone101/Articles/00_Introduction.html.

Competencies and experience

This tutorial assumes:

- You are familiar with Objective-C and C++ programming languages.
- You have a C++ compiler installed on your computer.
- You have the iPhone SDK installed on your computer.
- You have enrolled in the iPhone Developer Program (this is only required to run the program on a physical device). Running on the iPhone Simulator does not require enrollment.

The goal for the tutorial is to gain competence with the process of developing an UltraLite C++ iPhone application.

Compiling the UltraLite iPhone library

To use UltraLite on the iPhone, you must first compile the library.

Compile the library for the iPhone simulator

1. In a Terminal window, navigate to the SQL Anywhere *ultralite/iphone* directory.
2. If your environment is not set up for SQL Anywhere, source the *sa_config* shell script for your Terminal's shell from the SQL Anywhere *System/bin64* directory.
3. Run the following command to invoke the interactive build script:

   ```
   ./build.sh
   ```
4. Select the options you prefer.
5. Once compilation is complete, verify that the *libulrt.a* universal archive has been created in the current directory.

Creating an UltraLite application on iPhone

The first part of this tutorial walks you through the process of creating an iPhone application that maintains a list of names in a simple, single-table UltraLite database. The second part of the tutorial adds synchronization with a SQL Anywhere database to the application.

Tip
All functions should be declared in the header file or declared before use.

See also

- "Lesson 6: Adding synchronization" on page 88

Lesson 1: Creating a new iPhone application project

Before you can start doing anything, you must first create an Xcode project.

Create an Xcode project

1. Launch Xcode.
2. Under the **File** menu, click **New Project**.
3. Click **iPhone OS Application** from the left side selection.
4. Click a **Navigation-based Application**, leaving the **Use Core Data for Storage** option cleared.
5. Click the **Choose** button and save the project **names** in the location of your choice.

The Navigation-based project automatically creates an application with a *UITableViewController* inside of a *UINavigationController*, as well as a *UIApplicationDelegate* that places the controllers in the window when the application is launched. So far, the application only has an empty navigation bar at the top and an empty table view. To fill the table with names, the application requests the data from an UltraLite database on the device.

Setting up the project for an UltraLite library

Set up the project for the UltraLite library

1. To use the UltraLite library, you must configure the Xcode project.
2. Control-click the *names* project on the left side of the project window and click **Get Info**.
3. Under the **Build** tab of the Info window, search for the **User Header Search Paths** setting using the search field and double-click the setting.
4. In the **Finder**, navigate to the SQL Anywhere *sdk* folder.
5. Click and drag the *include* folder into the modal window that opened when you double-clicked the setting in the **info** window.
6. Click **OK**.

Compiling the project as C++

This tutorial uses the UltraLite C++ API. To eliminate the need to cast to C types, compile the source as C ++.

Compile the project as C++

1. In the **Search in Build Settings** box, enter **Compile Sources As**.
2. Click **Objective-C++** from the options in the **Value** field.

   Note
   On some older versions of Xcode, this setting is only visible if a device target is selected for this project.

3. Close the **project info** window.

Adding the UltraLite library to the project

The only step left to be able to use UltraLite in the application is to add the library itself to the project.

Add the UltraLite library to the project

1. In the **Finder**, navigate to the *libulrt.a* library you compiled earlier. It should be located in *ultralite/iphone* in your SQL Anywhere folder.
2. Click and drag the *libulrt.a* library from the **Finder** into the **names** project in the project window (where you control-clicked to get the project info window).

3. You should now have *libulrt.a* listed in the project files with a yellow toolbox icon.

Adding frameworks required by UltraLite

UltraLite requires the **CFNetwork** and **Security** frameworks in addition to the default frameworks.

Add the frameworks required by UltraLite

1. Control-click the **Frameworks** folder under the *names* project in the project window and click » **Add** » **Existing Frameworks**.

2. Browse to the **Security** framework and click **Add**. From the list, click */Developer/Platforms/iPhoneOS.platform/Developer/SDKs/iPhoneOS3.1.3.sdk/System/Library/Frameworks/Security.framework*.

3. Repeat these steps for the **CFNetwork** framework.

Lesson 2: Adding a database to the application

Tto interact with the database, the application follows the MVC pattern prescribed by the Apple framework and defines a Model class.

Data Model Class

The Data Access class is the only class that interacts with the database. This way, if a schema change or other database-related change is required, there is only one place where updates are required.

Set up the Data Model Class

1. Control-click the **Classes** folder in the **Project** window. Click **Add** » **New File**.

2. Click **iPhone OS Cocoa Touch Class** in the left selection.

3. Click a new Objective-C class.

4. Make sure the **Subclass** of selection is **NSObject**.

5. Click **Next**.

6. Name the file *DataAccess.mm*. The *.mm* extension is important as it signals to Xcode that the file contains both Objective-C and C++.

7. Make sure **Also create DataAccess.h** is selected.

8. For **Location**, click the *names/Classes* subfolder.

9. Click **Finish**.

The DataAccess singleton

When the application launches, it needs to initialize the UltraLite database manager and connect to the local database. To do this, the class creates a singleton instance of this data access object which is used by the **RootViewController** to display and manage the list of names.

In the *DataAccess.mm* file, add the following code in the implementation:

```
static DataAccess *        sharedInstance = nil;

+ (DataAccess *)sharedInstance {
    // Create a new instance if none was created yet
    if (sharedInstance == nil) {
        sharedInstance = [[super alloc] init];
        [sharedInstance openConnection];
    }

    // Otherwise, just return the existing instance
    return sharedInstance;
}
```

Before you can use any UltraLite classes and methods, you must import the *ulcpp* header file. Add the following line to the existing imports in *DataAccess.h*:

```
#import "ulcpp.h"
```

Open the connection to the database

Before the connection can be opened, the database manager's Init method is used to initialize the UltraLite runtime. Once it is initialized, an attempt to connect to the database indicates whether the database exists. Add the following instance variable to the *DataAccess.h* header inside the interface block:

```
ULConnection *     connection;
```

In the *DataAccess.mm* file, add the following code in the implementation:

```
- (void)openConnection {
    NSLog(@"Connect to database.");
    if (ULDatabaseManager::Init()) {
        NSArray *    paths = NSSearchPathForDirectoriesInDomains(

NSDocumentDirectory,

NSUserDomainMask,
                                                                YES);
        NSString *    documentsDirectory = [paths objectAtIndex:0];
        NSString *    writableDBPath = [documentsDirectory
                                        stringByAppendingPathComponent:
                                        @"Names.udb"];
        ULConnection *    conn = nil;
        const char *    connectionParms;
        ULError           error;

        connectionParms = [[NSString stringWithFormat:@"DBF=%@",
                            writableDBPath]
                            UTF8String];

        // Attempt connection to the database
        conn = ULDatabaseManager::OpenConnection(
                                                connectionParms,
                                                &error);
```

```
        // If database file not found, create it and create the schema
        if (error.GetSQLCode() == SQLE_ULTRALITE_DATABASE_NOT_FOUND) {
            conn = [self createDatabase:connectionParms];
        }
        connection = conn;
    } else {
        NSLog(@"UL Database Manager initialization failed.");
        connection = nil;
    }
}
```

Database access

The database schema of the application is composed of a single table with two columns. The Names table has an ID column that uses UUIDs and a name column that stores the names as VARCHARs. The ID column uses UUIDs to easily support row insertions from remote databases using MobiLink.

> **Note**
> In the following code samples, *openConnection* uses *createDatabase*, so either *createDatabase* must come before *openConnection* or its method signature must be added to the header file.

Displaying data in a table view requires that each row be accessible using a 1-based index. To do this, the database uses an ascending index on the name column. Each row's index is equal to its position in the alphabetical list of names.

In the *DataAccess.mm* file, add the following code in the implementation:

```
- (ULConnection *)createDatabase:(const char *)connectionParms {
    const char *    CREATE_TABLE =
    "CREATE TABLE Names ("
    "id UNIQUEIDENTIFIER DEFAULT NEWID() PRIMARY KEY,"
    "name VARCHAR(254) NOT NULL)";
    const char *    CREATE_INDEX =
    "CREATE UNIQUE INDEX namesIndex ON Names(name ASC)";
    const char *    createParms =
    "page_size=4k;utf8_encoding=true;collation=UTF8BIN";
    ULError             error;
    ULConnection *    conn;

    conn = ULDatabaseManager::CreateDatabase(
                                            connectionParms,
                                            createParms,
                                            &error);
    if (!conn) {
        NSLog(@"Error code creating the database: %ld",
              error.GetSQLCode());
    } else {
        NSLog(@"Creating Schema.");
        conn->ExecuteStatement(CREATE_TABLE);
        conn->ExecuteStatement(CREATE_INDEX);
    }
    return conn;
}
```

Add a dealloc method to finalize the UltraLite runtime:

```
- (void)dealloc {
    NSLog(@"Finalizing DB Manager.");
    connection->Close();
```



```
    ULDatabaseManager::Fini();

    [super dealloc];
}
```

And a fini method to release the instance:

```
+ (void)fini {
    [sharedInstance release];
}
```

Add the method signatures to the header file after the interface curly brace block:

```
// Release objects.
- (void)dealloc;

// Singleton instance of the DataAccess class.
+ (DataAccess*)sharedInstance;

// Finalize the Database Manager when done with the DB.
+ (void)fini;
```

Call the fini method from the **NamesAppDelegate**'s **applicationWillTerminate** method:

```
- (void)applicationWillTerminate:(UIApplication *)application {
    // Save data if appropriate
    [DataAccess fini];
}
```

Also, since the application delegate is calling fini, it must import the **DataAccess** header file. Add the following to *namesAppDelegate.h*:

```
#import "DataAccess.h"
```

Build the application

At this point you should build the application to test that it builds without errors. From the **Build** menu, click **Build**.

Lesson 3: Adding data to the database

Now that the application has a database on the device with its schema initialized, it can start adding names to the database. To do this, the application uses a new screen with a text field.

View controller for new names

To let the user input new names you create a new view controller.

Set up the view controller for new names

1. Ctrl-click *Classes* and from the **Add File** menu, click **New File**.
2. With the **iPhone OS Cocoa Touch Class** selected on the left selection, click **UIViewController** subclass.
3. Do not click **UITableViewController** subclass option and click **With XIB for user interface**.

4. Click **Next**.

5. Name the file *NewNameViewController.m*.

6. Click the *names/Classes/* location.

7. Click **Finish**.

This creates three files: the **NewNameViewController** (*.h*) header and implementation (*.m*) file, as well as a XIB file that can be edited in Interface Builder. To group all the XIB files together, you can move it to the *Resources* folder where two XIB files already exist (the **MainWindow** and **RootViewController**'s XIB files).

Before editing the XIB file to contain the text field, you must first add an **Outlet** property to the header so that you can tie in the text field from the XIB file into the code as well as an action to perform when the user has finished entering the name. Add the following to the *NewNameViewController.h* file

```
@interface NewNameViewController : UIViewController {
UITextField *newNameField;
}
@property (retain) IBOutlet UITextField *newNameField;
- (IBAction)doneAdding:(id)sender;
@end
```

Synthesize the property in the implementation file and release the text field in the dealloc method. You define the action later, once the **DataAccess** object can insert new names. For now, simply create an empty method stub in *NewNameViewController.m* to avoid compilation warnings.

```
@synthesize newNameField;
- (IBAction)doneAdding:(id)sender {}
```

The **IBAction** keyword lets **Interface Builder** know that the method should be made available for events to call. Make sure to save the header so that **Interface Builder** knows about the method.

Now that the outlet is set up, double-click the **NewNameViewController** XIB file to edit it in **Interface Builder**. In **Interface Builder**, you should see an empty view with a status bar showing the battery charging. Since this view also has a navigation bar showing, simulate showing it by doing the following:

1. In the **Document Window** (Command-0), click **View**.

2. In the **Attributes Inspector Window** (Command-1), under the **Simulated User Interface Elements**, set the **Top Bar** to **Navigation Bar**.

The view should now show an empty navigation bar just below the status bar.

To allow the user to input a new name, add a text field to the view:

1. In the **Library** window (Command-Shift-L), under **Inputs & Values**, click and drag a text field into the view.

2. Position the text field in the upper half of the view, so that the keyboard in the lower half won't hide it.

3. Make the text field a little wider so that a name would fit within it. Use a width of about 230 pixels. To see the size and placement options, use the **Size Inspector** (Command-3) while the text field is selected.

To make the text field more user-friendly, you can change a few properties:

1. Click the text field in the view.
2. With the text field selected, open the **Attributes Inspector Window** (Command-1).
3. Set the **Placeholder** to **Name**.
4. Make the font size 18 points.
5. Set **Capitalize** to **Words**.
6. Set the **Return Key** to **Done**.

For the controller to be made aware of when the user has finished entering the name, you must make action connections in **Interface Builder**.

1. With the text field selected, open the **Text Field Connections Inspector** (Command-2).
2. Click and drag from the **Did End on Exit** event circle to the **File's Owner** in the **Document Window**, and click **done Adding** to make the connection. If dragging and dropping over the **File's Owner** doesn't provide a selection, make sure the *NewNameViewController* header is saved and has the **IBOutlet doneAdding**. The **File's Owner** of this XIB file is the *NewNameViewController* class, as the **File's Owner's** type indicates.

In order to refer to the text field in the code, you need to connect this text field to the **IBOutlet** property you defined in the header earlier.

1. Click **File's Owner** in the **Document Window**. Open the **Connections Inspector** (Command-2).
2. Under **Outlets**, look for **newNameField**.
3. Click and drag from the **newNameField** circle to the actual name field in the view.
4. The view is now complete. Save the XIB file in **Interface Builder** and return to Xcode.

To finish the view, you need to set a few more properties in code. Uncomment the **viewDidLoad** method template in the **NewNameViewController** implementation and replace it with the following code:

```
- (void)viewDidLoad {
    [super viewDidLoad];

    // Set the title to display in the nav bar
    self.title = @"Add Name";

    // Set the text field to the first responder to display the keyboard.
    // Without this the user needs to tap on the text field.
    [newNameField becomeFirstResponder];
}
```

Root view controller setup

To display the view controller you just created, the **RootViewController** needs to be configured. Import the **NewNameViewController** and **DataAccess** headers in the **RootViewController**'s implementation file and add the following method signature to its header (*RootViewController.h*):

```
- (void)showAddNameScreen;
```

Implement the method in *RootViewController.m* with the following code:

```
- (void)showAddNameScreen {
    NewNameViewController * addNameScreen = [[NewNameViewController alloc]

initWithNibName:@"NewNameViewController" bundle:nil];
    [self.navigationController pushViewController:addNameScreen
animated:YES];
}
```

Set the title of the **RootViewController** (the same way you set the title for the **NewNameViewController**), as well as add a plus-sign button to the right side of the navigation bar that calls **showAddNameScreen**. Uncomment the **viewDidLoad** block and replace it with the following code:

```
- (void)viewDidLoad {
    [super viewDidLoad];
    // The Navigation Controller uses this to display the title in the nav
bar.
    self.title = @"Names";
    // Little button with the + sign on the right in the nav bar
    self.navigationItem.rightBarButtonItem =
    [[UIBarButtonItem alloc]
     initWithBarButtonSystemItem:UIBarButtonSystemItemAdd
     target:self
     action:@selector(showAddNameScreen)];
}
```

Inserting new names into the database

To insert the new names into the database, you add functionality to the **DataAccess** object. Add the following method signature to the **DataAccess** header:

```
// Adds the given name to the database.
- (void)addName:(NSString *)name;
```

Implement the **addName** method in the implementation file:

```
- (void)addName:(NSString *)name {
    const char *           INSERT = "INSERT INTO Names(name) VALUES(?)";
    ULPreparedStatement *    prepStmt = connection->PrepareStatement(INSERT);

    if (prepStmt) {
        // Convert the NSString to a C-Style string using UTF8 Collation
        prepStmt->SetParameterString(1, [name UTF8String], [name length]);
        prepStmt->ExecuteStatement();
        prepStmt->Close();
        connection->Commit();
    } else {
        NSLog(@"Could not prepare INSERT statement.");
    }
}
```


Now that the **addName** method is complete, add the **NewNameViewController**'s **doneAdding** method to the implementation file. Replace the method with the following code:

```
- (IBAction)doneAdding:(id)sender {
    if (newNameField.text > 0) {
        [[DataAccess sharedInstance] addName:newNameField.text];
    }
    [self.navigationController popViewControllerAnimated:YES];
}
```

Import the **DataAccess** header in *NewNameViewController.h*. This allows the keyboard's **Done** button to add the name to the database and return to the table view at runtime. However, the table view is not yet configured to display what is in the database. See "Lesson 4: Displaying data from the database" on page 85.

Build the application

At this point you should build the application to test that it builds without errors. From the **Build** menu, click **Build**.

Lesson 4: Displaying data from the database

By default, the data source of a table view is its controller. In the case of this application, the **RootViewController** is currently the data source for the table view. In this lesson, you implement the **UITableViewDataSource** protocol in the **DataAccess** class. This way, the **DataAccess** class is responsible for providing the table view with the data it requires.

To begin, add the **UITableViewDataSource** protocol to the interface.

```
@interface DataAccess : NSObject <UITableViewDataSource> {
    ULConnection *                  connection;
}
```

Getting the number of names in the database

The **UITableViewDataSource** protocol has two required methods. This section implements one of those methods, namely, the **tableView:numberOfRowsInSection:** method. The table view has a single section: a list of names. Therefore, this method simply returns the count of the rows in the database. Use the following SQL statement:

```
SELECT COUNT (*) FROM Names;
```

Add the following method to the *DataAccess.mm* implementation file to get the count:

```
- (NSInteger)tableView:(UITableView *)tableView
 numberOfRowsInSection:(NSInteger)section {
    const char *            COUNT = "SELECT COUNT (*) FROM Names";
    ULPreparedStatement *    prepStmt = connection->PrepareStatement(COUNT);

    if (prepStmt) {
        ULResultSet    *resultSet = prepStmt->ExecuteQuery();
        int            numberOfNames;

        resultSet->First();
```

```
        numberOfNames = resultSet->GetInt(1);
        resultSet->Close();
        prepStmt->Close();
        return numberOfNames;
    } else {
        NSLog(@"Couldn't prepare COUNT.");
    }

    return 0;
}
```

Creating the table cells

The second required method in the protocol is the **tableView:cellForRowAtIndexPath:** method. This method is responsible for creating the actual cell objects that the table view displays. The **RootViewController** defines a good template to reuse cells. Table cells are usually recycled using the pattern in the template to save memory and allow for fast scrolling. The only identifier the table view provides is a 0-based integer index. However, the database does not normally keep an ordered list of the names. To simulate an ordered list, the database uses an ordered index on the names and the application uses a SELECT statement with a row limitation and an ORDER BY clause to select a given index. Although this method doesn't require the FOR UPDATE clause, this same query is used to delete names in "Lesson 5: Deleting data from the database" on page 87.

Add the following code to *DataAccess.mm* implementation file just below the import statements:

```
#define SELECT_STMT @"SELECT TOP 1 START AT %d name FROM Names ORDER BY name
FOR UPDATE"
```

To use this query in the method, add the following to the *DataAccess.mm* implementation file.

```
- (UITableViewCell *)tableView:(UITableView *)tableView
         cellForRowAtIndexPath:(NSIndexPath *)indexPath {
    static NSString *        CellIdentifier = @"Cell";
    UITableViewCell *        cell =
     [tableView dequeueReusableCellWithIdentifier: CellIdentifier];
    ULPreparedStatement *    prepStmt;

    if (cell == nil) {
        cell = [[[UITableViewCell alloc]
                 initWithStyle:UITableViewCellStyleDefault
                 reuseIdentifier:CellIdentifier]
                 autorelease];
    }

    // Make it so the cell cannot be selected.
    cell.selectionStyle = UITableViewCellSelectionStyleNone;

    // +1 to the index since the DB uses a 1-based index rather than 0-based
    prepStmt = connection->PrepareStatement(
        [[NSString stringWithFormat:SELECT_STMT, indexPath.row + 1]
UTF8String]);
    if (prepStmt) {
        ULResultSet *    resultSet = prepStmt->ExecuteQuery();
        char             name[255];

        resultSet->First();
        resultSet->GetString("name", name, 255);
        resultSet->Close();
        prepStmt->Close();
```



```
            cell.textLabel.text = [NSString stringWithUTF8String:name];
        } else {
            NSLog(@"Couldn't prepare SELECT with index.");
        }

        return cell;
    }
```

Setting the DataAccess object as the data source

Now that the **DataAccess** object is configured as a data source, you need to set the data source of the table. To do this, add the following at the end of the **viewDidLoad** method of the **RootViewController** class (in *RootViewController.m*):

```
    // Set the data source.
    [self.tableView setDataSource:[DataAccess sharedInstance]];
```

You also need to uncomment the **viewWillAppear** method and add the following statement (otherwise changes are not picked up until the rows scroll out of view and back in again):

```
    [self.tableView reloadData];
```

Build and run the application

At this point you should build the application to test that it builds without errors. From the **Build** menu, click **Build and run**. The application is now able to display names from the database, and the user is able to add new names using a custom view.

Lesson 5: Deleting data from the database

The application is now able to display names from the database, and the user is able to add new names using a custom view. It is not yet possible for the user to remove a name. This lesson adds the deletion functionality using the swipe-to-delete functionality available in many iPhone applications.

Deleting the name from the database

To delete the name from the database, use the same SQL statement as the select, but open the result set for update so that you can delete the selected row. Add the following method to the **DataAccess** class:

```
- (void)removeNameAtIndexPath:(NSIndexPath *)indexPath {
    // +1 to the index since the DB uses a 1-based index rather than 0-based
    ULPreparedStatement *prepStmt =
     connection->PrepareStatement(
      [[NSString stringWithFormat:SELECT_STMT, indexPath.row + 1]
UTF8String]);

    if (prepStmt) {
        ULResultSet        *resultSet = prepStmt->ExecuteQuery();

        resultSet->First();
        resultSet->Delete();
        resultSet->Close();
        prepStmt->Close();
        connection->Commit(); // Commit the deletion.
    } else {
        NSLog(@"Couldn't prepare SELECT with index for delete.");
```

```
    }
}
```

Enabling swipe-to-delete

To enable the delete functionality, copy the comment template of the **tableView:commitEditingStyle:forRowAtIndexPath** method from **RootViewController** into the **DataAccess** class and call the **removeNameAtIndexPath** method you just created. Place this code after the **removeNameAtIndexPath** method.

```
- (void)tableView:(UITableView *)tableView commitEditingStyle:
(UITableViewCellEditingStyle)editingStyle forRowAtIndexPath:(NSIndexPath
*)indexPath {

    if (editingStyle == UITableViewCellEditingStyleDelete) {
        // Delete the row from the data source.
        [[DataAccess sharedInstance] removeNameAtIndexPath:indexPath];
        [tableView deleteRowsAtIndexPaths:[NSArray
arrayWithObject:indexPath]
                         withRowAnimation:UITableViewRowAnimationFade];
    } else if (editingStyle == UITableViewCellEditingStyleInsert) {
        // Create a new instance of the appropriate class, insert it into
the array, and add a new row to the table view.
    }
}
```

The application is now complete. The table view displays the list of names in alphabetical order. New names can be added using the **NewNameViewController** and names can be removed using the familiar swipe-to-delete gesture.

Lesson 6: Adding synchronization

> **Prerequisites**
> To complete this lesson you must have SQL Anywhere installed on your computer. If you receive any errors or warnings during this lesson, make sure you have configured your installation correctly and have the proper environment variables set.

Now that the Names application can add and remove names from an UltraLite database, you add synchronization to a consolidated database server on your computer.

Creating the consolidated database

1. Open Sybase Central.
2. From the **Tools** menu, click **SQL Anywhere 12** » **Create Database**.
3. Click **Next** on the **Welcome** screen and the **Select a Location** screen.
4. Choose a location to save the database file and name it *Names.db*. Click **Finish**.

   The rest of the options can remain at their defaults.
5. When the database has been created, close the popup window.

Create the consolidated table

To synchronize, you first need to create a table on the consolidated database:

1. If the left selection is **Tasks** or **Search**, change it to **Folders**, by clicking **Folders** from the **View** menu.
2. In the **Folders** pane, Control-click the **Tables** element under the **Names** database and click **New** » **Table**.
3. Name the table **Names** and click **Finish**.

   This creates the table and ready the cursor to name the primary key column of the table.
4. Name the primary key of the table **id** with a **uniqueidentifier** type.
5. Click the **[...]** button under the **Value** heading and set the default value to the **User-defined value**: **NEWID()**.
6. Make sure the **Literal string** option is cleared and click **OK**.
7. Add another column by clicking the **New Column** button on the toolbar.
8. Name the new column **name** of type VARCHAR and size 254.
9. Clear the **Null** option for the name column.
10. Click the **Unique** option for the name column.
11. Click the **Save** button on the toolbar.
12. Disconnect from the database.

Create an ODBC data source

To set up MobiLink, the system must have an ODBC data source for the consolidated database. Before you can setup the ODBC data source, you must first install the SQL Anywhere ODBC driver:

1. Open a Terminal.
2. Run the following command to source the SQL Anywhere configuration file:

   ```
   source ./sa_config.sh
   ```

   Sourcing the configuration file allows you to use the dbdsn utility.
3. Run the following command to create the ODBC data source:

   ```
   dbdsn -w "Names" -c "UID=dba;PWD=sql;DBF=/Users/user/Names.db"
   ```

If the location of your database is different than this, make sure to update the DBF option.

Creating a MobiLink synchronization model

In order for MobiLink to perform the synchronization, you need to configure the synchronization scripts. To make configuration easier, Sybase Central provides script templates for many common forms of synchronization.

1. Start Sybase Central.
2. Under the **Tools** menu, click **MobiLink 12** » **New Project**.
3. Name the project **NamesProject** and click **Next**.
4. Check **Add a consolidated database to the project**. Provide a **Database display name** of **NamesCondb**. Provide a connection string of `UID=dba;PWD=sql;DSN=Names`. You can also build this string by clicking the **Edit** button. Click **Next**.
5. Click **Create a new model** and click **Next**.
6. Click **Add a remote schema name to the project**, enter a name, and, with **UltraLite schema** selected, click **Finish**.

   You are asked whether you want to install the MobiLink system setup. This remote schema is not actually used, but since the remote schema is on the iPhone, this option is the simplest.
7. A message appears saying that MobiLink has not yet been installed and asks if it should be installed now. Click **Yes**.

   This creates the tables required by MobiLink in the database, as well as some stored procedures.
8. Type **NamesModel** as the model name and click **Next**.
9. Click all three checkboxes to acknowledge the MobiLink requirements, and click **Next**.

   For synchronization to function correctly, MobiLink assumes a few things about the primary keys of the tables. Since the Names application already abides by these assumptions, no changes are required.
10. Click the **NamesCondb** consolidated database and click **Next**.
11. Click **No, create a new remote database schema** and click **Next**.
12. Check the **Names** table from the list of tables and click **Next**.

    Any other tables listed are used by MobiLink behind the scenes and can safely be ignored for synchronization.
13. For the **Download type**, click **Timestamp-based download**.

    This option provides a good default implementation of synchronization by only synchronizing changes since the last synchronization. This option saves bandwidth for the iPhone since no unnecessary data is transferred.
14. Since all other settings should be kept as their defaults, click **Finish**.

15. If you would like to see the other options available as MobiLink templates, you can click through all of the steps in the wizard.

Deploying the MobiLink synchronization model

Now that the synchronization model has been created, it should be visible in the left side **Folders** view. If the **Folders** view is not showing, click **View** » **Folders**.

To deploy the synchronization model:

1. Control-click the **NamesModel** in the **Folder** view and click **Deploy**.
2. Clear **Remote database and synchronization client** leaving **Consolidated database** and **MobiLink server** selected and click **Next**.
3. Click **Save changes to the following SQL file** and **Connect to the consolidated database to directly apply the changes** with **NamesCondb** selected in the list. Click **Next**.
4. Accept the creation of the new directory if prompted.
5. Enter a MobiLink user and password.

   MobiLink users are distinct from SQL Anywhere database users. You should use a different username than **dba**. This tutorial uses "user" and "password" respectively. These values are used later in the tutorial.
6. Ensure that **Register this user in the consolidated database for MobiLink authentication** is selected and click **Finish**.
7. Accept the creation of the new directory if prompted.
8. Once the deployment window shows the deployment completing, dismiss it by clicking **Close**.

Starting the MobiLink server

Now that the synchronization model has been deployed, the consolidated database contains all the information required for MobiLink to function. The deployment also created scripts for launching the MobiLink server:

1. Open a Terminal.
2. Navigate the Terminal session to where deployment saved the launch script. By default this should be: *~/NamesProject/NamesModel/mlsrv*.
3. Ensure the script is executable:

   ```
   chmod u+x NamesModel_mlsrv.sh
   ```
4. Start the MobiLink server:

   ```
   ./NamesModel_mlsrv.sh "DSN=Names"
   ```

Adding synchronization to the iPhone application

You now have a consolidated database with a MobiLink server running.

For the application to be able to synchronize to it, you first have to enable synchronization. In the **openConnection** method in the **DataAccess** class after the **if-block** that contains the database creation (ULDatabaseManager::Init) add the following line:

```
ULDatabaseManager::EnableTcpipSynchronization();
```

Now that the database is expecting to be synchronized, you can add the following method to the **DataAccess** class. Also add the method signature to the header file.

```
- (void)synchronize {
    NSString *            result = nil;
    ul_sync_info        info;

    // Initialize the sync info struct
    connection->InitSyncInfo(&info);

    // Set the sync parameters
    info.user_name = (char*)"user";     // Set to your username
    info.password = (char*)"password";  // Set to your password
    info.version = (char*)"NamesModel";
    info.stream = "tcpip";
    info.stream_parms = (char*)"host=localhost";

    // Display the network activity indicator in the status bar
    [[UIApplication sharedApplication]
     setNetworkActivityIndicatorVisible:YES];

    // Sync and get the result
    if (connection->Synchronize(&info)) {
        result = @"Sync was successful.";
    } else {
        // Get the error message and log it.
        char    errorMsg[80];

        connection->GetLastError()->GetString(errorMsg, 80);
        NSLog(@"Sync failed: %s", errorMsg);
        result = [NSString stringWithFormat:@"Sync failed: %s", errorMsg];
    }

    // Stop showing the activity indicator
    [[UIApplication sharedApplication]
setNetworkActivityIndicatorVisible:NO];

    [[[[UIAlertView alloc]
       initWithTitle:@"Synchronization"
       message:result
       delegate:nil
       cancelButtonTitle:nil
       otherButtonTitles:@"OK", nil] autorelease] show];

}
```

This is the simplest synchronization a client can perform. In an enterprise application, you would likely want to set a callback method to get the progress of the synchronization and perform the synchronization on a separate thread so as to not block the application.

Also, in the current implementation, synchronization occurs on the main event thread. It is not recommended to block the main thread this way. In the next lesson you use a separate thread to perform the synchronization and add a callback method to observe the synchronization and display the progress.

To let the user choose when to synchronize, add a button to the navigation bar. Pressing the button calls the following method in **RootViewController**:

```
- (void)sync {
    [[DataAccess sharedInstance] synchronize];
    [self.tableView reloadData];
}
```

To create the button, add the following to the **RootViewController**'s **viewDidLoad** method:

```
    // Little button with the refresh sign on the left in the nav bar
    self.navigationItem.leftBarButtonItem =
    [[UIBarButtonItem alloc]
     initWithBarButtonSystemItem:UIBarButtonSystemItemRefresh
     target:self
     action:@selector(sync)];
```

You can now build and run the application. Whenever you press the **Refresh** button, the database on the iPhone is synchronized with the consolidated SQL Anywhere database.

Note
Before synchronizing, you need to insert data to the consolidated database to see it on the iPhone, or insert data on the iPhone to see it in the consolidated database.

Lesson 7: Adding a progress display

In the previous lesson, you added rudimentary synchronization that was performed on the main thread. Blocking the main thread in such a way is not recommended. In this lesson you move the synchronization to a background thread, and add a synchronization observing method to update a progress display.

Creating the progress toolbar

To show the progress of the synchronization, you use a progress bar with a label that is placed in the bottom navigation toolbar, similar to the look of the **Mail** application when downloading new messages. To recreate this look, you must create a custom view:

1. Control-click the **Classes** folder in the **Groups & Files** pane. From the **Add** menu click **New File**.
2. Under **Cocoa Touch Class**, click **UIViewController subclass**.
3. Clear **UITableViewController subclass**.
4. Click **With XIB for user interface**.
5. Click **Next**.
6. Name the file: *ProgressToolbarViewController.m* and place it in the *Classes* subdirectory. Click the box to have a header file created.

7. Click **Finish**.

8. Move the *ProgressToolbarViewController.xib* file to the *Resources* directory.

Before creating the layout in **Interface Builder**, replace the interface definition with the following in **ProgressToolbarViewController**:

```
@interface ProgressToolbarViewController : UIViewController {
    IBOutlet UILabel        *label;
    IBOutlet UIProgressView    *progressBar;
}
@end
```

Add the properties in the interface:

```
@property (readonly) IBOutlet UILabel            *label;
@property (readonly) IBOutlet UIProgressView    *progressBar;
```

Synthesize the properties in the implementation file:

```
@synthesize label;
@synthesize progressBar;
```

Add a release call in the **dealloc** method:

```
- (void)dealloc {
    [super dealloc];
    [label release];
    [progressBar release];
}
```

Open the **ProgressToolbarViewController** by double-clicking *ProgressToolbarViewController.xib* in the Xcode *Resources* folder. Since this view is displayed in a toolbar, you must size it appropriately, and set its background properties:

1. In the **Document** window (Command-0), click **View**.

2. In the **Attributes Inspector** (Command-1), set the simulated status bar to **Unspecified**.

3. Click **Background** and set **Background opacity** to 0%.

4. Clear the **Opaque** setting.

5. In the **Size Inspector** (Command-3), set the width to 232 and the height to 44.

Add the progress view

1. Click and drag a **UIProgressView** from the **Library** to the **View**.

2. In the **Size Inspector** (Command-3), set the position of the progress view to 26, 29.

3. Set the width of the progress view to 186.

4. In the **Attributes Inspector** (Command-1), set the style to **Bar** and the progress to zero.

Add the label

1. Click and drag a **UILabel** from the **Library** to the **View**.
2. In the **Size Inspector** (Command-3), set the position of the label to 14, 5.
3. Set the size of the label to 210, 16.
4. In the **Attribute Inspector** (Command-1), set the text to **Sync Progress**.
5. Set the layout alignment to center.
6. Set the font to Helvetica Bold, size 12.
7. Set the text color to white.
8. Set the shadow color to RGB: (103, 114, 130), with 100% opacity.

Connect the new label and progress view to the outlets

1. Click **File's Owner** in the **Document** window.
2. In the **Connectors Inspector** (Command-2), link the label outlet to the **UILabel** you created in the last steps.
3. Link the progress bar outlet to the **UIProgressView** you created in the last steps.
4. Save the XIB file and close **Interface Builder**.

The **ProgressBar** view is added to the **RootViewController**'s toolbar. However, the **DataAccess** object also uses a reference to it to display the progress of the synchronization. Add the following instance variable to the interface:

```
ProgressToolbarViewController * progressToolbar;
```

and add the property to the **DataAccess** class:

```
    @property (retain, readwrite) IBOutlet ProgressToolbarViewController
*    progressToolbar;
```

Import the **ProgressToolbarViewController** header into the DataAccess header:

```
#import "ProgressToolbarViewController.h"
```

Synthesize the property in the implementation.

```
@synthesize progressToolbar
```

To add the progress view to the toolbar, add the following to the **RootViewController**'s **viewDidLoad** method:

```
    // Create progress display
    ProgressToolbarViewController *    progress =
```

```
[[ProgressToolbarViewController alloc]
 initWithNibName:@"ProgressToolbarViewController"
 bundle:nil];

// Register the toolbar with the DataAccess
[[DataAccess sharedInstance] setProgressToolbar:progress];

// Setup UIBarButtonItems
UIBarButtonItem    *               space =
[[UIBarButtonItem alloc]
 initWithBarButtonSystemItem:UIBarButtonSystemItemFlexibleSpace
 target:nil
 action:nil];
UIBarButtonItem    *               progressButtonItem =
[[UIBarButtonItem alloc] initWithCustomView:progress.view];

// Put them in the toolbar
self.toolbarItems =
[NSArray arrayWithObjects:space, progressButtonItem, space, nil];
[space release];
[progressButtonItem release];
```

The **RootViewController** now has a progress view in its toolbar, even though the toolbar is hidden. In the next section, you move synchronization to a background thread and display the progress toolbar during the synchronization.

Performing synchronization in a background thread

So far, everything the application does is performed on the main thread of the application. Since this is also the thread used to draw the interface and process user event such as touches, blocking it is not recommended.

The flow of the application during the sync is as follows:

1. The user clicks the sync button in **RootViewController**. The toolbar progress view appears.
2. **RootViewController** initiates the sync on a detached thread, passing itself as an argument to the synchronization function. The main thread continues with the sync running in another thread.
3. The synchronization function updates the user interface using **performSelectorOnMainThread** to update the progress display throughout the synchronization.
4. When synchronization is complete, the background thread displays an alert box showing the result of the sync, with **RootViewController** set as the delegate to the alert so that it knows when the alert is dismissed.
5. Once **RootViewController** is told the alert box is dismissed, the toolbar progress view is hidden.

Since the **RootViewController** must implement the **UIAlertViewDelegate** protocol to handle the dismissing of the alert view, change its header file to the following:

```
@interface RootViewController : UITableViewController <UIAlertViewDelegate> {
}

// Displays the screen to add a name.
- (void)showAddNameScreen;

@end
```


The only method from the **UIAlertViewDelegate** protocol that the **RootViewController** needs to implement is the **alertView:clickedButtonAtIndex:** method. This method is called when the user presses a button of the **UIAlertView**. Since the view only has a single button, you don't need to inspect which button was pressed. Put the following code in the *RootViewController.m* file.

```
- (void)alertView:(UIAlertView *)alertView
clickedButtonAtIndex:(NSInteger)buttonIndex {
    NSLog(@"User dismissed sync alert, refreshing table.");
    [self.tableView reloadData];
    [self.navigationController setToolbarHidden:YES animated: YES];
}
```

Also in *RootViewController.m*, change the sync method to do the synchronization call on a separate thread by using **detachNewThreadSelector**:

```
- (void)sync {
    [self.navigationController setToolbarHidden:NO animated: YES];
    [NSThread detachNewThreadSelector:@selector(synchronize:)
                             toTarget:[DataAccess sharedInstance]
                           withObject:self];
}
```

Now that the synchronization is on a separate thread, you need to change a few things. As you might have noticed in the **RootViewController**'s sync method, **DataAccess**' synchronize method is being passed a parameter. Change its signature to the following:

```
- (void)synchronize:(id<UIAlertViewDelegate>)sender;
```

Update the *DataAccess.mm* implementation file with the following:

```
// Since the synchronization method uses auto-released object instances, you
also need
// to create an NSAutoreleasePool and release it at the end of the
synchronization:

- (void)synchronize:(id<UIAlertViewDelegate>)sender {
    NSAutoreleasePool *    pool = [[NSAutoreleasePool alloc] init];
    // Rest of synchronize method...

    [pool release];
}
```

Still in the synchronize method, you also need to set the passed-in **RootViewController** as the alert delegate. To do this, update the previously create **UIAlertView** and change the delegate from **nil** to **sender**:

```
    // Set RootViewController as the alert delegate
    if (sender != nil) {
        NSLog(@"Showing Alert with sync result.");
        [[[[UIAlertView alloc]
           initWithTitle:@"Synchronization"
           message:result
           delegate:sender // changed from nil to sender
           cancelButtonTitle:nil
           otherButtonTitles:@"OK", nil] autorelease] show];
    } else {
        NSLog(@"Not showing alert since sender was nil.");
    }
```

You now need to create the synchronization callback that updates the progress bar to the appropriate completion and display the correct message.

However, since the synchronization callback is executed on a non-main thread (the synchronization thread created by the **RootViewController**) you must use **performSelectorOnMainThread** to update the user interface. This method only allows the passing of a single parameter, so in order to pass the progress, as well as the message, create a Objective-C class named **UpdateInfo** with two properties, a float, and an **NSString** in the header file:

```
@interface UpdateInfo : NSObject {
    NSString *    message;
    float        progress;
}

// Message to display in the progress view
@property (readonly) NSString *    message;

// Progress of the sync [0.0-1.0]
@property (readonly) float        progress;

// Preferred initializer
- (id)initWithMessage:(NSString*) message andProgress:(float)progress;

@end
```

Add the following to the implementation file:

```
// The implementation is a single constructor that takes both parameters:
@implementation UpdateInfo

@synthesize message;
@synthesize progress;

- (id)initWithMessage:(NSString*) msg andProgress:(float) syncProgress {
    if (self = [super init]) {
        message = msg;
        progress = syncProgress;
    }

    return self;
}

@end
```

Now that you can bundle all the information needed by the user interface updating method, you can define it in the **DataAccess** class:

```
- (void)updateSyncProgress:(UpdateInfo *)info {
    progressToolbar.label.text = info.message;
    progressToolbar.progressBar.progress = info.progress;
}
```

Now import *UpdateInfo.h* in *DataAccess.h*.

```
#import "UpdateInfo.h"
```

You can also define the callback method in the *DataAccess.mm* implementation file along with the other static methods:


```
static void UL_CALLBACK_FN progressCallback(ul_synch_status * status) {
    // Sync information for the GUI
    float        percentDone = 0.0;
    NSString *    message;

    // Note: percentDone is approximate.
    switch (status->state) {
        case UL_SYNCH_STATE_STARTING:
            percentDone = 0;
            message = @"Starting Sync";
            break;
        case UL_SYNCH_STATE_CONNECTING:
            percentDone = 5;
            message = @"Connecting to Server";
            break;
        case UL_SYNCH_STATE_SENDING_HEADER:
            percentDone = 10;
            message = @"Sending Sync Header";
            break;
        case UL_SYNCH_STATE_SENDING_TABLE:
            percentDone =
            10 + 25 *
            (status->sync_table_index / status->sync_table_count);
            message =
            [NSString stringWithFormat:@"Sending Table %s: %d of %d",
             status->table_name,
             status->sync_table_index,
             status->sync_table_count];
            break;
        case UL_SYNCH_STATE_SENDING_DATA:
            percentDone =
            10 + 25 *
            (status->sync_table_index / status->sync_table_count);
            message = @"Sending Name Changes";
            break;
        case UL_SYNCH_STATE_FINISHING_UPLOAD:
        case UL_SYNCH_STATE_RECEIVING_UPLOAD_ACK:
            percentDone = 50;
            message = @"Finishing Upload";
            break;
        case UL_SYNCH_STATE_RECEIVING_TABLE:
        case UL_SYNCH_STATE_RECEIVING_DATA:
            percentDone =
            50 + 25 *
            (status->sync_table_index / status->sync_table_count);
            message =
            [NSString
             stringWithFormat:@"Receiving Table %s: %d of %d",
             status->table_name,
             status->sync_table_index,
             status->sync_table_count];
            break;
        case UL_SYNCH_STATE_COMMITTING_DOWNLOAD:
        case UL_SYNCH_STATE_SENDING_DOWNLOAD_ACK:
            percentDone = 80;
            message = @"Committing Downloaded Updates";
            break;
        case UL_SYNCH_STATE_DISCONNECTING:
            percentDone = 90;
            message = @"Disconnecting from Server";
            break;
        case UL_SYNCH_STATE_DONE:
            percentDone = 100;
            message = @"Finished Sync";
```

```
            break;
        case UL_SYNCH_STATE_ERROR:
            percentDone = 95;
            message = @"Error During Sync";
            break;
        case UL_SYNCH_STATE_ROLLING_BACK_DOWNLOAD:
            percentDone = 100;
            message = @"Rolling Back due to Error";
            break;
        default:
            percentDone = 100;
            NSLog(@"Unknown sync state: '%d'", status->state);
            break;
    }

    // Wrap the GUI info in an object and have the main thread update
    UpdateInfo * info = [[[UpdateInfo alloc]
                          initWithMessage:message
                          andProgress:percentDone / 100]
                         autorelease];
    [[DataAccess sharedInstance]
     performSelectorOnMainThread:@selector(updateSyncProgress:)
     withObject:info waitUntilDone:YES];
}
```

With everything in place, you can now set the sync observer to the callback method in the synchronize method in the *DataAccess.mm* file:

```
// Set the sync parameters
    info.user_name = (char*)"user";     // Set to your username
    info.password = (char*)"password";  // Set to your password
    info.version = (char*)"NamesModel";
    info.stream = "tcpip";
    info.stream_parms = (char*)"host=localhost";
    info.observer = progressCallback;  // Add this line
```

Conclusion

The Names application is now a fully functioning iPhone application with an UltraLite database which can monitor synchronization with a MobiLink server. Using this example application as a starting point, you can use these concepts to develop applications that make use of UltraLite and MobiLink technologies.

API reference

This section provides the UltraLite C/C++ API.

UltraLite C/C++ common API reference

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.

Header file

- `ulglobal.h`

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"$(VCDIR)\include" \
  /I"$(SQLANY12)\SDK\Include"

  sample.obj: sample.cpp
   cl $(CompileOptions) $(IncludeFolders) sample.cpp
  ```

  VCDIR is your Visual C++ directory and *SQLANY12* 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_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 Visual C++ Smart Device projects.

Remarks

Applies to Windows Mobile applications.

See also

- "UltraLite application development for Windows Mobile" on page 56

Example

```
/D UNDER_CE
```

UL_RS_STATE enumeration

Specifies possible result set or cursor states.

Syntax

```
public enum UL_RS_STATE
```

Members

Member name	Description
UL_RS_STATE_ERROR	Error.
UL_RS_STATE_UNPREPARED	Not prepared.
UL_RS_STATE_ON_ROW	On a valid row.
UL_RS_STATE_BEFORE_FIRST	Before the first row.
UL_RS_STATE_AFTER_LAST	After the last row.
UL_RS_STATE_COMPLETED	Closed.

ul_column_sql_type enumeration

Represents the SQL types for a column.

Syntax

```
public enum ul_column_sql_type
```


Members

Member name	Description
UL_SQLTYPE_BAD_INDEX	Represents that the column at the specified index does not exist.
UL_SQLTYPE_S_LONG	Represents that the column contains a signed long.
UL_SQLTYPE_U_LONG	Represents that the column contains an unsigned long.
UL_SQLTYPE_S_SHORT	Represents that the column contains a signed short.
UL_SQLTYPE_U_SHORT	Represents that the column contains an unsigned short.
UL_SQLTYPE_S_BIG	Represents that the column contains a signed 64-bit integer.
UL_SQLTYPE_U_BIG	Represents that the column contains an unsigned 64-bit integer.
UL_SQLTYPE_TINY	Represents that the column contains an unsigned 8-bit integer.
UL_SQLTYPE_BIT	Represents that the column contains a 1-bit flag.
UL_SQLTYPE_TIMESTAMP	Represents that the column contains timestamp information.
UL_SQLTYPE_DATE	Represents that the column contains date information.
UL_SQLTYPE_TIME	Represents that the column contains time information.
UL_SQLTYPE_DOUBLE	Represents that the column contains a double precision floating-point number (8 bytes).
UL_SQLTYPE_REAL	Represents that the column contains a single precision floating-point number (4 bytes).
UL_SQLTYPE_NUMERIC	Represents that the column contains exact numerical data, with specified precision and scale.
UL_SQLTYPE_BINARY	Represents that the column contains binary data with a specified maximum length.
UL_SQLTYPE_CHAR	Represents that the column contains character data with a specified length.
UL_SQLTYPE_LONGVARCHAR	Represents that the column contains character data with variable length.
UL_SQLTYPE_LONGBINARY	Represents that the column contains binary data with variable length.
UL_SQLTYPE_UUID	Represents that the column contains a UUID.

Member name	Description
UL_SQLTYPE_ST_GEOMETRY	Represents that the column contains spatial data in the form of points.
UL_SQLTYPE_TIME-STAMP_WITH_TIME_ZONE	Represents that the column contains timestamp and time zone information.

Remarks

These values correspond to SQL column types.

ul_column_storage_type enumeration

Represents the host variable types for a column.

Syntax

```
public enum ul_column_storage_type
```

Members

Member name	Description
UL_TYPE_BAD_INDEX	Represents an invalid value.
UL_TYPE_S_LONG	Represents a ul_s_long (32 bit signed int).
UL_TYPE_U_LONG	Represents a ul_u_long (32 bit unsigned int).
UL_TYPE_S_SHORT	Represents a ul_s_short (16 bit signed int).
UL_TYPE_U_SHORT	Represents a ul_u_short (16 bit unsigned int).
UL_TYPE_S_BIG	Represents a ul_s_big (64 bit signed int).
UL_TYPE_U_BIG	Represents a ul_u_big (64 bit unsigned int).
UL_TYPE_TINY	Represents a ul_byte (8 bit unsigned).
UL_TYPE_BIT	Represents a ul_byte (8 bit unsigned, 1 bit used).
UL_TYPE_DOUBLE	Represents a ul_double (double).
UL_TYPE_REAL	Represents a ul_real (float).
UL_TYPE_BINARY	Represents a ul_binary (2 byte length followed by byte array).

Member name	Description
UL_TYPE_TIMESTAMP_STRUCT	Represents a DECL_DATETIME.
UL_TYPE_TCHAR	Represents a character array (string buffer).
UL_TYPE_CHAR	Represents a char array (string buffer).
UL_TYPE_WCHAR	Represents a ul_wchar (UTF16) array.
UL_TYPE_GUID	Represents a GUID structure.

Remarks

These values are used to identify the host variable type required for a column, and to indicate how UltraLite should fetch values.

ul_error_action enumeration

Specifies possible error actions returned from callback.

Syntax

```
public enum ul_error_action
```

Members

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

Remarks

Not all actions apply to all error codes.

ul_sync_state enumeration

Indicates the current stage of synchronization.

Syntax

```
public enum ul_sync_state
```

Members

Member name	Description
UL_SYNC_STATE_STARTING	The synchronization is starting; initial parameter validation is complete and synchronization result will be saved.
UL_SYNC_STATE_CONNECTING	Connecting to the MobiLink server.
UL_SYNC_STATE_SENDING_HEADER	The synchronization connection is established and initial data is about to be sent.
UL_SYNC_STATE_SENDING_TABLE	A table is about to be sent.
UL_SYNC_STATE_SENDING_DATA	Schema information or row data is being sent.
UL_SYNC_STATE_FINISHING_UPLOAD	The upload stage is complete and state information is about to be committed.
UL_SYNC_STATE_RECEIVING_UPLOAD_ACK	About to read data from the server, starting with the upload acknowledgement.
UL_SYNC_STATE_RECEIVING_TABLE	A table is about to be received.
UL_SYNC_STATE_RECEIVING_DATA	Data for the most recently identified table is being received.
UL_SYNC_STATE_COMMITTING_DOWNLOAD	The download stage is complete and downloaded rows are about to be committed.
UL_SYNC_STATE_ROLLING_BACK_DOWNLOAD	An error occurred during download and the download is being rolled back.
UL_SYNC_STATE_SENDING_DOWNLOAD_ACK	An acknowledgement that the download is complete is being sent.
UL_SYNC_STATE_DISCONNECTING	About to disconnect from the server.
UL_SYNC_STATE_DONE	Synchronization has completed successfully.
UL_SYNC_STATE_ERROR	Synchronization has completed, but with an error.

Remarks

You should not assume that the synchronization states occur in the order listed below.

ul_validate_status_id enumeration

Specifies possible status IDs for the UltraLite validation tool.

Syntax

```
public enum ul_validate_status_id
```

Members

Member name	Description	Value
UL_VALID_NO_ERROR	No error occurred.	0
UL_VALID_START	Start validation.	1
UL_VALID_END	End validation. Parm1 tracks the resulting sqlcode, which indicates success or failure.	2
UL_VALID_CHECKING_PAGE	Send a periodic status message while checking database pages. Parm1 tracks a number associated with the page. The order is not defined.	10
UL_VALID_CHECKING_TABLE	Checking a table. Parm1 tracks the table name.	20
UL_VALID_CHECKING_INDEX	Checking an index. Parm1 stores the table name and parm2 stores the index name.	21
UL_VALID_HASH_REPORT	Reporting on the index hash use. (development version only) Parm1 tracks the table name, parm2 tracks the index name, parm3 tracks the number of visible rows, parm4 tracks the number of unique hash values, and parm5 tracks the maximum number of times a hash entry appears.	30

Member name	Description	Value
UL_VALID_REDUNDANT_INDEX	A redundant index was found. (development version only) Parm1 tracks the table name, parm2 tracks the redundant index name, and parm3 tracks the name of index that makes it redundant.	31
UL_VALID_DUPLICATE_INDEX	Two indexes are the same. (development version only) Parm1 tracks the table name, parm2 tracks the name of first index, and parm3 tracks the name of second index.	32
UL_VALID_DATABASE_ERROR	An error occurred accessing the database. Check the SQLCODE for more information.	100
UL_VALID_STARTUP_ERROR	Error starting the database. (for low-level access)	101
UL_VALID_CONNECT_ERROR	Error connecting to the database.	102
UL_VALID_INTERRUPTED	Validation process interrupted.	103
UL_VALID_CORRUPT_PAGE_TABLE	Page table is corrupt.	110
UL_VALID_FAILED_CHECKSUM	Page checksum failed. Parm1 tracks a number associated with the page	111
UL_VALID_CORRUPT_PAGE	A page is corrupt. Parm1 tracks a number associated with the page.	112
UL_VALID_ROWCOUNT_MISMATCH	The number of rows in the index is different from the table row count. Parm1 tracks the table name, and parm2 tracks index name.	120
UL_VALID_BAD_ROWID	There is an invalid row identifier in the index. Parm1 tracks the table name, and parm2 tracks the index name.	121

ul_binary structure

Sets and fetches binary values from a table in the database.

Syntax

```
public typedef struct ul_binary
```

Members

Member name	Type	Description
data	ul_byte	The actual data to be set (for insert) or that was fetched (for select).
len	ul_length	The number of bytes in the value.

ul_error_info structure

Stores complete information about an UltraLite error.

Syntax

```
public typedef struct ul_error_info
```

Members

Member name	Type	Description
sqlcode	an_sql_code	The SQLCODE value.
sqlcount	ul_s_long	The SQLCOUNT value.

See also

- “ULErrorInfoString method [UltraLite Embedded SQL]” on page 255
- “ULErrorInfoURL method [UltraLite Embedded SQL]” on page 256
- “ULErrorInfoInitFromSqlca method [UltraLite Embedded SQL]” on page 254
- “ULErrorInfoParameterCount method [UltraLite Embedded SQL]” on page 255
- “ULErrorInfoParameterAt method [UltraLite Embedded SQL]” on page 254

ul_stream_error structure

Stores synchronization communication stream error information.

Syntax

```
public typedef struct ul_stream_error
```

Members

Member name	Type	Description
error_string	char	A string with additional information, if available, for the stream_error_code value.
stream_error_code	ss_error_code	The specific stream error. See the ss_error_code enumeration for possible values.
system_error_code	asa_int32	A system-specific error code. For more information about error codes, see your platform documentation.

ul_sync_info structure

Stores synchronization data.

Syntax

```
public typedef struct ul_sync_info
```

Members

Member name	Type	Description
additional_parms	const char *	A string of name value pairs "name=value;" with extra parameters.
auth_parms	const 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_sync_info *	Initialize verification.

Member name	Type	Description
keep_partial_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	const 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_sync_observer_fn	A pointer to a callback function or event handler that monitors synchronization. This parameter is optional.
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	const 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 or not the client provides download acknowledgements.
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	const char *	The options to configure the network protocol you selected.

Member name	Type	Description
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	const char *	A string that the MobiLink server uses to identify a unique MobiLink user.
version	const 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 method 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 method returns an error (SQLE_SYNC_INFO_INVALID or its equivalent).

See also

- "MobiLink client network protocol options" [*MobiLink - Client Administration*]

ul_sync_result structure

Stores the synchronization result so that appropriate action can be taken in the application.

Syntax

```
public typedef struct ul_sync_result
```

Members

Member 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.

Member name	Type	Description
error_status	ul_error_info	The error status from the last synchronization.
ignored_rows	ul_bool	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.
received	ul_sync_stats	Download statistics.
sent	ul_sync_stats	Upload statistics.
stream_error	ul_stream_error	The communication stream error information.
timestamp	SQLDATETIME	The time and date of the last synchronization.
upload_ok	ul_bool	True if the upload was successful; false otherwise.

ul_sync_stats structure

Reports the statistics of the synchronization stream.

Syntax

```
public typedef struct ul_sync_stats
```

Members

Member 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_sync_status structure

Returns synchronization progress monitoring data.

Syntax

```
public typedef struct ul_sync_status
```

Members

Member 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_sync_info *	A pointer to the ul_sync_info_a structure.
received	ul_sync_stats	Returns download statistics.
sent	ul_sync_stats	Returns upload statistics.
sqlca	SQLCA *	The connection's active SQLCA.
state	ul_sync_state	One of the many supported states.
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	1..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.
table_name_w2	ul_wchar	Name of the current table.
user_data	ul_void *	User data passed in to the ULSetSynchronizationCallback method or set in the ul_sync_info structure.

See also

- “ul_sync_state enumeration [UltraLite C and Embedded SQL datatypes]” on page 105

ul_validate_data structure

Stores validation status information for the callback.

Syntax

```
public typedef struct ul_validate_data
```


Members

Member 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	enumeration	The possible parameter types.
parms	struct ul_validate_data::@3	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.

ULVF_DATABASE variable

Used to validate database.

Syntax

```
#define ULVF_DATABASE
```

Remarks

Verify all database pages using page checksums and additional checks.

See also

- “ULDatabaseManager.ValidateDatabase method [UltraLite C++]” on page 155
- “ULConnection.ValidateDatabase method [UltraLite C++]” on page 144

ULVF_EXPRESS variable

Used to perform a faster, though less thorough, validation.

Syntax

```
#define ULVF_EXPRESS
```

Remarks

This flag modifies others specified.

See also

- "ULDatabaseManager.ValidateDatabase method [UltraLite C++]" on page 155
- "ULConnection.ValidateDatabase method [UltraLite C++]" on page 144

ULVF_FULL_VALIDATE variable

Performs all types of validation on the database.

Syntax

```
#define ULVF_FULL_VALIDATE
```

See also

- "ULDatabaseManager.ValidateDatabase method [UltraLite C++]" on page 155
- "ULConnection.ValidateDatabase method [UltraLite C++]" on page 144

ULVF_IDX_HASH variable

Reports effectiveness of the index hashes (development version only).

Syntax

```
#define ULVF_IDX_HASH
```

Remarks

Check that table and index row counts match.

See also

- "ULDatabaseManager.ValidateDatabase method [UltraLite C++]" on page 155
- "ULConnection.ValidateDatabase method [UltraLite C++]" on page 144

ULVF_IDX_REDUNDANT variable

Checks redundant indexes (development version only).

Syntax

```
#define ULVF_IDX_REDUNDANT
```

Remarks

Check that table and index row counts match.

See also

- "ULDatabaseManager.ValidateDatabase method [UltraLite C++]" on page 155
- "ULConnection.ValidateDatabase method [UltraLite C++]" on page 144

ULVF_INDEX variable

Used to validate indexes.

Syntax

```
#define ULVF_INDEX
```

Remarks

Check the integrity of the index.

See also

- "ULDatabaseManager.ValidateDatabase method [UltraLite C++]" on page 155
- "ULConnection.ValidateDatabase method [UltraLite C++]" on page 144

ULVF_TABLE variable

Used to validate table(s).

Syntax

```
#define ULVF_TABLE
```

Remarks

Check that table and index row counts match.

See also

- "ULDatabaseManager.ValidateDatabase method [UltraLite C++]" on page 155
- "ULConnection.ValidateDatabase method [UltraLite C++]" on page 144

UL_AS_SYNCHRONIZE variable

Provides the name of the callback message used to indicate an ActiveSync synchronization.

Syntax

```
#define UL_AS_SYNCHRONIZE
```

Remarks

This applies to Windows Mobile applications that use ActiveSync only.

See also

- "Add ActiveSync synchronization to your application" on page 60

UL_SYNC_ALL variable

Synchronizes all tables in the database that are not marked as "no sync", including tables that are not in any publication.

Syntax

```
#define UL_SYNC_ALL
```

See also

- "ul_sync_info structure [UltraLite C and Embedded SQL datatypes]" on page 110
- "UL_SYNC_ALL_PUBS variable [UltraLite C and Embedded SQL datatypes]" on page 118

UL_SYNC_ALL_PUBS variable

Synchronizes all tables in a publication.

Syntax

```
#define UL_SYNC_ALL_PUBS
```

See also

- "ul_sync_info structure [UltraLite C and Embedded SQL datatypes]" on page 110
- "UL_SYNC_ALL variable [UltraLite C and Embedded SQL datatypes]" on page 118

UL_SYNC_STATUS_FLAG_IS_BLOCKING variable

Defines a bit set in the ul_sync_status.flags field to indicate that the synchronization is blocked awaiting a response from the MobiLink server.

Syntax

```
#define UL_SYNC_STATUS_FLAG_IS_BLOCKING
```

Remarks

Identical synchronization progress messages are generated periodically while this is the case.

UL_TEXT variable

Prepares constant strings to be compiled as single-byte strings or wide-character strings.

Syntax

```
#define UL_TEXT
```


Remarks

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_VALID_IS_ERROR variable

Evaluates true if a given ul_validate_status_id is an error status.

Syntax

```
#define UL_VALID_IS_ERROR
```

UL_VALID_IS_INFO variable

Evaluates true if a given ul_validate_status_id is an informational status.

Syntax

```
#define UL_VALID_IS_INFO
```

UltraLite C/C++ API reference

This section lists functions that you can use with the C++ interface.

> **Note**
> This version of the UltraLite C/C++ API supersedes all previous versions which have been deprecated. For more information, see "SQL Anywhere deprecated and discontinued features" [*SQL Anywhere 12 - Changes and Upgrading*].
>
> You can use the old implementation of the UltraLite C/C++ API by adding the *%SQLANY12%\SDK\C\ulcpp11.cpp* file to your UltraLite application project, where SQLANY12 is an environment variable that points to your SQL Anywhere installation directory.

Header file

- `ulcpp.h`

ULConnection class

Represents a connection to an UltraLite database.

Syntax

```
public class ULConnection
```

Members

All members of ULConnection class, including all inherited members.

Name	Description
CancelGetNotification method	Cancels any pending get-notification calls on all queues matching the given name.
ChangeEncryptionKey method	Changes the database encryption key for an UltraLite database.
Checkpoint method	Performs a checkpoint operation, flushing any pending committed transactions to the database.
Close method	Destroys this connection and any remaining associated objects.
Commit method	Commits the current transaction.
CountUploadRows method	Counts the number of rows that need to be uploaded for synchronization.
CreateNotificationQueue method	Creates an event notification queue for this connection.
DeclareEvent method	Declares an event which can then be registered for and triggered.
DestroyNotificationQueue method	Destroys the given event notification queue.
ExecuteScalar method	Executes a SQL SELECT statement directly, returning a single result.
ExecuteScalarV method	Executes a SQL SELECT statement string, along with a list of substitution values.
ExecuteStatement method	Executes a SQL statement string directly.
GetChildObjectCount method	Gets the number of currently open child objects on the connection.
GetDatabaseProperty method	Obtains the value of a database property.
GetDatabasePropertyInt method	Obtains the integer value of a database property.
GetDatabaseSchema method	Returns an object pointer used to query the schema of the database.
GetLastDownloadTime method	Obtains the last time a specified publication was downloaded.
GetLastError method	Returns the error information associated with the last call.
GetLastIdentity method	Gets the @@identity value.

Name	Description
GetNotification method	Reads an event notification.
GetNotificationParameter method	Gets a parameter for the event notification just read by the GetNotification method.
GetSqlca method	Gets the communication area associated with this connection.
GetSyncResult method	Gets the result of the last synchronization.
GetUserPointer method	Gets the pointer value last set by the SetUserPointer method.
GlobalAutoincUsage method	Obtains the percent of the default values used in all the columns that have global autoincrement defaults.
GrantConnectTo method	Grants access to an UltraLite database for a new or existing user ID with the given password.
InitSyncInfo method	Initializes the synchronization information structure.
OpenTable method	Opens a table.
PrepareStatement method	Prepares a SQL statement.
RegisterForEvent method	Registers or unregisters a queue to receive notifications of an event.
ResetLastDownloadTime method	Resets the last download time of a publication so that the application resynchronizes previously downloaded data.
RevokeConnectFrom method	Revokes access from an UltraLite database for a user ID.
Rollback method	Rolls back the current transaction.
RollbackPartialDownload method	Rolls back the changes from a failed synchronization.
SendNotification method	Sends a notification to all queues matching the given name.
SetDatabaseOption method	Sets the specified database option.
SetDatabaseOptionInt method	Sets a database option.
SetSynchronizationCallback method	Sets the callback to be invoked while performing a synchronization.
SetSyncInfo method	Creates a synchronization profile using the given name based on the given ul_sync_info structure.

Name	Description
SetUserPointer method	Sets an arbitrary pointer value in the connection for use by the calling application.
StartSynchronizationDelete method	Sets START SYNCHRONIZATION DELETE for this connection.
StopSynchronizationDelete method	Sets STOP SYNCHRONIZATION DELETE for this connection.
Synchronize method	Initiates synchronization in an UltraLite application.
SynchronizeFromProfile method	Synchronizes the database using the given profile and merge parameters.
TriggerEvent method	Triggers a user-defined event and sends notifications to all registered queues.
ValidateDatabase method	Validates the database on this connection.

CancelGetNotification method

Cancels any pending get-notification calls on all queues matching the given name.

Syntax

```
public virtual ul_u_long CancelGetNotification(const char * queueName)
```

Parameters

- **queueName** The name of the queue.

Returns

The number of affected queues. (not the number of blocked reads necessarily)

ChangeEncryptionKey method

Changes the database encryption key for an UltraLite database.

Syntax

```
public virtual bool ChangeEncryptionKey(const char * newKey)
```

Parameters

- **newKey** The new encryption key for the database.

Returns

True on success; otherwise, returns false.

Remarks

Applications that call this method 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 the ChangeEncryptionKey method 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.

Checkpoint method

Performs a checkpoint operation, flushing any pending committed transactions to the database.

Syntax

```
public virtual bool Checkpoint()
```

Returns

True on success; otherwise, returns false.

Remarks

Any current transaction is not committed by calling the Checkpoint method. This method is used in conjunction with deferring automatic transaction checkpoints (using the **commit_flush** connection parameter) as a performance enhancement.

The Checkpoint method ensures that all pending committed transactions have been written to the database.

Close method

Destroys this connection and any remaining associated objects.

Syntax

```
public virtual void Close(ULError * error)
```

Parameters

- **error** An optional ULError object to receive error information.

Commit method

Commits the current transaction.

Syntax

```
public virtual bool Commit()
```

Returns

True on success; otherwise, returns false.

CountUploadRows method

Counts the number of rows that need to be uploaded for synchronization.

Syntax

```
public virtual ul_u_long CountUploadRows(
     const char * pubList,
    ul_u_long threshold
)
```

Parameters

- **pubList** A string containing a comma-separated list of publications to check. An empty string (the UL_SYNC_ALL macro) implies all tables except tables marked as "no sync". A string containing just an asterisk (the 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 this value 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) and 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 method to prompt users to synchronize, or determine when automatic background synchronization should take place.

The following call checks the entire database for the total number of rows to be synchronized:

```
count = conn->CountUploadRows( UL_SYNC_ALL, 0 );
```

The following call checks publications PUB1 and PUB2 for a maximum of 1000 rows:

```
count = conn->CountUploadRows( "PUB1,PUB2", 1000 );
```

The following call checks to see if any rows need to be synchronized in publications PUB1 and PUB2:

```
anyToSync = conn->CountUploadRows( "PUB1,PUB2", 1 ) != 0;
```


CreateNotificationQueue method

Creates an event notification queue for this connection.

Syntax

```
public virtual bool CreateNotificationQueue(
     const char * name,
    const char * parameters
)
```

Parameters

- **name** The name for the new queue.
- **parameters** Reserved. Set to NULL.

Returns

True on success; otherwise, returns false.

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 the RegisterForEvent method if no queue is specified. This call fails with an error if the name already exists or isn't valid.

See also

- "ULConnection.RegisterForEvent method [UltraLite C++]" on page 137

DeclareEvent method

Declares an event which can then be registered for and triggered.

Syntax

```
public virtual bool DeclareEvent(const char * eventName)
```

Parameters

- **eventName** The name for the new user-defined event.

Returns

True if the event was declared successfully; otherwise, returns 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 method declares user-defined events. User-defined events are triggered with the TriggerEvent method. The event name must be unique. Names are case insensitive.

See also

- "ULConnection.TriggerEvent method [UltraLite C++]" on page 144

DestroyNotificationQueue method

Destroys the given event notification queue.

Syntax

```
public virtual bool DestroyNotificationQueue(const char * name)
```

Parameters

- **name** The name of the queue to destroy.

Returns

True on success; otherwise, returns false.

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.

ExecuteScalar method

Executes a SQL SELECT statement directly, returning a single result.

Syntax

```
public virtual bool ExecuteScalar(
     void * dstPtr,
    size_t dstSize,
    ul_column_storage_type dstType,
    const char * sql,
    ...
)
```

Parameters

- **dstPtr** A pointer to a variable of the required type to receive the value.
- **dstSize** The size of variable to receive value, if applicable.
- **dstType** The type of value to retrieve. This value must match the variable type.

- **sql** The SELECT statement, optionally containing '?' parameters.
- **...** String (char *) parameter values to substitute.

Returns

True if the query is successfully executed and a value is successfully retrieved; otherwise, returns false when a value is not fetched. Check the SQLCODE error code to determine why false is returned. The selected value is NULL if no warning or error (SQLE_NOERROR) is indicated.

Remarks

The dstPtr value must point to a variable of the correct type, matching the dstType value. The dstSize parameter is only required for variable-sized values, such as strings and binaries, and is otherwise ignored. The variable list of parameter values must correspond to parameters in the statement, and all values are assumed to be strings. (internally, UltraLite casts the parameter values as required for the statement)

The following types are supported:

- **UL_TYPE_BIT/UL_TYPE_TINY** Use variable type ul_byte (8 bit, unsigned).
- **UL_TYPE_U_SHORT/UL_TYPE_S_SHORT** Use variable type ul_u_short/ul_s_short (16 bit).
- **UL_TYPE_U_LONG/UL_TYPE_S_LONG** Use variable type ul_u_long/ul_s_long (32 bit).
- **UL_TYPE_U_BIG/UL_TYPE_S_BIG** Use variable type ul_u_big/ul_s_big (64 bit).
- **UL_TYPE_DOUBLE** Use variable type ul_double (double).
- **UL_TYPE_REAL** Use variable type ul_real (float).
- **UL_TYPE_BINARY** Use variable type ul_binary and specify **dstSize** (as in GetBinary()).
- **UL_TYPE_TIMESTAMP_STRUCT** Use variable type DECL_DATETIME.
- **UL_TYPE_CHAR** Use variable type char [] (a character buffer), and set **dstSize** to the size of the buffer (as in GetString()).
- **UL_TYPE_WCHAR** Use variable type ul_wchar [] (a wide character buffer), and set **dstSize** to the size of the buffer (as in GetString()).
- **UL_TYPE_TCHAR** Same as UL_TYPE_CHAR or UL_TYPE_WCHAR, depending on which version of the method is called.

The following example demonstrates integer fetching:

```
ul_u_long    val;
ok = conn->ExecuteScalar( &val, 0, UL_TYPE_U_LONG,
    "SELECT count(*) FROM t WHERE col LIKE ?", "ABC%" );
```

The following example demonstrates string fetching:

```
char    val[40];
ok = conn->ExecuteScalar( &val, sizeof(val), UL_TYPE_CHAR,
    "SELECT uuidtostr( newid() )" );
```

See also

- "ul_column_storage_type enumeration [UltraLite C and Embedded SQL datatypes]" on page 104

ExecuteScalarV method

Executes a SQL SELECT statement string, along with a list of substitution values.

Syntax

```
public virtual bool ExecuteScalarV(
     void * dstPtr,
    size_t dstSize,
    ul_column_storage_type dstType,
    const char * sql,
    va_list args
)
```

Parameters

- **dstPtr** A pointer to a variable of the required type to receive the value.
- **dstSize** The size of variable to receive value, if applicable.
- **dstType** The type of value to retrieve. This value must match the variable type.
- **sql** The SELECT statement, optionally containing '?' parameters.
- **args** A list of string (char *) values to substitute.

Returns

True if the query is successfully executed and a value is successfully retrieved; otherwise, returns false when a value is not fetched. Check the SQLCODE error code to determine why false is returned. The selected value is NULL if no warning or error (SQLE_NOERROR) is indicated.

Remarks

The dstPtr value must point to a variable of the correct type, matching the dstType value. The dstSize parameter is only required for variable-sized values, such as strings and binaries, and is otherwise ignored. The variable list of parameter values must correspond to parameters in the statement, and all values are assumed to be strings. (internally, UltraLite casts the parameter values as required for the statement)

The following types are supported:

- **UL_TYPE_BIT/UL_TYPE_TINY** Use variable type ul_byte (8 bit, unsigned).
- **UL_TYPE_U_SHORT/UL_TYPE_S_SHORT** Use variable type ul_u_short/ul_s_short (16 bit).
- **UL_TYPE_U_LONG/UL_TYPE_S_LONG** Use variable type ul_u_long/ul_s_long (32 bit).

- **UL_TYPE_U_BIG/UL_TYPE_S_BIG** Use variable type ul_u_big/ul_s_big (64 bit).
- **UL_TYPE_DOUBLE** Use variable type ul_double (double).
- **UL_TYPE_REAL** Use variable type ul_real (float).
- **UL_TYPE_BINARY** Use variable type ul_binary and specify **dstSize** (as in GetBinary()).
- **UL_TYPE_TIMESTAMP_STRUCT** Use variable type DECL_DATETIME.
- **UL_TYPE_CHAR** Use variable type char [] (a character buffer), and set **dstSize** to the size of the buffer (as in GetString()).
- **UL_TYPE_WCHAR** Use variable type ul_wchar [] (a wide character buffer), and set **dstSize** to the size of the buffer (as in GetString()).
- **UL_TYPE_TCHAR** Same as UL_TYPE_CHAR or UL_TYPE_WCHAR, depending on which version of the method is called.

See also

- "ul_column_storage_type enumeration [UltraLite C and Embedded SQL datatypes]" on page 104

ExecuteStatement method

Executes a SQL statement string directly.

Syntax

```
public virtual bool ExecuteStatement(const char * sql)
```

Parameters

- **sql** The SQL script to execute.

Returns

True on success; otherwise, returns false.

Remarks

Use this method to execute a SELECT statement directly and retrieve a single result.

Use the PrepareStatement method to execute a statement repeatedly with variable parameters, or to fetch multiple results.

See also

- "ULConnection.PrepareStatement method [UltraLite C++]" on page 136

GetChildObjectCount method

Gets the number of currently open child objects on the connection.

Syntax

```
public virtual ul_u_long GetChildObjectCount()
```

Returns

The number of currently open child objects.

Remarks

This method can be used to detect object leaks.

GetDatabaseProperty method

Obtains the value of a database property.

Syntax

```
public virtual const char * GetDatabaseProperty(const char * propName)
```

Parameters

- **propName** The name of the property being requested.

Returns

A pointer to a string buffer containing the database property value is returned when run successfully; otherwise, returns NULL.

Remarks

The returned value points to a static buffer whose contents may be changed by any subsequent UltraLite call, so you must make a copy of the value if you need to save it.

See also

- "Reading database properties" [*UltraLite - Database Management and Reference*]

Example

The following example illustrates how to get the value of the CharSet database property.

```
const char * charset = GetDatabaseProperty( "CharSet" );
```

GetDatabasePropertyInt method

Obtains the integer value of a database property.

Syntax

```
public virtual ul_u_long GetDatabasePropertyInt(const char * propName)
```

Parameters

- **propName** The name of the property being requested.

Returns

If successful, the integer value of the property; otherwise, returns 0.

See also

- "Reading database properties" [*UltraLite - Database Management and Reference*]

Example

The following example illustrates how to get the value of the ConnCount database property.

```
unsigned connectionCount = GetDatabasePropertyInt( "ConnCount" );
```

GetDatabaseSchema method

Returns an object pointer used to query the schema of the database.

Syntax

```
public virtual ULDatabaseSchema * GetDatabaseSchema()
```

Returns

A ULDatabaseSchema object used to query the schema of the database.

GetLastDownloadTime method

Obtains the last time a specified publication was downloaded.

Syntax

```
public virtual bool GetLastDownloadTime(
     const char * publication,
    DECL_DATETIME * value
)
```

Parameters

- **publication** The publication name.
- **value** A pointer to the DECL_DATETIME structure to be populated. The value of January 1, 1900 indicates that the publication has yet to be synchronized, or the time was reset.

Returns

True when the value is successfully populated by the last download time of the publication specified; otherwise, returns false.

Remarks

The following call populates the dt structure with the date and time that the 'pub1' publication was downloaded:

```
DECL_DATETIME dt;
ok = conn->GetLastDownloadTime( "pub1", &dt );
```

GetLastError method

Returns the error information associated with the last call.

Syntax

```
public virtual const ULError * GetLastError()
```

Returns

A pointer to the ULError object with information associated with the last call.

Remarks

The error object whose address is returned remains valid while the connection is open, but not updated automatically on subsequent calls. You must call GetLastError to retrieve updated status information.

See also

- "ULError class [UltraLite C++]" on page 159

GetLastIdentity method

Gets the @@identity value.

Syntax

```
public virtual ul_u_big GetLastIdentity()
```

Returns

The last value inserted into an autoincrement or global autoincrement column

Remarks

This value is the last value inserted into an autoincrement or global autoincrement column for the database. This value is not recorded when the database is shutdown, so calling this method before any autoincrement values have been inserted returns 0.

Note
The last value inserted may have been on another connection.

GetNotification method

Reads an event notification.

Syntax

```
public virtual const char * GetNotification(
     const char * queueName,
    ul_u_long waitms
)
```

Parameters

- **queueName** The queue to read or NULL for the default connection queue.
- **waitms** The time, in milliseconds to wait (block) before returning.

Returns

The name of the event read or NULL on error.

Remarks

This call blocks until a notification is received or until the given wait period expires. To wait indefinitely, set the waitms parameter to UL_READ_WAIT_INFINITE. To cancel a wait, send another notification to the given queue or use the CancelGetNotification method. Use the GetNotificationParameter method after reading a notification to retrieve additional parameters by name.

See also

- "ULConnection.CancelGetNotification method [UltraLite C++]" on page 122
- "ULConnection.GetNotificationParameter method [UltraLite C++]" on page 133

GetNotificationParameter method

Gets a parameter for the event notification just read by the GetNotification method.

Syntax

```
public virtual const char * GetNotificationParameter(
     const char * queueName,
    const char * parameterName
)
```

Parameters

- **queueName** The queue to read or NULL for default connection queue.

- **parameterName** The name of the parameter to read (or "*").

Returns

The parameter value or NULL 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

- "ULConnection.GetNotification method [UltraLite C++]" on page 133

GetSqlca method

Gets the communication area associated with this connection.

Syntax

```
public virtual SQLCA * GetSqlca()
```

Returns

A pointer to the SQLCA object for this connection.

GetSyncResult method

Gets the result of the last synchronization.

Syntax

```
public virtual bool GetSyncResult(ul_sync_result * syncResult)
```

Parameters

- **syncResult** A pointer to the ul_sync_result structure to be populated.

Returns

True on success, otherwise false.

See also

- "ul_sync_result structure [UltraLite C and Embedded SQL datatypes]" on page 112

GetUserPointer method

Gets the pointer value last set by the SetUserPointer method.

Syntax

```
public virtual void * GetUserPointer()
```

See also

- "ULConnection.SetUserPointer method [UltraLite C++]" on page 141

GlobalAutoincUsage method

Obtains the percent of the default values used in all the columns that have global autoincrement defaults.

Syntax

```
public virtual ul_u_short GlobalAutoincUsage()
```

Returns

The percent of the global autoincrement values used by the counter.

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.

GrantConnectTo method

Grants access to an UltraLite database for a new or existing user ID with the given password.

Syntax

```
public virtual bool GrantConnectTo(const char * uid, const char * pwd)
```

Parameters

- **uid** A character array that holds the user ID. The maximum length is 31 characters.
- **pwd** A character array that holds the password for the user ID.

Returns

True on success; otherwise, returns false.

Remarks

This method updates the password for an existing user when you specify an existing user ID.

See also

- "ULConnection.RevokeConnectFrom method [UltraLite C++]" on page 138

InitSyncInfo method

Initializes the synchronization information structure.

Syntax

```
public virtual void InitSyncInfo(ul_sync_info * info)
```

Parameters

- **info** A pointer to the ul_sync_info structure that holds the synchronization parameters.

Remarks

Call this method before setting the values of fields in the ul_sync_info structure.

OpenTable method

Opens a table.

Syntax

```
public virtual ULTable * OpenTable(
     const char * tableName,
    const char * indexName
)
```

Parameters

- **tableName** The name of the table to open.
- **indexName** The name of the index to open the table on. Pass NULL to open on the primary key and the empty string to open the table unordered.

Returns

The ULTable object when the call is successful; otherwise, returns NULL.

Remarks

The cursor position is set before the first row when the application first opens a table.

PrepareStatement method

Prepares a SQL statement.

Syntax

```
public virtual ULPreparedStatement * PrepareStatement(const char * sql)
```

Parameters

- **sql** The SQL statement to prepare.

Returns

The ULPreparedStatement object on success; otherwise, returns NULL.

RegisterForEvent method

Registers or unregisters a queue to receive notifications of an event.

Syntax

```
public virtual bool RegisterForEvent(
     const char * eventName,
    const char * objectName,
    const char * queueName,
    bool register_not_unreg
)
```

Parameters

- **eventName** The system- or user-defined event to register for.
- **objectName** The object to which the event applies. (for example, a table name).
- **queueName** NULL means use the default connection queue.
- **register_not_unreg** Set true to register, or false to unregister.

Returns

True if the registration succeeded; otherwise, returns 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 you to specify an object name to which the event applies. For example, the TableModified event can specify the table name. Unlike the SendNotification method, only the specific queue registered receives notifications of the event. Other queues with the same name on different connections do not receive notifications, unless they are also explicitly 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.

ResetLastDownloadTime method

Resets the last download time of a publication so that the application resynchronizes previously downloaded data.

Syntax

```
public virtual bool ResetLastDownloadTime(const char * pubList)
```

Parameters

- **pubList** A string containing a comma-separated list of publications to reset. An empty string means all tables except tables marked as "no sync". A string containing just an asterisk ("*") denotes all publications. Some tables may not be part of any publication and are not included if this value is "*".

Returns

True on success; otherwise, returns false.

Remarks

The following method call resets the last download time for all tables:

```
conn->ResetLastDownloadTime( "" );
```

RevokeConnectFrom method

Revokes access from an UltraLite database for a user ID.

Syntax

```
public virtual bool RevokeConnectFrom(const char * uid)
```

Parameters

- **uid** A character array holding the user ID to be excluded from database access.

Returns

True on success, otherwise false.

Rollback method

Rolls back the current transaction.

Syntax

```
public virtual bool Rollback()
```

Returns

True on success, otherwise false.

RollbackPartialDownload method

Rolls back the changes from a failed synchronization.

Syntax

```
public virtual bool RollbackPartialDownload()
```

Returns

True on success, otherwise false.

Remarks

When using resumable downloads (synchronizing with the keep-partial-download option turned on), and a communication error occurs during the download phase of synchronization, UltraLite retains the changes which were downloaded (so the synchronization can resume from the place it was interrupted). Use this method to discard this partial download when you no longer wish to attempt resuming.

This method has effect only when using resumable downloads.

SendNotification method

Sends a notification to all queues matching the given name.

Syntax

```
public virtual ul_u_long SendNotification(
     const char * queueName,
    const char * eventName,
    const char * parameters
)
```

Parameters

- **queueName** The target queue name (or "*").
- **eventName** The identity for notification.
- **parameters** Optional parameters option list.

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** value specifies a semicolon delimited name=value pairs option list. After the notification is read, the parameter values are read with the GetNotificationParameter method.

See also

- "ULConnection.GetNotificationParameter method [UltraLite C++]" on page 133

SetDatabaseOption method

Sets the specified database option.

Syntax

```
public virtual bool SetDatabaseOption(
     const char * optName,
    const char * value
)
```

Parameters

- **optName** The name of the option being set.
- **value** The new value of the option.

Returns

True on success, otherwise false.

See also

- "UltraLite database options" [*UltraLite - Database Management and Reference*]

SetDatabaseOptionInt method

Sets a database option.

Syntax

```
public virtual bool SetDatabaseOptionInt(
     const char * optName,
    ul_u_long value
)
```

Parameters

- **optName** The name of the option being set.
- **value** The new value of the option.

Returns

True on success; otherwise, returns false.

SetSynchronizationCallback method

Sets the callback to be invoked while performing a synchronization.

Syntax

```
public virtual void SetSynchronizationCallback(
     ul_sync_observer_fn callback,
    void * userData
)
```

Parameters

- **callback** The ul_sync_observer_fn callback.
- **userData** User context information passed to the callback.

SetSyncInfo method

Creates a synchronization profile using the given name based on the given ul_sync_info structure.

Syntax

```
public virtual bool SetSyncInfo(
     char const * profileName,
    ul_sync_info * info
)
```

Parameters

- **profileName** The name of the synchronization profile.
- **info** A pointer to the ul_sync_info structure that holds the synchronization parameters.

Returns

True on success; otherwise, returns false.

Remarks

The synchronization profile replaces any previous profile with the same name. The named profile is deleted by specifying a null pointer for the structure.

SetUserPointer method

Sets an arbitrary pointer value in the connection for use by the calling application.

Syntax

```
public virtual void * SetUserPointer(void * ptr)
```

Returns

The previously set pointer value.

Remarks

This can be used to associate application data with the connection.

StartSynchronizationDelete method

Sets START SYNCHRONIZATION DELETE for this connection.

Syntax

```
public virtual bool StartSynchronizationDelete()
```

Returns

True on success, otherwise false.

StopSynchronizationDelete method

Sets STOP SYNCHRONIZATION DELETE for this connection.

Syntax

```
public virtual bool StopSynchronizationDelete()
```

Returns

True on success, otherwise false.

Synchronize method

Initiates synchronization in an UltraLite application.

Syntax

```
public virtual bool Synchronize(ul_sync_info * info)
```

Parameters

- **info** A pointer to the ul_sync_info structure that holds the synchronization parameters.

Returns

True on success; otherwise, returns false.

Remarks

This method initiates synchronization with the MobiLink server. This method does not return until synchronization is complete, however additional threads on separate connections may continue to access the database during synchronization.

Before calling this method, enable the protocol and encryption you are using with methods in the ULDatabaseManager class. For example, when using "HTTP", call the ULDatabaseManager.EnableHttpSynchronization method.

```
ul_sync_info info;
conn->InitSyncInfo( &info );
info.user_name = "my_user";
info.version = "myapp_1_2";
info.stream = "HTTP";
info.stream_parms = "host=myserver.com";
conn->Synchronize( &info );
```

See also

- "ULDatabaseManager.EnableHttpSynchronization method [UltraLite C++]" on page 150
- "MobiLink client network protocol options" [*MobiLink - Client Administration*]

SynchronizeFromProfile method

Synchronizes the database using the given profile and merge parameters.

Syntax

```
public virtual bool SynchronizeFromProfile(
     const char * profileName,
    const char * mergeParms,
    ul_sync_observer_fn observer,
    void * userData
)
```

Parameters

- **profileName** The name of the profile to synchronize.
- **mergeParms** Merge parameters for the synchronization.
- **observer** The observer callback to send status updates to.
- **userData** User context data passed to callback.

Returns

True on success; otherwise, returns false.

Remarks

This method is identical to executing the SYNCHRONIZE statement.

See also

- "ULConnection.Synchronize method [UltraLite C++]" on page 142
- "SYNCHRONIZE statement [UltraLite]" [*UltraLite - Database Management and Reference*]

TriggerEvent method

Triggers a user-defined event and sends notifications to all registered queues.

Syntax

```
public virtual ul_u_long TriggerEvent(
     const char * eventName,
    const char * parameters
)
```

Parameters

- **eventName** The name of the system or user-defined event to trigger.
- **parameters** Optional parameters option list.

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

- "ULConnection.GetNotificationParameter method [UltraLite C++]" on page 133

ValidateDatabase method

Validates the database on this connection.

Syntax

```
public virtual bool ValidateDatabase(
     ul_u_short flags,
    ul_validate_callback_fn fn,
    void * user_data,
    const char * tableName
)
```

Parameters

- **flags** Flags controlling the type of validation. See the example below.
- **fn** Function to receive validation progress information.

- **user_data** User data to send back to the caller via the callback.
- **tableName** Optional. A specific table to validate.

Returns

True on success; otherwise, returns false.

Remarks

Tables, indexes, and database pages can be validated depending on the flags passed to this routine. 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.

The flags parameter is combination of the following values:

- ULVF_TABLE
- ULVF_INDEX
- ULVF_DATABASE
- ULVF_EXPRESS
- ULVF_FULL_VALIDATE

See also

- “ULVF_TABLE variable [UltraLite C and Embedded SQL datatypes]” on page 117
- “ULVF_INDEX variable [UltraLite C and Embedded SQL datatypes]” on page 117
- “ULVF_DATABASE variable [UltraLite C and Embedded SQL datatypes]” on page 115
- “ULVF_EXPRESS variable [UltraLite C and Embedded SQL datatypes]” on page 115
- “ULVF_FULL_VALIDATE variable [UltraLite C and Embedded SQL datatypes]” on page 116

Example

```
The following example demonstrates table and index validation in express
mode:
flags = ULVF_TABLE | ULVF_INDEX | ULVF_EXPRESS;
```

ULDatabaseManager class

Manages connections and databases.

Syntax

```
public class ULDatabaseManager
```

Members

All members of ULDatabaseManager class, including all inherited members.

Name	Description
CreateDatabase method	Creates a new database.
DropDatabase method	Erases an existing database that is not currently running.
EnableAesDBEncryption method	Enables AES database encryption.
EnableAesFipsDBEncryption method	Enables FIPS 140-2 certified AES database encryption.
EnableEccE2ee method	Enables ECC end-to-end encryption.
EnableEccSyncEncryption method	Enables ECC synchronization encryption for SSL or TLS streams.
EnableHttpsSynchronization method	Enables HTTPS synchronization.
EnableHttpSynchronization method	Enables HTTP synchronization.
EnableRsaE2ee method	Enables RSA end-to-end encryption.
EnableRsaFipsE2ee method	Enables FIPS 140-2 certified RSA end-to-end encryption.
EnableRsaFipsSyncEncryption method	Enables FIPS 140-2 certified RSA synchronization encryption for SSL or TLS streams.
EnableRsaSyncEncryption method	Enables RSA synchronization encryption.
EnableTcpipSynchronization method	Enables TCP/IP synchronization.
EnableTlsSynchronization method	Enables TLS synchronization.
EnableZlibSyncCompression method	Enables Zlib compression for a synchronization stream.
Fini method	Finalizes the UltraLite runtime.
Init method	Initializes the UltraLite runtime.
OpenConnection method	Opens a new connection to an existing database.
SetErrorCallback method	Sets the callback to be invoked when an error occurs.
ValidateDatabase method	Performs low level and index validation on a database.

Remarks

The Init method must be called in a thread-safe environment before any other calls can be made. The Fini method must be called in a similarly thread-safe environment when finished.

Note
This class is static. Do not create an instance of it.

CreateDatabase method

Creates a new database.

Syntax

```
public static ULConnection * CreateDatabase(
     const char * connParms,
    const char * createParms,
    ULError * error
)
```

Parameters

- **connParms** 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 can be specified when you connect to a database.
- **createParms** A semicolon separated string of database creation parameters, which are set as keyword value pairs. For example: page_size=2048;obfuscate=yes.
- **error** An optional ULError object to receive error information.

Returns

A ULConnection object to the new database is returned if the database was created successfully. NULL is returned if the method fails. Failure is usually caused by an invalid file name or denied access.

Remarks

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

The connParms parameter is a set of standard connection parameters that are applicable whenever the database is accessed, such as the file name or the encryption key.

The createParms parameter is a set of parameters that are only relevant when creating a database, such as checksum-level, page-size, collation, and time and date format.

The following code illustrates how to use the CreateDatabase method to create an UltraLite database as the file *mydb.udb*:

```
ULConnection * conn;
conn = ULDatabaseManager::CreateDatabase( "DBF=mydb.udb",
"checksum_level=2" );
if( conn != NULL ) {
    // success
} else {
    // unable to create
}
```


See also

- "UltraLite connection parameters" [*UltraLite - Database Management and Reference*]
- "UltraLite creation parameters" [*UltraLite - Database Management and Reference*]

DropDatabase method

Erases an existing database that is not currently running.

Syntax

```
public static bool DropDatabase(const char * parms, ULError * error)
```

Parameters

- **parms** The database identification parameters. (a connection string)
- **error** An optional ULError object to receive error information.

Returns

True if the database was successfully deleted; otherwise, returns false.

EnableAesDBEncryption method

Enables AES database encryption.

Syntax

```
public static void EnableAesDBEncryption()
```

Remarks

Call this method to use AES database encryption. Use the DBKEY connection parameter to specify the encryption passphrase. You must call this method before opening the database connection.

See also

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

EnableAesFipsDBEncryption method

Enables FIPS 140-2 certified AES database encryption.

Syntax

```
public static void EnableAesFipsDBEncryption()
```


Remarks

Call this method to use FIPS AES database encryption. Use the DBKEY connection parameter to specify the encryption passphrase.

You must specify 'fips=yes' in the database creation parameters string. You must call this method before opening the database connection.

Note
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 12 - Introduction*].

See also

- "ULDatabaseManager.EnableAesDBEncryption method [UltraLite C++]" on page 148
- "UltraLite DBKEY connection parameter" [*UltraLite - Database Management and Reference*]

EnableEccE2ee method

Enables ECC end-to-end encryption.

Syntax

```
public static void EnableEccE2ee()
```

Remarks

You must call this method before the Synchronize method.

To use end-to-end encryption, set the **e2ee_public_key** network protocol option. In this case, the **e2ee_type** network protocol option must be "ECC".

See also

- "MobiLink client network protocol options" [*MobiLink - Client Administration*]

EnableEccSyncEncryption method

Enables ECC synchronization encryption for SSL or TLS streams.

Syntax

```
public static void EnableEccSyncEncryption()
```

Remarks

You must call this method before the Synchronize method.

This method is required when you set the **stream** parameter to "TLS" or "HTTPS" for ECC encryption. In this case, you must also set the **tls_type** network protocol option to "ECC".

Note
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 12 - Introduction*].

See also

- "ULDatabaseManager.EnableZlibSyncCompression method [UltraLite C++]" on page 153
- "ULDatabaseManager.EnableRsaFipsSyncEncryption method [UltraLite C++]" on page 151
- "MobiLink client network protocol options" [*MobiLink - Client Administration*]

EnableHttpsSynchronization method

Enables HTTPS synchronization.

Syntax

```
public static void EnableHttpsSynchronization()
```

Remarks

You must call this method before the Synchronize method.

When initiating synchronization, set the **stream** parameter to "HTTPS". Also set the network protocol certificate options.

See also

- "MobiLink client network protocol options" [*MobiLink - Client Administration*]

EnableHttpSynchronization method

Enables HTTP synchronization.

Syntax

```
public static void EnableHttpSynchronization()
```

Remarks

You must call this method before the Synchronize method.

When initiating synchronization, set the **stream** parameter to "HTTP".

See also

- "MobiLink client network protocol options" [*MobiLink - Client Administration*]

EnableRsaE2ee method

Enables RSA end-to-end encryption.

Syntax

```
public static void EnableRsaE2ee()
```

Remarks

You must call this method before the Synchronize method.

To use end-to-end encryption, set the **e2ee_public_key** network protocol option. In this case, the network protocol option **e2ee_type** must be "RSA" (the default).

See also

- "MobiLink client network protocol options" [*MobiLink - Client Administration*]

EnableRsaFipsE2ee method

Enables FIPS 140-2 certified RSA end-to-end encryption.

Syntax

```
public static void EnableRsaFipsE2ee()
```

Remarks

You must call this method before the Synchronize method.

To use end-to-end encryption, set the **e2ee_public_key** network protocol option. In this case, the **e2ee_type** network protocol option must be "RSA" (the default), and **fips** must be set to "yes".

See also

- "MobiLink client network protocol options" [*MobiLink - Client Administration*]

EnableRsaFipsSyncEncryption method

Enables FIPS 140-2 certified RSA synchronization encryption for SSL or TLS streams.

Syntax

```
public static void EnableRsaFipsSyncEncryption()
```

Remarks

You must call this method before the Synchronize method.

This is required when setting the **stream** parameter to "TLS" or "HTTPS" for FIPS RSA encryption. In this case, the **tls_type** network protocol option must be set to "RSA" (the default), and **fips** must be set to "yes".

See also

- "ULDatabaseManager.EnableRsaSyncEncryption method [UltraLite C++]" on page 152
- "ULDatabaseManager.EnableEccSyncEncryption method [UltraLite C++]" on page 149
- "MobiLink client network protocol options" [*MobiLink - Client Administration*]

EnableRsaSyncEncryption method

Enables RSA synchronization encryption.

Syntax

```
public static void EnableRsaSyncEncryption()
```

Remarks

You must call this method before the Synchronize method.

This is required when setting the **stream** parameter to "TLS" or "HTTPS" for RSA encryption. In this case, the network protocol option **tls_type** must be "RSA" (the default).

See also

- "MobiLink client network protocol options" [*MobiLink - Client Administration*]

EnableTcpipSynchronization method

Enables TCP/IP synchronization.

Syntax

```
public static void EnableTcpipSynchronization()
```

Remarks

You must call this method before the Synchronize method.

When initiating synchronization, set the **stream** parameter to "TCPIP".

See also

- "MobiLink client network protocol options" [*MobiLink - Client Administration*]

EnableTlsSynchronization method

Enables TLS synchronization.

Syntax

```
public static void EnableTlsSynchronization()
```

Remarks

You must call this method before the Synchronize method.

When initiating synchronization, set the **stream** parameter to "TLS". Also set the network protocol certificate options.

See also

- "MobiLink client network protocol options" [*MobiLink - Client Administration*]

EnableZlibSyncCompression method

Enables Zlib compression for a synchronization stream.

Syntax

```
public static void EnableZlibSyncCompression()
```

Remarks

You must call this method before the Synchronize method.

To use compression, set the **compression** network protocol option to "zlib".

See also

- "MobiLink client network protocol options" [*MobiLink - Client Administration*]

Fini method

Finalizes the UltraLite runtime.

Syntax

```
public static void Fini()
```

Remarks

This method must be called only once by a single thread when the application is finished. This method is not thread-safe.

Init method

Initializes the UltraLite runtime.

Syntax

```
public static bool Init()
```

Returns

True on success; otherwise, returns false. False can also be returned if the method is called more than once.

Remarks

This method must be called only once by a single thread before any other calls can be made. This method is not thread-safe.

This method does not usually fail unless memory is unavailable.

OpenConnection method

Opens a new connection to an existing database.

Syntax

```
public static ULConnection * OpenConnection(
     const char * connParms,
    ULError * error,
    void * reserved
)
```

Parameters

- **connParms** The connection string.
- **error** An optional ULError object to return error information.
- **reserved** Reserved for internal use. Omit or set to null.

Returns

A new ULConnection object if the method succeeds; otherwise, returns NULL.

Remarks

The connection string is a set of option=value connection parameters (semicolon separated) that indicates which database to connect to, and options to use for the connection. For example, after securely obtaining your encryption passphrase, the resulting connection string might be:
"DBF=mydb.udb;DBKEY=iyntTZld9OEa#&G".

To get error information, pass in a pointer to a ULError object. The following is a list of possible errors:

- **SQLE_INVALID_PARSE_PARAMETER** **connParms** was not formatted properly.

- **SQLE_UNRECOGNIZED_OPTION** A connection option name was likely misspelled.
- **SQLE_INVALID_OPTION_VALUE** A connection option value was not specified properly.
- **SQLE_ULTRALITE_DATABASE_NOT_FOUND** The specified database could not be found.
- **SQLE_INVALID_LOGON** You supplied an invalid user ID or an incorrect password.
- **SQLE_TOO_MANY_CONNECTIONS** You exceeded the maximum number of concurrent database connections.

See also

- "UltraLite connection strings and parameters" [*UltraLite - Database Management and Reference*]
- "UltraLite connection parameters" [*UltraLite - Database Management and Reference*]

SetErrorCallback method

Sets the callback to be invoked when an error occurs.

Syntax

```
public static void SetErrorCallback(
     ul_cpp_error_callback_fn callback,
    void * userData
)
```

Parameters

- **callback** The callback function.
- **userData** User context information passed to the callback.

Remarks

This method is not thread-safe.

ValidateDatabase method

Performs low level and index validation on a database.

Syntax

```
public static bool ValidateDatabase(
     const char * connParms,
    ul_u_short flags,
    ul_validate_callback_fn fn,
    void * userData,
    ULError * error
)
```

Parameters

- **connParms** The parameters used to connect to the database.
- **flags** The flags controlling the type of validation; see the example below.
- **fn** A function to receive validation progress information.
- **userData** The user data to send back to the caller via the callback.
- **error** An optional ULError object to receive error information.

Returns

True if the validation succeeds; otherwise, returns false.

Remarks

The flags parameter is combination of the following values:

- ULVF_TABLE
- ULVF_INDEX
- ULVF_DATABASE
- ULVF_EXPRESS
- ULVF_FULL_VALIDATE

See also

- "ULVF_TABLE variable [UltraLite C and Embedded SQL datatypes]" on page 117
- "ULVF_INDEX variable [UltraLite C and Embedded SQL datatypes]" on page 117
- "ULVF_DATABASE variable [UltraLite C and Embedded SQL datatypes]" on page 115
- "ULVF_EXPRESS variable [UltraLite C and Embedded SQL datatypes]" on page 115
- "ULVF_FULL_VALIDATE variable [UltraLite C and Embedded SQL datatypes]" on page 116

Example

The following example demonstrates table and index validation in express mode:

```
flags = ULVF_TABLE | ULVF_INDEX | ULVF_EXPRESS;
```

ULDatabaseSchema class

Represents the schema of an UltraLite database.

Syntax

```
public class ULDatabaseSchema
```


Members

All members of ULDatabaseSchema class, including all inherited members.

Name	Description
Close method	Destroys this object.
GetConnection method	Gets the ULConnection object.
GetNextPublication method	Gets the name of the next publication in the database.
GetNextTable method	Gets the next table (schema) in the database.
GetPublicationCount method	Gets the number of publications in the database.
GetTableCount method	Returns the number of tables in the database.
GetTableSchema method	Returns the schema of the named table.

Close method

Destroys this object.

Syntax

```
public virtual void Close()
```

GetConnection method

Gets the ULConnection object.

Syntax

```
public virtual ULConnection * GetConnection()
```

Returns

The ULConnection associated with this object.

GetNextPublication method

Gets the name of the next publication in the database.

Syntax

```
public virtual const char * GetNextPublication(
     ul_publication_iter * iter
)
```

Parameters

- **iter** A pointer to the iterator variable.

Returns

The name of the next publication. This value points to a static buffer whose contents may be changed by any subsequent UltraLite call, so make a copy of the value if you need to retain it. NULL is returned when the iteration is complete.

Remarks

Initialize the iter value to the ul_publication_iter_start constant before the first call.

See also

- "ul_publication_iter_start variable [UltraLite C++]" on page 237

GetNextTable method

Gets the next table (schema) in the database.

Syntax

```
public virtual ULTableSchema * GetNextTable(ul_table_iter * iter)
```

Parameters

- **iter** A pointer to the iterator variable.

Returns

A ULTableSchema object or NULL when the iteration is complete.

Remarks

Initialize the iter value to the ul_table_iter_start constant before the first call.

See also

- "ul_table_iter_start variable [UltraLite C++]" on page 237

GetPublicationCount method

Gets the number of publications in the database.

Syntax

```
public virtual ul_publication_count GetPublicationCount()
```

Returns

The number of publications in the database.

Remarks

Publication IDs range from 1 to the number returned by this method.

GetTableCount method

Returns the number of tables in the database.

Syntax

```
public virtual ul_table_num GetTableCount()
```

Returns

An integer that represents the number of tables.

GetTableSchema method

Returns the schema of the named table.

Syntax

```
public virtual ULTableSchema * GetTableSchema(const char * tableName)
```

Parameters

- **tableName** The name of the table.

Returns

A ULTableSchema object for the given table; otherwise, returns UL_NULL if the table does not exist.

ULError class

Manages the errors returned from the UltraLite runtime.

Syntax

```
public class ULError
```

Members

All members of ULError class, including all inherited members.

Name	Description
ULError constructor	Constructs a ULError object.
Clear method	Clears the current error.

Name	Description
GetErrorInfo method	Returns a pointer to the underlying ul_error_info object.
GetParameter method	Copies the specified error parameter into the provided buffer.
GetParameterCount method	Returns the number of error parameters.
GetSQLCode method	Returns the SQLCODE error code for the last operation.
GetSQLCount method	Returns a value that depends on the last operation, and the result of that operation.
GetString method	Returns the description of the current error.
GetURL method	Returns a URL to the documentation page for this error.
IsOK method	Tests the error code.

ULError constructor

Constructs a ULError object.

Syntax

```
public ULError()
```

Clear method

Clears the current error.

Syntax

```
public void Clear()
```

Remarks

The current error is cleared automatically on most calls, so this is not normally called by applications.

GetErrorInfo method

Returns a pointer to the underlying ul_error_info object.

Overload list

Name	Description
GetErrorInfo() method	Returns a pointer to the underlying ul_error_info object.
GetErrorInfo() method	Returns a pointer to the underlying ul_error_info object.

GetErrorInfo() method

Returns a pointer to the underlying ul_error_info object.

Syntax

```
public const ul_error_info * GetErrorInfo()
```

Returns

A pointer to the underlying ul_error_info object.

See also

- "ul_error_info structure [UltraLite C and Embedded SQL datatypes]" on page 109

GetErrorInfo() method

Returns a pointer to the underlying ul_error_info object.

Syntax

```
public ul_error_info * GetErrorInfo()
```

Returns

A pointer to the underlying ul_error_info object.

See also

- "ul_error_info structure [UltraLite C and Embedded SQL datatypes]" on page 109

GetParameter method

Copies the specified error parameter into the provided buffer.

Syntax

```
public size_t GetParameter(ul_u_short parmNo, char * dst, size_t len)
```

Parameters

- **parmNo** A 1-based parameter number.

- **dst** The buffer to receive the parameter.
- **len** The size of the buffer.

Returns

The size required to store the parameter, or zero if the ordinal is not valid. The parameter is truncated if the return value is larger than the len value.

Remarks

The output string is always null-terminated, even when the buffer is too small and the string is truncated.

GetParameterCount method

Returns the number of error parameters.

Syntax

```
public ul_u_short GetParameterCount()
```

Returns

The number of error parameters.

GetSQLCode method

Returns the SQLCODE error code for the last operation.

Syntax

```
public an_sql_code GetSQLCode()
```

Returns

The sqlcode value.

GetSQLCount method

Returns a value that depends on the last operation, and the result of that operation.

Syntax

```
public ul_s_long GetSQLCount()
```

Returns

The value for the last operation, if applicable; otherwise, returns -1 if not applicable.

Remarks

The following list outlines the possible operations, and their returned results:

- **INSERT, UPDATE, or DELETE operation executed successfully** Returns the number of rows that were affected by the statement.
- **SQL statement syntax error (SQLE_SYNTAX_ERROR)** Returns the approximate character position within the statement where the error was detected.

GetString method

Returns the description of the current error.

Syntax

```
public size_t GetString(char * dst, size_t len)
```

Parameters

- **dst** The buffer to receive the error description.
- **len** The size, in array elements, of the buffer.

Returns

The size required to store the string. The string is truncated when the return value is larger than the len value.

Remarks

The string includes the error code and all parameters. A full description of the error can be obtained by loading the URL returned by the ULError.GetURL method.

The output string is always null-terminated, even if the buffer is too small and the string is truncated.

See also

- "ULError.GetURL method [UltraLite C++]" on page 163

GetURL method

Returns a URL to the documentation page for this error.

Syntax

```
public size_t GetURL(char * buffer, size_t len, const char * reserved)
```

Parameters

- **buffer** The buffer to receive the URL.
- **len** The size of the buffer.

- **reserved** Reserved for future use; you must pass NULL, the default.

Returns

The size required to store the URL. The URL is truncated if the return value is larger is larger than the len value.

IsOK method

Tests the error code.

Syntax

```
public bool IsOK()
```

Returns

True if the current code is SQLE_NOERROR or a warning; otherwise, returns false if the current code indicates an error.

ULIndexSchema class

Represents the schema of an UltraLite table index.

Syntax

```
public class ULIndexSchema
```

Members

All members of ULIndexSchema class, including all inherited members.

Name	Description
Close method	Destroys this object.
GetColumnCount method	Gets the number of columns in the index.
GetColumnName method	Gets the name of the column given the position of the column in the index.
GetConnection method	Gets the ULConnection object.
GetIndexColumnID method	Gets the 1-based index column ID from its name.
GetIndexFlags method	Gets the index property flags bit field.
GetName method	Gets the name of the index.

Name	Description
GetReferencedIndexName method	Gets the associated primary index name.
GetReferencedTableName method	Gets the associated primary table name.
GetTableName method	Gets the name of the table containing this index.
IsColumnDescending method	Determines if the column is in descending order.

Close method

Destroys this object.

Syntax

```
public virtual void Close()
```

GetColumnCount method

Gets the number of columns in the index.

Syntax

```
public virtual ul_column_num GetColumnCount()
```

Returns

The number of columns in the index.

GetColumnName method

Gets the name of the column given the position of the column in the index.

Syntax

```
public virtual const char * 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

The name of the column. This value points to a static buffer whose contents may be changed by any subsequent UltraLite call, so make a copy of the value if you need to retain it.

GetConnection method

Gets the ULConnection object.

Syntax

```
public virtual ULConnection * GetConnection()
```

Returns

The connection associated with this object.

GetIndexColumnID method

Gets the 1-based index column ID from its name.

Syntax

```
public virtual ul_column_num GetIndexColumnID(const char * columnName)
```

Parameters

- **columnName** The column name.

Returns

0, and sets SQLE_COLUMN_NOT_FOUND if the column name does not exist.

GetIndexFlags method

Gets the index property flags bit field.

Syntax

```
public virtual ul_index_flag GetIndexFlags()
```

See also

- "ul_index_flag enumeration [UltraLite C++]" on page 235

GetName method

Gets the name of the index.

Syntax

```
public virtual const char * GetName()
```


Returns

The name of the index. This value points to a static buffer whose contents may be changed by any subsequent UltraLite call, so make a copy of the value if you need to retain it.

GetReferencedIndexName method

Gets the associated primary index name.

Syntax

```
public virtual const char * GetReferencedIndexName()
```

Returns

The name of the referenced index. This value points to a static buffer whose contents may be changed by any subsequent UltraLite call, so make a copy of the value if you need to retain it.

Remarks

This method applies to foreign keys only.

GetReferencedTableName method

Gets the associated primary table name.

Syntax

```
public virtual const char * GetReferencedTableName()
```

Returns

The name of the referenced table. This value points to a static buffer whose contents may be changed by any subsequent UltraLite call, so make a copy of the value if you need to retain it.

Remarks

This method applies to foreign keys only.

GetTableName method

Gets the name of the table containing this index.

Syntax

```
public virtual const char * GetTableName()
```

Returns

The name of the table containing this index. This value points to a static buffer whose contents may be changed by any subsequent UltraLite call, so make a copy of the value if you need to retain it.

IsColumnDescending method

Determines if the column is in descending order.

Syntax

```
public virtual bool IsColumnDescending(ul_column_num cid)
```

Parameters

- **cid** The 1-based ordinal column number.

Returns

True if the column is in descending order; otherwise, returns false.

ULPreparedStatement class

Represents a prepared SQL statement.

Syntax

```
public class ULPreparedStatement
```

Members

All members of ULPreparedStatement class, including all inherited members.

Name	Description
AppendParameterByteChunk method	Sets a large binary parameter broken down into several chunks.
AppendParameterStringChunk method	Sets a large string parameter broken down into several chunks.
Close method	Destroys this object.
ExecuteQuery method	Executes a SQL SELECT statement as a query.
ExecuteStatement method	Executes a statement that does not return a result set, such as a SQL INSERT, DELETE or UPDATE statement.
GetConnection method	Gets the connection object.
GetParameterCount method	Gets the number of input parameters for this statement.
GetParameterID method	Get the 1-based ordinal for a parameter name.
GetParameterType method	Gets the storage/host variable type of a parameter.

Name	Description
GetPlan method	Gets a text-based description of the query execution plan.
GetResultSetSchema method	Gets the schema for the result set.
GetRowsAffectedCount method	Gets the number of rows affected by the last statement.
HasResultSet method	Determines if the SQL statement has a result set.
SetParameterBinary method	Sets a parameter to a ul_binary value.
SetParameterDateTime method	Sets a parameter to a DECL_DATETIME value.
SetParameterDouble method	Sets a parameter to a double value.
SetParameterFloat method	Sets a parameter to a float value.
SetParameterGuid method	Sets a parameter to a GUID value.
SetParameterInt method	Sets a parameter to an integer value.
SetParameterIntWithType method	Sets a parameter to an integer value of the specified integer type.
SetParameterNull method	Sets a parameter to null.
SetParameterString method	Sets a parameter to a string value.

AppendParameterByteChunk method

Sets a large binary parameter broken down into several chunks.

Syntax

```
public virtual bool AppendParameterByteChunk(
     ul_column_num pid,
    const ul_byte * value,
    size_t valueSize
)
```

Parameters

- **pid** The 1-based ordinal of the parameter.
- **value** The byte chunk to append.
- **valueSize** The size of the buffer.

Returns

True on success; otherwise, returns false.

AppendParameterStringChunk method

Sets a large string parameter broken down into several chunks.

Syntax

```
public virtual bool AppendParameterStringChunk(
     ul_column_num pid,
    const char * value,
    size_t len
)
```

Parameters

- **pid** The 1-based ordinal of the parameter.
- **value** The string chunk to append.
- **len** Optional. Set to the length of the string chunk in bytes or UL_NULL_TERMINATED_STRING if the string chunk is null-terminated.

Returns

True on success; otherwise, returns false.

Close method

Destroys this object.

Syntax

```
public virtual void Close()
```

ExecuteQuery method

Executes a SQL SELECT statement as a query.

Syntax

```
public virtual ULResultSet * ExecuteQuery()
```

Returns

The ULResultSet object that contains the results of the query, as a set of rows.

ExecuteStatement method

Executes a statement that does not return a result set, such as a SQL INSERT, DELETE or UPDATE statement.

Syntax

```
public virtual bool ExecuteStatement()
```

Returns

True on success; otherwise, returns false.

GetConnection method

Gets the connection object.

Syntax

```
public virtual ULConnection * GetConnection()
```

Returns

The ULConnection object associated with this prepared statement.

GetParameterCount method

Gets the number of input parameters for this statement.

Syntax

```
public virtual ul_u_short GetParameterCount()
```

Returns

The number of input parameters for this statement.

GetParameterID method

Get the 1-based ordinal for a parameter name.

Syntax

```
public virtual ul_column_num GetParameterID(const char * name)
```

Parameters

- **name** The name of the host variable.

Returns

The 1-based ordinal for a parameter name.

GetParameterType method

Gets the storage/host variable type of a parameter.

Syntax

```
public virtual ul_column_storage_type GetParameterType(
     ul_column_num pid
)
```

Parameters

- **pid** The 1-based ordinal of the parameter.

Returns

The type of the specified parameter.

See also

- "ul_column_storage_type enumeration [UltraLite C and Embedded SQL datatypes]" on page 104

GetPlan method

Gets a text-based description of the query execution plan.

Syntax

```
public virtual size_t GetPlan(char * dst, size_t dstSize)
```

Parameters

- **dst** The destination buffer for the plan text. Pass NULL to determine the size of the buffer required to hold the plan.
- **dstSize** The size of the destination buffer.

Returns

The number of bytes copied to the buffer; otherwise, if the dst value is NULL, returns the number of bytes required to store the plan, excluding the null-terminator.

Remarks

This method is intended primarily for use during development.

An empty string is returned if there is no plan. Plans exist when the prepared statement is a SQL query.

When the plan is obtained before the associated query has been executed, the plan shows the operations used to execute the query. The plan additionally shows the number of rows each operation produced when the plan is obtained after the query has been executed. This plan can be used to gain insight about the execution of the query.

GetResultSetSchema method

Gets the schema for the result set.

Syntax

```
public virtual const ULResultSetSchema & GetResultSetSchema()
```

Returns

A ULResultSetSchema object that can be used to get information about the schema of the result set.

GetRowsAffectedCount method

Gets the number of rows affected by the last statement.

Syntax

```
public virtual ul_s_long GetRowsAffectedCount()
```

Returns

The number of rows affected by the last statement. If the number of rows is not available (for instance, the statement alters the schema rather than data) the return value is -1.

HasResultSet method

Determines if the SQL statement has a result set.

Syntax

```
public virtual bool HasResultSet()
```

Returns

True if a result set is generated when this statement is executed; otherwise, returns false if no result set is generated.

SetParameterBinary method

Sets a parameter to a ul_binary value.

Syntax

```
public virtual bool SetParameterBinary(
     ul_column_num pid,
    const p_ul_binary value
)
```

Parameters

- **pid** The 1-based ordinal of the parameter.
- **value** The ul_binary value.

Returns

True on success; otherwise, returns false.

SetParameterDateTime method

Sets a parameter to a DECL_DATETIME value.

Syntax

```
public virtual bool SetParameterDateTime(
     ul_column_num pid,
    DECL_DATETIME * value
)
```

Parameters

- **pid** The 1-based ordinal of the parameter.
- **value** The DECL_DATETIME value.

Returns

True on success; otherwise, returns false.

SetParameterDouble method

Sets a parameter to a double value.

Syntax

```
public virtual bool SetParameterDouble(
     ul_column_num pid,
    ul_double value
)
```

Parameters

- **pid** The 1-based ordinal of the parameter.
- **value** The double value.

Returns

True on success; otherwise, returns false.

SetParameterFloat method

Sets a parameter to a float value.

Syntax

```
public virtual bool SetParameterFloat(ul_column_num pid, ul_real value)
```

Parameters

- **pid** The 1-based ordinal of the parameter.
- **value** The float value.

Returns

True on success; otherwise, returns false.

SetParameterGuid method

Sets a parameter to a GUID value.

Syntax

```
public virtual bool SetParameterGuid(ul_column_num pid, GUID * value)
```

Parameters

- **pid** The 1-based ordinal of the parameter.
- **value** The GUID value.

Returns

True on success; otherwise, returns false.

SetParameterInt method

Sets a parameter to an integer value.

Syntax

```
public virtual bool SetParameterInt(ul_column_num pid, ul_s_long value)
```

Parameters

- **pid** The 1-based ordinal of the parameter.

- **value** The integer value.

Returns

True on success; otherwise, returns false.

SetParameterIntWithType method

Sets a parameter to an integer value of the specified integer type.

Syntax

```
public virtual bool SetParameterIntWithType(
     ul_column_num pid,
    ul_s_big value,
    ul_column_storage_type type
)
```

Parameters

- **pid** The 1-based ordinal of the parameter.
- **value** The integer value.
- **type** The integer type to treat the value as.

Returns

True on success; otherwise, returns false.

Remarks

The following is a list of integer values that can be used for the value parameter:

- UL_TYPE_BIT
- UL_TYPE_TINY
- UL_TYPE_S_SHORT
- UL_TYPE_U_SHORT
- UL_TYPE_S_LONG
- UL_TYPE_U_LONG
- UL_TYPE_S_BIG
- UL_TYPE_U_BIG

See also

- “ul_column_storage_type enumeration [UltraLite C and Embedded SQL datatypes]” on page 104

SetParameterNull method

Sets a parameter to null.

Syntax

```
public virtual bool SetParameterNull(ul_column_num pid)
```

Parameters

- **pid** The 1-based ordinal of the parameter.

Returns

True on success; otherwise, returns false.

SetParameterString method

Sets a parameter to a string value.

Syntax

```
public virtual bool SetParameterString(
     ul_column_num pid,
    const char * value,
    size_t len
)
```

Parameters

- **pid** The 1-based ordinal of the parameter.
- **value** The string value.
- **len** Optional. Set to the length of the string in bytes or UL_NULL_TERMINATED_STRING if the string in null-terminated. SQLE_INVALID_PARAMETER is set if this parameter is greater than 32K. For large strings, call the AppendParameterStringChunk method instead.

Returns

True on success, otherwise false.

See also

- "ULPreparedStatement.AppendParameterStringChunk method [UltraLite C++]" on page 170

ULResultSet class

Represents a result set in an UltraLite database.

Syntax

```
public class ULResultSet
```

Derived classes

- "ULTable class [UltraLite C++]" on page 218

Members

All members of ULResultSet class, including all inherited members.

Name	Description
AfterLast method	Moves the cursor after the last row.
AppendByteChunk method	Appends bytes to a column.
AppendStringChunk method	Appends a string chunk to a column.
BeforeFirst method	Moves the cursor before the first row.
Close method	Destroys this object.
Delete method	Deletes the current row and moves it to the next valid row.
DeleteNamed method	Deletes the current row and moves it to the next valid row.
First method	Moves the cursor to the first row.
GetBinary method	Fetches a value from a column as a ul_binary value.
GetBinaryLength method	Gets the binary length of the value of a column.
GetByteChunk method	Gets a binary chunk from the column.
GetConnection method	Gets the connection object.
GetDateTime method	Fetches a value from a column as a DECL_DATETIME.
GetDouble method	Fetches a value from a column as a double.
GetFloat method	Fetches a value from a column as a float.
GetGuid method	Fetches a value from a column as a GUID.
GetInt method	Fetches a value from a column as an integer.
GetIntWithType method	Fetches a value from a column as the specified integer type.
GetResultSetSchema method	Returns an object that can be used to get information about the result set.

Name	Description
GetRowCount method	Gets the number of rows in the table.
GetState method	Gets the internal state of the cursor.
GetString method	Fetches a value from a column as a null-terminated string.
GetStringChunk method	Gets a string chunk from the column.
GetStringLength method	Gets the string length of the value of a column.
IsNull method	Checks if a column is NULL.
Last method	Moves the cursor to the last row.
Next method	Moves the cursor forward one row.
Previous method	Moves the cursor back one row.
Relative method	Moves the cursor by offset rows from the current cursor position.
SetBinary method	Sets a column to a ul_binary value.
SetDateTime method	Sets a column to a DECL_DATETIME value.
SetDefault method	Sets a column to its default value.
SetDouble method	Sets a column to a double value.
SetFloat method	Sets a column to a float value.
SetGuid method	Sets a column to a GUID value.
SetInt method	Sets a column to an integer value.
SetIntWithType method	Sets a column to an integer value of the specified integer type.
SetNull method	Sets a column to null.
SetString method	Sets a column to a string value.
Update method	Updates the current row.
UpdateBegin method	Selects the update mode for setting columns.

AfterLast method

Moves the cursor after the last row.

Syntax

```
public virtual bool AfterLast()
```

Returns

True on success; otherwise, returns false.

AppendByteChunk method

Appends bytes to a column.

Overload list

Name	Description
AppendByteChunk(const char *, const ul_byte *, size_t) method	Appends bytes to a column.
AppendByteChunk(ul_column_num, const ul_byte *, size_t) method	Appends bytes to a column.

AppendByteChunk(const char *, const ul_byte *, size_t) method

Appends bytes to a column.

Syntax

```
public virtual bool AppendByteChunk(
     const char * cname,
    const ul_byte * value,
    size_t valueSize
)
```

Parameters

- **cname** The name of the column.
- **value** The byte chunk to append.
- **valueSize** The size of the byte chunk in bytes.

Returns

True on success; otherwise, returns false.

Remarks

The given bytes are appended to the end of the column written so far by AppendBinaryChunk method calls.

See also

- "ULResultSet.AppendByteChunk method [UltraLite C++]" on page 180

AppendByteChunk(ul_column_num, const ul_byte *, size_t) method

Appends bytes to a column.

Syntax

```
public virtual bool AppendByteChunk(
     ul_column_num cid,
    const ul_byte * value,
    size_t valueSize
)
```

Parameters

- **cid** The 1-based ordinal column number.
- **value** The byte chunk to append.
- **valueSize** The size of the byte chunk in bytes.

Returns

True on success; otherwise, returns false.

Remarks

The given bytes are appended to the end of the column written so far by AppendBinaryChunk method calls.

See also

- "ULResultSet.AppendByteChunk method [UltraLite C++]" on page 180

AppendStringChunk method

Appends a string chunk to a column.

Overload list

Name	Description
AppendStringChunk(const char *, const char *, size_t) method	Appends a string chunk to a column.
AppendStringChunk(ul_column_num, const char *, size_t) method	Appends a string chunk to a column.

AppendStringChunk(const char *, const char *, size_t) method

Appends a string chunk to a column.

Syntax

```
public virtual bool AppendStringChunk(
     const char * cname,
    const char * value,
    size_t len
)
```

Parameters

- **cname** The name of the column.
- **value** The string chunk to append.
- **len** Optional. The length of the string chunk in bytes or the UL_NULL_TERMINATED_STRING constant if the string is null-terminated.

Returns

True on success; otherwise, returns false.

Remarks

This method appends the given string to the end of the string written so far by AppendStringChunk method calls.

See also

- "ULResultSet.AppendStringChunk method [UltraLite C++]" on page 181

AppendStringChunk(ul_column_num, const char *, size_t) method

Appends a string chunk to a column.

Syntax

```
public virtual bool AppendStringChunk(
     ul_column_num cid,
    const char * value,
    size_t len
)
```

Parameters

- **cid** The 1-based ordinal column number.
- **value** The string chunk to append.
- **len** Optional. The length of the string chunk in bytes or the UL_NULL_TERMINATED_STRING constant if the string is null-terminated.

Returns

True on success; otherwise, returns false.

Remarks

This method appends the given string to the end of the string written so far by AppendStringChunk method calls.

See also

- "ULResultSet.AppendStringChunk method [UltraLite C++]" on page 181

BeforeFirst method

Moves the cursor before the first row.

Syntax

```
public virtual bool BeforeFirst()
```

Returns

True on success; otherwise, returns false.

Close method

Destroys this object.

Syntax

```
public virtual void Close()
```

Delete method

Deletes the current row and moves it to the next valid row.

Syntax

```
public virtual bool Delete()
```

Returns

True on success, otherwise false.

DeleteNamed method

Deletes the current row and moves it to the next valid row.

Syntax

```
public virtual bool DeleteNamed(const char * tableName)
```

Parameters

- **tableName** A table name or its correlation (required when the database has multiple columns that share the same table name).

Returns

True on success; otherwise, returns false.

First method

Moves the cursor to the first row.

Syntax

```
public virtual bool First()
```

Returns

True on success; otherwise, returns false.

GetBinary method

Fetches a value from a column as a ul_binary value.

Overload list

Name	Description
GetBinary(const char *, p_ul_binary, size_t) method	Fetches a value from a column as a ul_binary value.
GetBinary(ul_column_num, p_ul_binary, size_t) method	Fetches a value from a column as a ul_binary value.

GetBinary(const char *, p_ul_binary, size_t) method

Fetches a value from a column as a ul_binary value.

Syntax

```
public virtual bool GetBinary(
     const char * cname,
    p_ul_binary dst,
    size_t len
)
```


Parameters

- **cname** The name of the column.
- **dst** The ul_binary result.
- **len** The size of the ul_binary object.

Returns

True if the value was successfully fetched.

GetBinary(ul_column_num, p_ul_binary, size_t) method

Fetches a value from a column as a ul_binary value.

Syntax

```
public virtual bool GetBinary(
     ul_column_num cid,
    p_ul_binary dst,
    size_t len
)
```

Parameters

- **cid** The 1-based ordinal column number.
- **dst** The ul_binary result.
- **len** The size of the ul_binary object.

Returns

True if the value was successfully fetched.

GetBinaryLength method

Gets the binary length of the value of a column.

Overload list

Name	Description
GetBinaryLength(const char *) method	Gets the binary length of the value of a column.
GetBinaryLength(ul_column_num) method	Gets the binary length of the value of a column.

GetBinaryLength(const char *) method

Gets the binary length of the value of a column.

Syntax

```
public virtual size_t GetBinaryLength(const char * cname)
```

Parameters

- **cname** The name of the column.

Returns

The size of the column value as a binary

GetBinaryLength(ul_column_num) method

Gets the binary length of the value of a column.

Syntax

```
public virtual size_t GetBinaryLength(ul_column_num cid)
```

Parameters

- **cid** The 1-based ordinal column number.

Returns

The size of the column value as a binary

GetByteChunk method

Gets a binary chunk from the column.

Overload list

Name	Description
GetByteChunk(const char *, ul_byte *, size_t, size_t) method	Gets a binary chunk from the column.
GetByteChunk(ul_column_num, ul_byte *, size_t, size_t) method	Gets a binary chunk from the column.

GetByteChunk(const char *, ul_byte *, size_t, size_t) method

Gets a binary chunk from the column.

Syntax

```
public virtual size_t GetByteChunk(
     const char * cname,
    ul_byte * dst,
    size_t len,
```



```
    size_t offset
)
```

Parameters

- **cname** The name of the column.
- **dst** The buffer to hold the bytes.
- **len** The size of the buffer in bytes.
- **offset** The offset into the value at which to start reading or the UL_BLOB_CONTINUE constant to continue from where the last read ended.

Returns

The number of bytes copied to the destination buffer. If the dst value is NULL, then the number of bytes left is returned. An empty string is returned in the dst parameter when the column is null; use the IsNull method to differentiate between null and empty strings.

Remarks

The end of the value has been reached if 0 is returned.

See also

- "ULResultSet.IsNull method [UltraLite C++]" on page 200

GetByteChunk(ul_column_num, ul_byte *, size_t, size_t) method

Gets a binary chunk from the column.

Syntax

```
public virtual size_t GetByteChunk(
     ul_column_num cid,
    ul_byte * dst,
    size_t len,
    size_t offset
)
```

Parameters

- **cid** The 1-based ordinal column number.
- **dst** The buffer to hold the bytes.
- **len** The size of the buffer in bytes.
- **offset** The offset into the value at which to start reading or the UL_BLOB_CONTINUE constant to continue from where the last read ended.

Returns

The number of bytes copied to the destination buffer. If the dst value is NULL, then the number of bytes left is returned. An empty string is returned in the dst parameter when the column is null; use the IsNull method to differentiate between null and empty strings.

Remarks

The end of the value has been reached if 0 is returned.

See also

- "ULResultSet.IsNull method [UltraLite C++]" on page 200

GetConnection method

Gets the connection object.

Syntax

```
public virtual ULConnection * GetConnection()
```

Returns

The ULConnection object associated with this result set.

GetDateTime method

Fetches a value from a column as a DECL_DATETIME.

Overload list

Name	Description
GetDateTime(const char *, DECL_DATETIME *) method	Fetches a value from a column as a DECL_DATETIME.
GetDateTime(ul_column_num, DECL_DATETIME *) method	Fetches a value from a column as a DECL_DATETIME.

GetDateTime(const char *, DECL_DATETIME *) method

Fetches a value from a column as a DECL_DATETIME.

Syntax

```
public virtual bool GetDateTime(const char * cname, DECL_DATETIME * dst)
```


Parameters

- **cname** The name of the column.
- **dst** The DECL_DATETIME value.

Returns

True if the value was successfully fetched.

GetDateTime(ul_column_num, DECL_DATETIME *) method

Fetches a value from a column as a DECL_DATETIME.

Syntax

```
public virtual bool GetDateTime(ul_column_num cid, DECL_DATETIME * dst)
```

Parameters

- **cid** The 1-based ordinal column number.
- **dst** The DECL_DATETIME value.

Returns

True if the value was successfully fetched.

GetDouble method

Fetches a value from a column as a double.

Overload list

Name	Description
GetDouble(const char *) method	Fetches a value from a column as a double.
GetDouble(ul_column_num) method	Fetches a value from a column as a double.

GetDouble(const char *) method

Fetches a value from a column as a double.

Syntax

```
public virtual ul_double GetDouble(const char * cname)
```

Parameters

- **cname** The name of the column.

Returns

The column value as a double.

GetDouble(ul_column_num) method

Fetches a value from a column as a double.

Syntax

```
public virtual ul_double GetDouble(ul_column_num cid)
```

Parameters

- **cid** The 1-based ordinal column number.

Returns

The column value as a double.

GetFloat method

Fetches a value from a column as a float.

Overload list

Name	Description
GetFloat(const char *) method	Fetches a value from a column as a float.
GetFloat(ul_column_num) method	Fetches a value from a column as a float.

GetFloat(const char *) method

Fetches a value from a column as a float.

Syntax

```
public virtual ul_real GetFloat(const char * cname)
```

Parameters

- **cname** The name of the column.

Returns

The column value as a float.

GetFloat(ul_column_num) method

Fetches a value from a column as a float.

Syntax

```
public virtual ul_real GetFloat(ul_column_num cid)
```

Parameters

- **cid** The 1-based ordinal column number.

Returns

The column value as a float.

GetGuid method

Fetches a value from a column as a GUID.

Overload list

Name	Description
GetGuid(const char *, GUID *) method	Fetches a value from a column as a GUID.
GetGuid(ul_column_num, GUID *) method	Fetches a value from a column as a GUID.

GetGuid(const char *, GUID *) method

Fetches a value from a column as a GUID.

Syntax

```
public virtual bool GetGuid(const char * cname, GUID * dst)
```

Parameters

- **cname** The name of the column.
- **dst** The GUID value.

Returns

True if the value was successfully fetched.

GetGuid(ul_column_num, GUID *) method

Fetches a value from a column as a GUID.

Syntax

```
public virtual bool GetGuid(ul_column_num cid, GUID * dst)
```

Parameters

- **cid** The 1-based ordinal column number.
- **dst** The GUID value.

Returns

True if the value was successfully fetched.

GetInt method

Fetches a value from a column as an integer.

Overload list

Name	Description
GetInt(const char *) method	Fetches a value from a column as an integer.
GetInt(ul_column_num) method	Fetches a value from a column as an integer.

GetInt(const char *) method

Fetches a value from a column as an integer.

Syntax

```
public virtual ul_s_long GetInt(const char * cname)
```

Parameters

- **cname** The name of the column.

Returns

The column value as an integer.

GetInt(ul_column_num) method

Fetches a value from a column as an integer.

Syntax

```
public virtual ul_s_long GetInt(ul_column_num cid)
```

Parameters

- **cid** The 1-based ordinal column number.

Returns

The column value as an integer.

GetIntWithType method

Fetches a value from a column as the specified integer type.

Overload list

Name	Description
GetIntWithType(const char *, ul_column_storage_type) method	Fetches a value from a column as the specified integer type.
GetIntWithType(ul_column_num, ul_column_storage_type) method	Fetches a value from a column as the specified integer type.

GetIntWithType(const char *, ul_column_storage_type) method

Fetches a value from a column as the specified integer type.

Syntax

```
public virtual ul_s_big GetIntWithType(
     const char * cname,
    ul_column_storage_type type
)
```

Parameters

- **cname** The name of the column.
- **type** The integer type to fetch as.

Returns

The column value as an integer.

Remarks

The following is a list of integer values that can be used for the type parameter:

- UL_TYPE_BIT
- UL_TYPE_TINY
- UL_TYPE_S_SHORT
- UL_TYPE_U_SHORT
- UL_TYPE_S_LONG

- UL_TYPE_U_LONG
- UL_TYPE_S_BIG
- UL_TYPE_U_BIG

See also

- "ul_column_storage_type enumeration [UltraLite C and Embedded SQL datatypes]" on page 104

GetIntWithType(ul_column_num, ul_column_storage_type) method

Fetches a value from a column as the specified integer type.

Syntax

```
public virtual ul_s_big GetIntWithType(
     ul_column_num cid,
    ul_column_storage_type type
)
```

Parameters

- **cid** The 1-based ordinal column number.
- **type** The integer type to fetch as.

Returns

The column value as an integer.

Remarks

The following is a list of integer values that can be used for the type parameter:

- UL_TYPE_BIT
- UL_TYPE_TINY
- UL_TYPE_S_SHORT
- UL_TYPE_U_SHORT
- UL_TYPE_S_LONG
- UL_TYPE_U_LONG
- UL_TYPE_S_BIG
- UL_TYPE_U_BIG

See also

- "ul_column_storage_type enumeration [UltraLite C and Embedded SQL datatypes]" on page 104

GetResultSetSchema method

Returns an object that can be used to get information about the result set.

Syntax

```
public virtual const ULResultSetSchema & GetResultSetSchema()
```

Returns

A ULResultSetSchema object that can be used to get information about the result set.

GetRowCount method

Gets the number of rows in the table.

Syntax

```
public virtual ul_u_long GetRowCount(ul_u_long threshold)
```

Parameters

- **threshold** The limit on the number of rows to count. Set to 0 to indicate no limit.

Returns

The number of rows in the table.

Remarks

This method is equivalent to executing the "SELECT COUNT(*) FROM table" statement.

GetState method

Gets the internal state of the cursor.

Syntax

```
public virtual UL_RS_STATE GetState()
```

Returns

The state of the cursor.

See also

- "UL_RS_STATE enumeration [UltraLite C and Embedded SQL datatypes]" on page 102

GetString method

Fetches a value from a column as a null-terminated string.

Overload list

Name	Description
GetString(const char *, char *, size_t) method	Fetches a value from a column as a null-terminated string.
GetString(ul_column_num, char *, size_t) method	Fetches a value from a column as a null-terminated string.

GetString(const char *, char *, size_t) method

Fetches a value from a column as a null-terminated string.

Syntax

```
public virtual bool GetString(
     const char * cname,
    char * dst,
    size_t len
)
```

Parameters

- **cname** The name of the column.
- **dst** The buffer to hold the string value. The string is null-terminated even if truncated.
- **len** The size of the buffer in bytes.

Returns

True if the value was successfully fetched.

Remarks

The string is truncated in the buffer when it isn't large enough to hold the entire value.

GetString(ul_column_num, char *, size_t) method

Fetches a value from a column as a null-terminated string.

Syntax

```
public virtual bool GetString(ul_column_num cid, char * dst, size_t len)
```

Parameters

- **cid** The 1-based ordinal column number.

- **dst** The buffer to hold the string value. The string is null-terminated even if truncated.
- **len** The size of the buffer in bytes.

Returns

True if the value was successfully fetched.

Remarks

The string is truncated in the buffer when it isn't large enough to hold the entire value.

GetStringChunk method

Gets a string chunk from the column.

Overload list

Name	Description
GetStringChunk(const char *, char *, size_t, size_t) method	Gets a string chunk from the column.
GetStringChunk(ul_column_num, char *, size_t, size_t) method	Gets a string chunk from the column.

GetStringChunk(const char *, char *, size_t, size_t) method

Gets a string chunk from the column.

Syntax

```
public virtual size_t GetStringChunk(
     const char * cname,
    char * dst,
    size_t len,
    size_t offset
)
```

Parameters

- **cname** The name of the column.
- **dst** The buffer to hold the string chunk. The string is null-terminated even if truncated.
- **len** The size of the buffer in bytes.
- **offset** The offset into the value at which to start reading or the UL_BLOB_CONTINUE constant to continue from where the last read ended.

Returns

The number of bytes copied to the destination buffer excluding the null-terminator. If the dst value is set to NULL, then the number of bytes left in the string is returned. An empty string is returned in the dst parameter when the column is null; use the IsNull method to differentiate between null and empty strings.

Remarks

The end of the value has been reached if 0 is returned.

See also

- "ULResultSet.IsNull method [UltraLite C++]" on page 200

GetStringChunk(ul_column_num, char *, size_t, size_t) method

Gets a string chunk from the column.

Syntax

```
public virtual size_t GetStringChunk(
     ul_column_num cid,
    char * dst,
    size_t len,
    size_t offset
)
```

Parameters

- **cid** The 1-based ordinal column number.
- **dst** The buffer to hold the string chunk. The string is null-terminated even if truncated.
- **len** The size of the buffer in bytes.
- **offset** Set to the offset into the value at which to start reading or set to the UL_BLOB_CONTINUE constant to continue from where the last read ended.

Returns

The number of bytes copied to the destination buffer excluding the null-terminator. If the dst value is set to NULL, then the number of bytes left in the string is returned. An empty string is returned in the dst parameter when the column is null; use the IsNull method to differentiate between null and empty strings.

Remarks

The end of the value has been reached if 0 is returned.

See also

- "ULResultSet.IsNull method [UltraLite C++]" on page 200

GetStringLength method

Gets the string length of the value of a column.

Overload list

Name	Description
GetStringLength(const char *) method	Gets the string length of the value of a column.
GetStringLength(ul_column_num) method	Gets the string length of the value of a column.

GetStringLength(const char *) method

Gets the string length of the value of a column.

Syntax

```
public virtual size_t GetStringLength(const char * cname)
```

Parameters

- **cname** The name of the column.

Returns

The number of bytes or characters required to hold the string returned by one of the GetString methods, not including the null-terminator.

Remarks

The following example demonstrates how to get the string length of a column:

```
len = result_set->GetStringLength( cid );
dst = new char[ len + 1 ];
result_set->GetString( cid, dst, len + 1 );
```

For wide characters, the usage is as follows:

```
len = result_set->GetStringLength( cid );
dst = new ul_wchar[ len + 1 ];
result_set->GetString( cid, dst, len + 1 );
```

See also

- "ULResultSet.GetString method [UltraLite C++]" on page 196

GetStringLength(ul_column_num) method

Gets the string length of the value of a column.

Syntax

```
public virtual size_t GetStringLength(ul_column_num cid)
```

Parameters

- **cid** The 1-based ordinal column number.

Returns

The number of bytes or characters required to hold the string returned by one of the GetString methods, not including the null-terminator.

Remarks

The following example illustrates how to get the string length of a column:

```
len = result_set->GetStringLength( cid );
dst = new char[ len + 1 ];
result_set->GetString( cid, dst, len + 1 );
```

For wide characters, the usage is as follows:

```
len = result_set->GetStringLength( cid );
dst = new ul_wchar[ len + 1 ];
result_set->GetString( cid, dst, len + 1 );
```

See also

- "ULResultSet.GetString method [UltraLite C++]" on page 196

IsNull method

Checks if a column is NULL.

Overload list

Name	Description
IsNull(const char *) method	Checks if a column is NULL.
IsNull(ul_column_num) method	Checks if a column is NULL.

IsNull(const char *) method

Checks if a column is NULL.

Syntax

```
public virtual bool IsNull(const char * cname)
```

Parameters

- **cname** The name of the column.

Returns

True if the value for the column is NULL.

IsNull(ul_column_num) method

Checks if a column is NULL.

Syntax

```
public virtual bool IsNull(ul_column_num cid)
```

Parameters

- **cid** The 1-based ordinal column number.

Returns

True if the value for the column is NULL.

Last method

Moves the cursor to the last row.

Syntax

```
public virtual bool Last()
```

Returns

True on success; otherwise, returns false.

Next method

Moves the cursor forward one row.

Syntax

```
public virtual bool 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 is returned if it fails to move forward. For example, there may not be a next row. In this case, the resulting cursor position is set after the last row.

Previous method

Moves the cursor back one row.

Syntax

```
public virtual bool Previous()
```

Returns

True, if the cursor successfully moves back one row. False, if it fails to move backward. The resulting cursor position is set before the first row.

Relative method

Moves the cursor by offset rows from the current cursor position.

Syntax

```
public virtual bool Relative(ul_fetch_offset offset)
```

Parameters

- **offset** The number of rows to move.

Returns

True on success; otherwise, returns false.

SetBinary method

Sets a column to a ul_binary value.

Overload list

Name	Description
SetBinary(const char *, p_ul_binary) method	Sets a column to a ul_binary value.
SetBinary(ul_column_num, p_ul_binary) method	Sets a column to a ul_binary value.

SetBinary(const char *, p_ul_binary) method

Sets a column to a ul_binary value.

Syntax

```
public virtual bool SetBinary(const char * cname, p_ul_binary value)
```

Parameters

- **cname** The name of the column.
- **value** The ul_binary value. Passing NULL is equivalent to calling the SetNull method.

Returns

True on success; otherwise, returns false.

SetBinary(ul_column_num, p_ul_binary) method

Sets a column to a ul_binary value.

Syntax

```
public virtual bool SetBinary(ul_column_num cid, p_ul_binary value)
```

Parameters

- **cid** The 1-based ordinal column number.
- **value** The ul_binary value. Passing NULL is equivalent to calling the SetNull method.

Returns

True on success; otherwise, returns false.

SetDateTime method

Sets a column to a DECL_DATETIME value.

Overload list

Name	Description
SetDateTime(const char *, DECL_DATETIME *) method	Sets a column to a DECL_DATETIME value.
SetDateTime(ul_column_num, DECL_DATETIME *) method	Sets a column to a DECL_DATETIME value.

SetDateTime(const char *, DECL_DATETIME *) method

Sets a column to a DECL_DATETIME value.

Syntax

```
public virtual bool SetDateTime(
     const char * cname,
    DECL_DATETIME * value
)
```

Parameters

- **cname** The name of the column.
- **value** The DECL_DATETIME value. Passing NULL is equivalent to calling the SetNull method.

Returns

True on success; otherwise, returns false.

SetDateTime(ul_column_num, DECL_DATETIME *) method

Sets a column to a DECL_DATETIME value.

Syntax

```
public virtual bool SetDateTime(
     ul_column_num cid,
    DECL_DATETIME * value
)
```

Parameters

- **cid** The 1-based ordinal column number.
- **value** The DECL_DATETIME value. Passing NULL is equivalent to calling the SetNull method.

Returns

True on success; otherwise, returns false.

SetDefault method

Sets a column to its default value.

Overload list

Name	Description
SetDefault(const char *) method	Sets a column to its default value.
SetDefault(ul_column_num) method	Sets a column to its default value.

SetDefault(const char *) method

Sets a column to its default value.

Syntax

```
public virtual bool SetDefault(const char * cname)
```

Parameters

- **cname** The name of the column.

Returns

True on success; otherwise, returns false.

SetDefault(ul_column_num) method

Sets a column to its default value.

Syntax

```
public virtual bool SetDefault(ul_column_num cid)
```

Parameters

- **cid** The 1-based ordinal column number.

Returns

True on success; otherwise, returns false.

SetDouble method

Sets a column to a double value.

Overload list

Name	Description
SetDouble(const char *, ul_double) method	Sets a column to a double value.
SetDouble(ul_column_num, ul_double) method	Sets a column to a double value.

SetDouble(const char *, ul_double) method

Sets a column to a double value.

Syntax

```
public virtual bool SetDouble(const char * cname, ul_double value)
```

Parameters

- **cname** The name of the column.
- **value** The double value.

Returns

True on success; otherwise, returns false.

SetDouble(ul_column_num, ul_double) method

Sets a column to a double value.

Syntax

```
public virtual bool SetDouble(ul_column_num cid, ul_double value)
```

Parameters

- **cid** The 1-based ordinal column number.
- **value** The double value.

Returns

True on success; otherwise, returns false.

SetFloat method

Sets a column to a float value.

Overload list

Name	Description
SetFloat(const char *, ul_real) method	Sets a column to a float value.
SetFloat(ul_column_num, ul_real) method	Sets a column to a float value.

SetFloat(const char *, ul_real) method

Sets a column to a float value.

Syntax

```
public virtual bool SetFloat(const char * cname, ul_real value)
```

Parameters

- **cname** The name of the column.
- **value** The float value.

Returns

True on success; otherwise, returns false.

SetFloat(ul_column_num, ul_real) method

Sets a column to a float value.

Syntax

```
public virtual bool SetFloat(ul_column_num cid, ul_real value)
```

Parameters

- **cid** The 1-based ordinal column number.

- **value** The float value.

Returns

True on success; otherwise, returns false.

SetGuid method

Sets a column to a GUID value.

Overload list

Name	Description
SetGuid(const char *, GUID *) method	Sets a column to a GUID value.
SetGuid(ul_column_num, GUID *) method	Sets a column to a GUID value.

SetGuid(const char *, GUID *) method

Sets a column to a GUID value.

Syntax

```
public virtual bool SetGuid(const char * cname, GUID * value)
```

Parameters

- **cname** The name of the column.
- **value** The GUID value. Passing NULL is equivalent to calling the SetNull method.

Returns

True on success; otherwise, returns false.

SetGuid(ul_column_num, GUID *) method

Sets a column to a GUID value.

Syntax

```
public virtual bool SetGuid(ul_column_num cid, GUID * value)
```

Parameters

- **cid** The 1-based ordinal column number.
- **value** The GUID value. Passing NULL is equivalent to calling the SetNull method.

Returns

True on success; otherwise, returns false.

SetInt method

Sets a column to an integer value.

Overload list

Name	Description
SetInt(const char *, ul_s_long) method	Sets a column to an integer value.
SetInt(ul_column_num, ul_s_long) method	Sets a column to an integer value.

SetInt(const char *, ul_s_long) method

Sets a column to an integer value.

Syntax

```
public virtual bool SetInt(const char * cname, ul_s_long value)
```

Parameters

- **cname** The name of the column.
- **value** The signed integer value.

Returns

True on success; otherwise, returns false.

SetInt(ul_column_num, ul_s_long) method

Sets a column to an integer value.

Syntax

```
public virtual bool SetInt(ul_column_num cid, ul_s_long value)
```

Parameters

- **cid** The 1-based ordinal column number.
- **value** The signed integer value.

Returns

True on success; otherwise, returns false.

SetIntWithType method

Sets a column to an integer value of the specified integer type.

Overload list

Name	Description
SetIntWithType(const char *, ul_s_big, ul_column_storage_type) method	Sets a column to an integer value of the specified integer type.
SetIntWithType(ul_column_num, ul_s_big, ul_column_storage_type) method	Sets a column to an integer value of the specified integer type.

SetIntWithType(const char *, ul_s_big, ul_column_storage_type) method

Sets a column to an integer value of the specified integer type.

Syntax

```
public virtual bool SetIntWithType(
     const char * cname,
    ul_s_big value,
    ul_column_storage_type type
)
```

Parameters

- **cname** The name of the column.
- **value** The integer value.
- **type** The integer type to treat the value as.

Returns

True on success; otherwise, returns false.

Remarks

The following is a list of integer values that can be used for the value parameter:

- UL_TYPE_BIT
- UL_TYPE_TINY
- UL_TYPE_S_SHORT
- UL_TYPE_U_SHORT
- UL_TYPE_S_LONG

- UL_TYPE_U_LONG
- UL_TYPE_S_BIG
- UL_TYPE_U_BIG

See also

- "ul_column_storage_type enumeration [UltraLite C and Embedded SQL datatypes]" on page 104

SetIntWithType(ul_column_num, ul_s_big, ul_column_storage_type) method

Sets a column to an integer value of the specified integer type.

Syntax

```
public virtual bool SetIntWithType(
     ul_column_num cid,
    ul_s_big value,
    ul_column_storage_type type
)
```

Parameters

- **cid** The 1-based ordinal column number.
- **value** The integer value.
- **type** The integer type to treat the value as.

Returns

True on success; otherwise, returns false.

Remarks

The following is a list of integer values that can be used for the value parameter:

- UL_TYPE_BIT
- UL_TYPE_TINY
- UL_TYPE_S_SHORT
- UL_TYPE_U_SHORT
- UL_TYPE_S_LONG
- UL_TYPE_U_LONG
- UL_TYPE_S_BIG
- UL_TYPE_U_BIG

See also

- "ul_column_storage_type enumeration [UltraLite C and Embedded SQL datatypes]" on page 104

SetNull method

Sets a column to null.

Overload list

Name	Description
SetNull(const char *) method	Sets a column to null.
SetNull(ul_column_num) method	Sets a column to null.

SetNull(const char *) method

Sets a column to null.

Syntax

```
public virtual bool SetNull(const char * cname)
```

Parameters

- **cname** The name of the column.

Returns

True on success; otherwise, returns false.

SetNull(ul_column_num) method

Sets a column to null.

Syntax

```
public virtual bool SetNull(ul_column_num cid)
```

Parameters

- **cid** The 1-based ordinal column number.

Returns

True on success; otherwise, returns false.

SetString method

Sets a column to a string value.

Overload list

Name	Description
SetString(const char *, const char *, size_t) method	Sets a column to a string value.
SetString(ul_column_num, const char *, size_t) method	Sets a column to a string value.

SetString(const char *, const char *, size_t) method

Sets a column to a string value.

Syntax

```
public virtual bool SetString(
     const char * cname,
    const char * value,
    size_t len
)
```

Parameters

- **cname** The name of the column.
- **value** The string value. Passing NULL is equivalent to calling the SetNull method.
- **len** Optional. The length of the string in bytes or the UL_NULL_TERMINATED_STRING constant if the string is null-terminated. The SQLE_INVALID_PARAMETER constant is set if the len value is set larger than 32K. For large strings, call the AppendStringChunk method instead.

Returns

True on success; otherwise, returns false.

See also

- "ULResultSet.AppendStringChunk method [UltraLite C++]" on page 181

SetString(ul_column_num, const char *, size_t) method

Sets a column to a string value.

Syntax

```
public virtual bool SetString(
     ul_column_num cid,
    const char * value,
    size_t len
)
```


Parameters

- **cid** The 1-based ordinal column number.
- **value** The string value. Passing NULL is equivalent to calling the SetNull method.
- **len** Optional. The length of the string in bytes or the UL_NULL_TERMINATED_STRING constant if the string is null-terminated. The SQLE_INVALID_PARAMETER constant is set if the len value is set larger than 32K. For large strings, call the AppendStringChunk method instead.

Returns

True on success; otherwise, returns false.

See also

- "ULResultSet.AppendStringChunk method [UltraLite C++]" on page 181

Update method

Updates the current row.

Syntax

```
public virtual bool Update()
```

Returns

True on success, otherwise false.

UpdateBegin method

Selects the update mode for setting columns.

Syntax

```
public virtual bool UpdateBegin()
```

Returns

True on success, otherwise false.

Remarks

Columns in the primary key may not be modified when in update mode.

ULResultSetSchema class

Represents the schema of an UltraLite result set.

Syntax

```
public class ULResultSetSchema
```

Derived classes

- "ULTableSchema class [UltraLite C++]" on page 226

Members

All members of ULResultSetSchema class, including all inherited members.

Name	Description
GetColumnCount method	Gets the number of columns in the result set or table.
GetColumnID method	Gets the 1-based column ID from its name.
GetColumnName method	Gets the name of a column given its 1-based ID.
GetColumnPrecision method	Gets the precision of a numeric column.
GetColumnScale method	Gets the scale of a numeric column.
GetColumnSize method	Gets the size of the column.
GetColumnSQLType method	Gets the SQL type of a column.
GetColumnType method	Gets the storage/host variable type of a column.
GetConnection method	Gets the ULConnection object.
IsAliased method	Indicates whether the column in a result set was given an alias.

GetColumnCount method

Gets the number of columns in the result set or table.

Syntax

```
public virtual ul_column_num GetColumnCount()
```

Returns

The number of columns in the result set or table.

GetColumnID method

Gets the 1-based column ID from its name.

Syntax

```
public virtual ul_column_num GetColumnID(const char * columnName)
```

Parameters

- **columnName** The column name.

Returns

0 if the column does not exist; otherwise, returns SQLE_COLUMN_NOT_FOUND if the column name does not exist.

GetColumnName method

Gets the name of a column given its 1-based ID.

Syntax

```
public virtual const char * GetColumnName(
     ul_column_num cid,
    ul_column_name_type type
)
```

Parameters

- **cid** The 1-based ordinal column number.
- **type** The desired column name type.

Returns

A pointer to a string buffer containing the column name, if found. The pointer points to a static buffer whose contents may be changed by any subsequent UltraLite call, so you need to make a copy of the value if you need to keep it for a while. If the column does not exist, NULL is returned and SQLE_COLUMN_NOT_FOUND is set.

Remarks

Depending on the type selected and how the column was declared in the SELECT statement, the column name may be returned in the form [table- name].[column-name].

The type parameter is used to specify what type of column name to return.

See also

- "ul_column_name_type enumeration [UltraLite C++]" on page 234

GetColumnPrecision method

Gets the precision of a numeric column.

Syntax

```
public virtual size_t GetColumnPrecision(ul_column_num cid)
```

Parameters

- **cid** The 1-based ordinal column number.

Returns

0 if the column is not a numeric type or if the column does not exist. SQLE_COLUMN_NOT_FOUND is set if the column name does not exist. SQLE_DATATYPE_NOT_ALLOWED is set if the column type is not numeric.

GetColumnScale method

Gets the scale of a numeric column.

Syntax

```
public virtual size_t GetColumnScale(ul_column_num cid)
```

Parameters

- **cid** The 1-based ordinal column number.

Returns

0 if the column is not a numeric type or if the column does not exist. SQLE_COLUMN_NOT_FOUND is set if the column name does not exist. SQLE_DATATYPE_NOT_ALLOWED is set if the column type is not numeric.

GetColumnSize method

Gets the size of the column.

Syntax

```
public virtual size_t GetColumnSize(ul_column_num cid)
```

Parameters

- **cid** The 1-based ordinal column number.

Returns

0 if the column does not exist or if the column type does not have a variable length. SQLE_COLUMN_NOT_FOUND is set if the column name does not exist. SQLE_DATATYPE_NOT_ALLOWED is set if the column type is not UL_SQLTYPE_CHAR or UL_SQLTYPE_BINARY.

GetColumnSQLType method

Gets the SQL type of a column.

Syntax

```
public virtual ul_column_sql_type GetColumnSQLType(ul_column_num cid)
```

Parameters

- **cid** The 1-based ordinal column number.

Returns

UL_SQLTYPE_BAD_INDEX if the column does not exist.

See also

- "ul_column_sql_type enumeration [UltraLite C and Embedded SQL datatypes]" on page 102

GetColumnType method

Gets the storage/host variable type of a column.

Syntax

```
public virtual ul_column_storage_type GetColumnType(ul_column_num cid)
```

Parameters

- **cid** The 1-based ordinal column number.

Returns

UL_TYPE_BAD_INDEX if the column does not exist.

See also

- "ul_column_storage_type enumeration [UltraLite C and Embedded SQL datatypes]" on page 104

GetConnection method

Gets the ULConnection object.

Syntax

```
public virtual ULConnection * GetConnection()
```

Returns

The ULConnection object associated with this result set schema.

IsAliased method

Indicates whether the column in a result set was given an alias.

Syntax

```
public virtual bool IsAliased(ul_column_num cid)
```

Parameters

- **cid** The 1-based ordinal column number.

Returns

True if the column is aliased; otherwise, returns false.

ULTable class

Represents a table in an UltraLite database.

Syntax

```
public class ULTable : ULResultSet
```

Base classes

- "ULResultSet class [UltraLite C++]" on page 177

Members

All members of ULTable class, including all inherited members.

Name	Description
AfterLast method	Moves the cursor after the last row.
AppendByteChunk method	Appends bytes to a column.
AppendStringChunk method	Appends a string chunk to a column.
BeforeFirst method	Moves the cursor before the first row.
Close method	Destroys this object.
Delete method	Deletes the current row and moves it to the next valid row.
DeleteAllRows method	Deletes all rows from a table.
DeleteNamed method	Deletes the current row and moves it to the next valid row.

Name	Description
Find method	Performs an exact match lookup based on the current index scanning forward through the table.
FindBegin method	Prepares to perform a new Find call on a table by entering find mode.
FindFirst method	Performs an exact match lookup based on the current index scanning forward through the table.
FindLast method	Performs an exact match lookup based on the current index scanning backward through the table.
FindNext method	Gets the next row that exactly matches the index.
FindPrevious method	Gets the previous row that exactly matches the index.
First method	Moves the cursor to the first row.
GetBinary method	Fetches a value from a column as a ul_binary value.
GetBinaryLength method	Gets the binary length of the value of a column.
GetByteChunk method	Gets a binary chunk from the column.
GetConnection method	Gets the connection object.
GetDateTime method	Fetches a value from a column as a DECL_DATETIME.
GetDouble method	Fetches a value from a column as a double.
GetFloat method	Fetches a value from a column as a float.
GetGuid method	Fetches a value from a column as a GUID.
GetInt method	Fetches a value from a column as an integer.
GetIntWithType method	Fetches a value from a column as the specified integer type.
GetResultSetSchema method	Returns an object that can be used to get information about the result set.
GetRowCount method	Gets the number of rows in the table.
GetState method	Gets the internal state of the cursor.
GetString method	Fetches a value from a column as a null-terminated string.
GetStringChunk method	Gets a string chunk from the column.

Name	Description
GetStringLength method	Gets the string length of the value of a column.
GetTableSchema method	Returns a ULTableSchema object that can be used to get schema information about the table.
Insert method	Inserts a new row into the table.
InsertBegin method	Selects the insert mode for setting columns.
IsNull method	Checks if a column is NULL.
Last method	Moves the cursor to the last row.
Lookup method	Performs a lookup based on the current index scanning forward through the table.
LookupBackward method	Performs a lookup based on the current index scanning backward through the table.
LookupBegin method	Prepares to perform a new lookup on a table.
LookupForward method	Performs a lookup based on the current index scanning forward through the table.
Next method	Moves the cursor forward one row.
Previous method	Moves the cursor back one row.
Relative method	Moves the cursor by offset rows from the current cursor position.
SetBinary method	Sets a column to a ul_binary value.
SetDateTime method	Sets a column to a DECL_DATETIME value.
SetDefault method	Sets a column to its default value.
SetDouble method	Sets a column to a double value.
SetFloat method	Sets a column to a float value.
SetGuid method	Sets a column to a GUID value.
SetInt method	Sets a column to an integer value.
SetIntWithType method	Sets a column to an integer value of the specified integer type.
SetNull method	Sets a column to null.

Name	Description
SetString method	Sets a column to a string value.
TruncateTable method	Truncates the table and temporarily activates STOP SYNCHRONIZATION DELETE.
Update method	Updates the current row.
UpdateBegin method	Selects the update mode for setting columns.

DeleteAllRows method

Deletes all rows from a table.

Syntax

```
public virtual bool DeleteAllRows()
```

Returns

True on success; otherwise, returns false. For example, false is returned when the table is not open, or a SQL error occurred.

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 synchronization property on the connection, the deleted rows are not synchronized.

Note
Any uncommitted inserts from other connections are not deleted. They are also not deleted if the other connection performs a rollback after it calls the DeleteAllRows method.

If this table has been opened without an index, then it is considered read-only and data cannot be deleted.

Find method

Performs an exact match lookup based on the current index scanning forward through the table.

Syntax

```
public virtual bool Find(ul_column_num ncols)
```

Parameters

- **ncols** For composite indexes, the number of columns to use during the search.

Returns

If no row matches the index value, the cursor position is set after the last row and the method returns 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 first row that exactly matches the index value.

FindBegin method

Prepares to perform a new Find call on a table by entering find mode.

Syntax

```
public virtual bool FindBegin()
```

Returns

True on success; otherwise, returns false.

Remarks

You may only set columns in the index that the table was opened with. This method cannot be called if the table was opened without an index.

FindFirst method

Performs an exact match lookup based on the current index scanning forward through the table.

Syntax

```
public virtual bool FindFirst(ul_column_num ncols)
```

Parameters

- **ncols** For composite indexes, the number of columns to use during the search.

Returns

If no row matches the index value, the cursor position is set after the last row and the method returns 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 first row that exactly matches the index value.

FindLast method

Performs an exact match lookup based on the current index scanning backward through the table.

Syntax

```
public virtual bool FindLast(ul_column_num ncols)
```

Parameters

- **ncols** For composite indexes, the number of columns to use during the search.

Returns

If no row matches the index value, the cursor position is set before the first row and the method returns 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 first row that exactly matches the index value.

FindNext method

Gets the next row that exactly matches the index.

Syntax

```
public virtual bool FindNext(ul_column_num ncols)
```

Parameters

- **ncols** For composite indexes, the number of columns to use during the search.

Returns

False if no more rows match the index. In this case, the cursor is positioned after the last row.

FindPrevious method

Gets the previous row that exactly matches the index.

Syntax

```
public virtual bool FindPrevious(ul_column_num ncols)
```

Parameters

- **ncols** For composite indexes, the number of columns to use during the search.

Returns

False if no more rows match the index. In this case, the cursor is positioned before the first row.

GetTableSchema method

Returns a ULTableSchema object that can be used to get schema information about the table.

Syntax

```
public virtual ULTableSchema * GetTableSchema()
```

Returns

A ULTableSchema object that can be used to get schema information about the table.

Insert method

Inserts a new row into the table.

Syntax

```
public virtual bool Insert()
```

Returns

True on success; otherwise returns false.

InsertBegin method

Selects the insert mode for setting columns.

Syntax

```
public virtual bool InsertBegin()
```

Returns

True on success; otherwise, returns false.

Remarks

All columns are set to their default value during an insert unless an alternative value is supplied via Set method calls.

Lookup method

Performs a lookup based on the current index scanning forward through the table.

Syntax

```
public virtual bool Lookup(ul_column_num ncols)
```


Parameters

- **ncols** For composite indexes, the number of columns to use in the lookup.

Returns

False if the resulting cursor position is set after the last row.

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, the ncols parameter specifies the number of columns to use in the lookup.

LookupBackward method

Performs a lookup based on the current index scanning backward through the table.

Syntax

```
public virtual bool LookupBackward(ul_column_num ncols)
```

Parameters

- **ncols** For composite indexes, the number of columns to use in the lookup.

Returns

False if the resulting cursor position is set before the first row.

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, the ncols parameter specifies the number of columns to use in the lookup.

LookupBegin method

Prepares to perform a new lookup on a table.

Syntax

```
public virtual bool LookupBegin()
```

Returns

True on success; otherwise, returns false.

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 method

Performs a lookup based on the current index scanning forward through the table.

Syntax

```
public virtual bool LookupForward(ul_column_num ncols)
```

Parameters

- **ncols** For composite indexes, the number of columns to use in the lookup.

Returns

False if the resulting cursor position is set after the last row.

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, the ncols parameter specifies the number of columns to use in the lookup.

TruncateTable method

Truncates the table and temporarily activates STOP SYNCHRONIZATION DELETE.

Syntax

```
public virtual bool TruncateTable()
```

Returns

True on success; otherwise, returns false.

ULTableSchema class

Represents the schema of an UltraLite table.

Syntax

```
public class ULTableSchema : ULResultSetSchema
```

Base classes

- "ULResultSetSchema class [UltraLite C++]" on page 213

Members

All members of ULTableSchema class, including all inherited members.

Name	Description
Close method	Destroys this object.
GetColumnCount method	Gets the number of columns in the result set or table.
GetColumnDefault method	Gets the default value for the column if it exists.
GetColumnDefaultType method	Gets the type of column default.
GetColumnID method	Gets the 1-based column ID from its name.
GetColumnName method	Gets the name of a column given its 1-based ID.
GetColumnPrecision method	Gets the precision of a numeric column.
GetColumnScale method	Gets the scale of a numeric column.
GetColumnSize method	Gets the size of the column.
GetColumnSQLType method	Gets the SQL type of a column.
GetColumnType method	Gets the storage/host variable type of a column.
GetConnection method	Gets the ULConnection object.
GetGlobalAutoincPartitionSize method	Gets the partition size.
GetIndexCount method	Gets the number of indexes in the table.
GetIndexSchema method	Gets the schema of an index, given the name.
GetName method	Gets the name of the table.
GetNextIndex method	Gets the next index (schema) in the table.
GetOptimalIndex method	Determines the best index to use for searching for a column value.
GetPrimaryKey method	Gets the primary key for the table.
GetPublicationPredicate method	Gets the publication predicate as a string.
GetTableSyncType method	Gets the table synchronization type.
InPublication method	Checks whether the table is contained in the named publication.

Name	Description
IsAliased method	Indicates whether the column in a result set was given an alias.
IsColumnInIndex method	Checks whether the column is contained in the named index.
IsColumnNullable method	Checks whether the specified column is nullable.

Close method

Destroys this object.

Syntax

```
public virtual void Close()
```

GetColumnDefault method

Gets the default value for the column if it exists.

Syntax

```
public virtual const char * GetColumnDefault(ul_column_num cid)
```

Parameters

- **cid** A 1-based ordinal column number.

Returns

The default value. An empty string is returned if the column has no default value. This value points to a static buffer whose contents may be changed by any subsequent UltraLite call, so make a copy of the value if you need to retain it.

GetColumnDefaultType method

Gets the type of column default.

Syntax

```
public virtual ul_column_default_type GetColumnDefaultType(
     ul_column_num cid
)
```

Parameters

- **cid** A 1-based ordinal column number.

Returns

The type of column default.

See also

- "ul_column_default_type enumeration [UltraLite C++]" on page 233

GetGlobalAutoincPartitionSize method

Gets the partition size.

Syntax

```
public virtual bool GetGlobalAutoincPartitionSize(
     ul_column_num cid,
    ul_u_big * size
)
```

Parameters

- **cid** A 1-based ordinal column number.
- **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

True on success; otherwise, returns false.

GetIndexCount method

Gets the number of indexes in the table.

Syntax

```
public virtual ul_index_num 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.

GetIndexSchema method

Gets the schema of an index, given the name.

Syntax

```
public virtual ULIndexSchema * GetIndexSchema(const char * indexName)
```

Parameters

- **indexName** The name of the index.

Returns

A ULIndexSchema object for the specified index, or NULL if the object does not exist.

GetName method

Gets the name of the table.

Syntax

```
public virtual const char * GetName()
```

Returns

The name of the table. This value points to a static buffer whose contents may be changed by any subsequent UltraLite call, so make a copy of the value if you need to retain it.

GetNextIndex method

Gets the next index (schema) in the table.

Syntax

```
public virtual ULIndexSchema * GetNextIndex(ul_index_iter * iter)
```

Parameters

- **iter** A pointer to the iterator variable.

Returns

A ULIndexSchema object, or NULL when the iteration is complete.

Remarks

Initialize the iter value to the ul_index_iter_start constant before the first call.

See also

- "ul_index_iter_start variable [UltraLite C++]" on page 237

GetOptimalIndex method

Determines the best index to use for searching for a column value.

Syntax

```
public virtual const char * GetOptimalIndex(ul_column_num cid)
```

Parameters

- **cid** A 1-based ordinal column number.

Returns

The name of the index or NULL if the column isn't indexed. This value points to a static buffer whose contents may be changed by any subsequent UltraLite call, so make a copy of the value if you need to keep it for a while.

GetPrimaryKey method

Gets the primary key for the table.

Syntax

```
public virtual ULIndexSchema * GetPrimaryKey()
```

Returns

a ULIndexSchema object for the table's primary key.

GetPublicationPredicate method

Gets the publication predicate as a string.

Syntax

```
public virtual const char * GetPublicationPredicate(
     const char * pubName
)
```

Parameters

- **pubName** The name of the publication.

Returns

The publication predicate string for the specified publication. This value points to a static buffer whose contents may be changed by any subsequent UltraLite call, so make a copy of the value if you need to retain it.

GetTableSyncType method

Gets the table synchronization type.

Syntax

```
public virtual ul_table_sync_type GetTableSyncType()
```

Returns

The table synchronization type.

Remarks

This method indicates how the table participates in synchronization, and is defined when the table is created with the SYNCHRONIZE constraint clause of the CREATE TABLE statement.

See also

- "ul_table_sync_type enumeration [UltraLite C++]" on page 236

InPublication method

Checks whether the table is contained in the named publication.

Syntax

```
public virtual bool InPublication(const char * pubName)
```

Parameters

- **pubName** The name of the publication.

Returns

True if the table is contained in the publication; otherwise, returns false.

IsColumnInIndex method

Checks whether the column is contained in the named index.

Syntax

```
public virtual bool IsColumnInIndex(
     ul_column_num cid,
    const char * indexName
)
```

Parameters

- **cid** A 1-based ordinal column number.
- **indexName** The name of the index.

Returns

True if the column is contained in the index; otherwise, returns false.

IsColumnNullable method

Checks whether the specified column is nullable.

Syntax

```
public virtual bool IsColumnNullable(ul_column_num cid)
```

Parameters

- **cid** A 1-based ordinal column number.

Returns

True if the column is nullable; otherwise, returns false.

ul_column_default_type enumeration

Identifies a column default type.

Syntax

```
public enum ul_column_default_type
```

Members

Member name	Description
ul_column_default_none	The column has no default value.
ul_column_default_autoincrement	The column default is AUTOINCREMENT.
ul_column_default_global_autoincrement	The column default is GLOBAL AUTOINCREMENT.
ul_column_default_current_timestamp	The column default is CURRENT TIMESTAMP.
ul_column_default_current_utc_timestamp	The column default is CURRENT UTC TIMESTAMP.
ul_column_default_current_time	The column default is CURRENT TIME.
ul_column_default_current_date	The column default is CURRENT DATE.
ul_column_default_newid	The column default is NEWID().
ul_column_default_other	The column default is a user-specified constant.

See also

- "ULTableSchema.GetColumnDefaultType method [UltraLite C++]" on page 228

ul_column_name_type enumeration

Specifies values that control how a column name is retrieved when describing a result set.

Syntax

```
public enum ul_column_name_type
```

Members

Member name	Description
ul_name_type_sql	For SELECT statements, returns the alias or correlation name. For tables, returns the column name.
ul_name_type_sql_column_only	For SELECT statements, returns the alias or correlation name and exclude any table names that were specified. For tables, returns the column name.
ul_name_type_base_table	Returns the underlying table name if it can be determined. If the table does not exist in the database schema, returns an empty string.
ul_name_type_base_column	Returns the underlying column name if it can be determined. If the column does not exist in the database schema, returns an empty string.
ul_name_type_qualified	Returns the underlying qualified column name, if it can be determined, when used in conjunction with the ULResultSetSchema.GetColumnName method. The returned name can be one of the following values, and is determined in this order: 1. The represented correlated table 2. The name of the represented table column 3. The alias name of the column 4. An empty string

Member name	Description
ul_name_type_base	Indicates that a column name qualified with its table name should be returned when used with the GetColumnName method. If the column name being retrieved is associated with a base table in the query, then the base table name is used as the column qualifier (that is, the base_table_name.column_name value is returned). If the column name being retrieved refers to a column in a correlated table in the query, then the correlation name is used as the column qualifier (that is, the correl_table_name.col_name value is returned). If the column has an alias, then the qualified name of the column being aliased is returned; the alias is not part of the qualified name. Otherwise, an empty string is returned.

See also

- "ULResultSetSchema.GetColumnName method [UltraLite C++]" on page 215

ul_index_flag enumeration

Flags (bit fields) which identify properties of an index.

Syntax

```
public enum ul_index_flag
```

Members

Member name	Description
ul_index_flag_primary_key	The index is a primary key.
ul_index_flag_unique_key	The index is a primary key or index created for a unique constraint (nulls not allowed).
ul_index_flag_unique_index	The index was created with the UNIQUE flag (or is a primary key).
ul_index_flag_foreign_key	The index is a foreign key.
ul_index_flag_foreign_key_nullable	The foreign key allows nulls.
ul_index_flag_foreign_key_check_on_commit	Referential integrity checks are performed on commit (rather than on insert/update).

See also

- "ULIndexSchema.GetIndexFlags method [UltraLite C++]" on page 166

ul_table_sync_type enumeration

Identifies a table synchronization type.

Syntax

```
public enum ul_table_sync_type
```

Members

Member name	Description
ul_table_sync_on	All changed rows are synchronized, which is the default behavior. This initializer corresponds to the SYNCHRONIZE ON clause in a CREATE TABLE statement.
ul_table_sync_off	Table is never synchronized. This initializer corresponds to the SYNCHRONIZE OFF clause in a CREATE TABLE statement.
ul_table_sync_upload_all_rows	Always upload every row, including unchanged rows. This initializer corresponds to the SYNCHRONIZE ALL clause in a CREATE TABLE statement.
ul_table_sync_download_only	Changes are never uploaded. This initializer corresponds to the SYNCHRONIZE DOWNLOAD clause in a CREATE TABLE statement.

See also

- "ULTableSchema.GetTableSyncType method [UltraLite C++]" on page 231

UL_BLOB_CONTINUE variable

Used when reading data with the ULResultSet.GetStringChunk or ULResultSet.GetByteChunk methods.

Syntax

```
#define UL_BLOB_CONTINUE
```

Remarks

This value indicates that the chunk of data to be read should continue from where the last chunk was read.

See also

- "ULResultSet.GetStringChunk method [UltraLite C++]" on page 197
- "ULResultSet.GetByteChunk method [UltraLite C++]" on page 186

ul_index_iter_start variable

Used by the GetNextIndex method to initialize index iteration in a table.

Syntax

```
#define ul_index_iter_start
```

See also

- "ULTableSchema.GetNextIndex method [UltraLite C++]" on page 230

ul_publication_iter_start variable

Used by the GetNextPublication method to initialize publication iteration in a database.

Syntax

```
#define ul_publication_iter_start
```

See also

- "ULDatabaseSchema.GetNextPublication method [UltraLite C++]" on page 157

ul_table_iter_start variable

Used by the GetNextTable method to initialize table iteration in a database.

Syntax

```
#define ul_table_iter_start
```

See also

- "ULDatabaseSchema.GetNextTable method [UltraLite C++]" on page 158

UltraLite Embedded SQL API reference

This section lists functions that support UltraLite functionality in embedded SQL applications.

For general information about SQL statements that can be used, see "UltraLite C++ application development using embedded SQL" on page 27.

Use the EXEC SQL INCLUDE SQLCA command to include prototypes for the functions in this chapter.

Header files

- `mlfiletransfer.h`
- `ulprotos.h`

db_fini method

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 method" on page 238

db_init method

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.

Usually 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 method" on page 238

MLFTEnableEccE2ee method

Enables you to specify the ECC end-to-end encryption feature.

Syntax

```
public void MLFTEnableEccE2ee(ml_file_transfer_info * info)
```

Parameters

- **info** A structure containing the file transfer information.

See also

- "ml_file_transfer_info structure [UltraLite Embedded SQL]" on page 278

MLFTEnableEccEncryption method

Enables you to specify the ECC encryption feature.

Syntax

```
public void MLFTEnableEccEncryption(ml_file_transfer_info * info)
```

Parameters

- **info** A structure containing the file transfer information.

See also

- "ml_file_transfer_info structure [UltraLite Embedded SQL]" on page 278

MLFTEnableRsaE2ee method

Enables you to specify the RSA end-to-end encryption feature.

Syntax

```
public void MLFTEnableRsaE2ee(ml_file_transfer_info * info)
```

Parameters

- **info** A structure containing the file transfer information.

See also

- "ml_file_transfer_info structure [UltraLite Embedded SQL]" on page 278

MLFTEnableRsaEncryption method

Enables you to specify the RSA encryption feature.

Syntax

```
public void MLFTEnableRsaEncryption(ml_file_transfer_info * info)
```

Parameters

- **info** A structure containing the file transfer information.

See also

- "ml_file_transfer_info structure [UltraLite Embedded SQL]" on page 278

MLFTEnableRsaFipsE2ee method

Enables you to specify the RSAFIPS end-to-end encryption feature.

Syntax

```
public void MLFTEnableRsaFipsE2ee(ml_file_transfer_info * info)
```

Parameters

- **info** A structure containing the file transfer information.

See also

- "ml_file_transfer_info structure [UltraLite Embedded SQL]" on page 278

MLFTEnableRsaFipsEncryption method

Enables you to specify the RSAFIPS encryption feature.

Syntax

```
public void MLFTEnableRsaFipsEncryption(ml_file_transfer_info * info)
```

Parameters

- **info** A structure containing the file transfer information.

See also

- "ml_file_transfer_info structure [UltraLite Embedded SQL]" on page 278

MLFTEnableZlibCompression method

Enables you to specify the ZLIB compression feature.

Syntax

```
public void MLFTEnableZlibCompression(ml_file_transfer_info * info)
```

Parameters

- **info** A structure containing the file transfer information.

See also

- "ml_file_transfer_info structure [UltraLite Embedded SQL]" on page 278

MLFileDownload method

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

Syntax

```
public bool MLFileDownload(ml_file_transfer_info * info)
```

Parameters

- **info** A structure containing the file transfer information.

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.

See also

- "ml_file_transfer_info structure [UltraLite Embedded SQL]" on page 278

Example

The following example illustrates how to use the MLFileDownload method:

```
ml_file_transfer_info info;
MLInitFileTransferInfo( &info );
MLFTEnableZlibCompression( &info );
info.filename = "myfile";
info.username = "user1";
info.password = "pwd";
info.version = "ver1";
info.stream = "HTTP";
info.stream_parms = "host=myhost.com;compression=zlib";
if( ! MLFileDownload( &info ) ) {
    // file download failed
}
MLFiniFileTransferInfo( &info );
```

MLFileUpload method

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

Syntax

```
public bool MLFileUpload(ml_file_transfer_info * info)
```

Parameters

- **info** A structure containing the file transfer information.

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 upload 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.

See also

- "ml_file_transfer_info structure [UltraLite Embedded SQL]" on page 278

Example

The following example illustrates how to use the MLFileUpload method:


```
ml_file_transfer_info info;
MLInitFileTransferInfo( &info );
MLFTEnableZlibCompression( &info );
info.filename = "myfile";
info.username = "user1";
info.password = "pwd";
info.version = "ver1";
info.stream = "HTTP";
info.stream_parms = "host=myhost.com;compression=zlib";
if( ! MLFileUpload( &info ) ) {
       // file upload failed
}
MLFiniFileTransferInfo( &info );
```

MLFiniFileTransferInfo method

Finalizes any resources allocated in the ml_file_transfer_info structure when it is initialized.

Syntax

```
public void MLFiniFileTransferInfo(ml_file_transfer_info * info)
```

Parameters

- **info** A structure containing the file transfer information.

Remarks

This method should be called after the file upload/download has completed.

See also

- "ml_file_transfer_info structure [UltraLite Embedded SQL]" on page 278

MLInitFileTransferInfo method

Initializes the ml_file_transfer_info structure.

Syntax

```
public bool MLInitFileTransferInfo(ml_file_transfer_info * info)
```

Parameters

- **info** A structure containing the file transfer information.

Remarks

This method should be called before starting the file upload/download.

See also

- "ml_file_transfer_info structure [UltraLite Embedded SQL]" on page 278

ULCancelGetNotification method

Cancels any pending get-notification calls on all queues matching the given name.

Syntax

```
public ul_u_long ULCancelGetNotification(
     SQLCA * sqlca,
    char const * queue_name
)
```

Parameters

- **sqlca** A pointer to the SQLCA.
- **queue_name** The name of the queue.

Returns

The number of affected queues (not the number of blocked reads necessarily).

ULChangeEncryptionKey method

Changes the encryption key for an UltraLite database.

Syntax

```
public ul_bool ULChangeEncryptionKey(
     SQLCA * sqlca,
    char const * new_key
)
```

Parameters

- **sqlca** A pointer to the SQLCA.
- **new_key** The new encryption key.

Remarks

Applications that call this method 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 this method 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.

ULCheckpoint method

Performs a checkpoint operation, flushing any pending committed transactions to the database.

Syntax

```
public ul_ret_void ULCheckpoint(SQLCA * sqlca)
```

Parameters

- **sqlca** A pointer to the SQLCA.

Remarks

Any current transaction is not committed by calling this method. This method is used in conjunction with deferring automatic transaction checkpoints as a performance enhancement.

This method ensures that all pending committed transactions have been written to the database.

ULCountUploadRows method

Counts the number of rows that need to be uploaded for synchronization.

Syntax

```
public ul_u_long ULCountUploadRows(
     SQLCA * sqlca,
    char const * pub_list,
    ul_u_long threshold
)
```

Parameters

- **sqlca** A pointer to the SQL.
- **pub_list** A string containing a comma-separated list of publications to check. An empty string (the UL_SYNC_ALL macro) implies all tables except tables marked as "no sync". A string containing just an asterisk (the 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, the method counts all the rows that need to be synchronized), and 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 method to prompt users to synchronize, or determine when automatic background synchronization should take place.

The following call checks the entire database for the total number of rows to be synchronized:

```
count = ULCountUploadRows( sqlca, UL_SYNC_ALL, 0 );
```

The following call checks the PUB1 and PUB2 publications for a maximum of 1000 rows:

```
count = ULCountUploadRows( sqlca, UL_TEXT("PUB1,PUB2"), 1000 );
```

The following call checks if any rows need to be synchronized in the PUB1 and PUB2 publications:

```
count = ULCountUploadRows( sqlca, UL_TEXT("PUB1,PUB2"), 1 );
```

ULCreateDatabase method

Creates an UltraLite database.

Syntax

```
public ul_bool ULCreateDatabase(
     SQLCA * sqlca,
    char const * connect_parms,
    char const * create_parms,
    void * reserved
)
```

Parameters

- **sqlca** A pointer to the initialized SQLCA.
- **connect_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 can be specified when you connect to a database.
- **create_parms** A semicolon-separated string of creation parameters, a set as keyword=value pairs, such as page_size=2048;obfuscate=yes.
- **reserved** This parameter is reserved for future use.

Returns

ul_true if database was successfully created; otherwise, returns ul_false. Typically ul_false is caused by an invalid file name or denied access.

Remarks

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

The connect_parms parameter is a list of connection parameters that are applicable whenever the database is accessed. Some examples include file name, user ID, password, or optional encryption key.

The create_parms parameter is a list of parameters that are only relevant when creating a database. Some examples include obfuscation, page-size, and time and date format).

Applications can call this method after initializing the SQLCA.

The following code illustrates how to use the ULCreateDatabase method 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
};
```

See also

- "UltraLite connection parameters" [*UltraLite - Database Management and Reference*]
- "UltraLite creation parameters" [*UltraLite - Database Management and Reference*]

ULCreateNotificationQueue method

Creates an event notification queue for this connection.

Syntax

```
public ul_bool ULCreateNotificationQueue(
     SQLCA * sqlca,
    char const * name,
    char const * parameters
)
```

Parameters

- **sqlca** A pointer to the SQLCA.
- **name** The name for the new queue.
- **parameters** Currently unused. Set to NULL.

Returns

True on success; otherwise, returns false.

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 the ULRegisterForEvent method if no queue is specified. This call fails with an error if the name already exists or isn't valid.

ULDeclareEvent method

Declares an event which can then be registered for and triggered.

Syntax

```
public ul_bool ULDeclareEvent(SQLCA * sqlca, char const * event_name)
```

Parameters

- **sqlca** A pointer to the SQLCA.
- **event_name** The name for the new user-defined event.

Returns

True if the event was declared successfully; otherwise, returns 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 ULTriggerEvent method. The event name must be unique. Names are case insensitive.

See also

- "ULTriggerEvent method [UltraLite Embedded SQL]" on page 272

ULDeleteAllRows method

Deletes all rows from a table.

Syntax

```
public ul_ret_void ULDeleteAllRows(SQLCA * sqlca, ul_table_num number)
```

Parameters

- **sqlca** A pointer to the SQLCA.
- **number** The ID of the table to truncate.

Returns

True on success; otherwise, returns False. 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 synchronization property on the connection, the deleted rows are not synchronized.

> **Note**
> 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 the DeleteAllRows method.

If this table has been opened without an index, then it is considered read-only and data cannot be deleted.

ULDestroyNotificationQueue method

Destroys the given event notification queue.

Syntax

```
public ul_bool ULDestroyNotificationQueue(
     SQLCA * sqlca,
    char const * name
)
```

Parameters

- **sqlca** A pointer to the SQLCA.
- **name** The name of the queue to destroy.

Returns

True on success; otherwise, returns false.

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.

ULECCLibraryVersion method

Returns the version number of the ECC encryption library.

Syntax

```
public char const * ULECCLibraryVersion(void)
```

Returns

The version number of the ECC encryption library.

ULEnableAesDBEncryption method

Enables AES database encryption.

Syntax

```
public ul_ret_void ULEnableAesDBEncryption(SQLCA * sqlca)
```

Parameters

- **sqlca** A pointer to the initialized SQLCA.

Remarks

You can use this method in C++ API applications and embedded SQL applications. You must call this method before calling the ULInitDatabaseManager method.

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

ULEnableAesFipsDBEncryption method

Enables FIPS 140-2 certified AES database encryption.

Syntax

```
public ul_ret_void ULEnableAesFipsDBEncryption(SQLCA * sqlca)
```

Parameters

- **sqlca** A pointer to the initialized SQLCA.

Remarks

> **Note**
> Calling this method causes the appropriate routines to be included in the application and increases the size of the application code.

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

> **Note**
> 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 12 - Introduction*].

See also

- “ULEnableAesDBEncryption method [UltraLite Embedded SQL]” on page 249

ULEnableEccE2ee method

Enables ECC end-to-end encryption.

Syntax

```
public ul_ret_void ULEnableEccE2ee(SQLCA * sqlca)
```

Parameters

- **sqlca** A pointer to the SQLCA.

ULEnableEccSyncEncryption method

Enables ECC encryption for SSL or TLS streams.

Syntax

```
public ul_ret_void ULEnableEccSyncEncryption(SQLCA * sqlca)
```

Parameters

- **sqlca** A pointer to the SQLCA.

Remarks

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

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

Note
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 12 - Introduction*].

See also

- "ULEnableZlibSyncCompression method [UltraLite Embedded SQL]" on page 254
- "ULEnableRsaFipsSyncEncryption method [UltraLite Embedded SQL]" on page 252

ULEnableHttpSynchronization method

Enables HTTP synchronization.

Syntax

```
public ul_ret_void ULEnableHttpSynchronization(SQLCA * sqlca)
```

Parameters

- **sqlca** A pointer to the SQLCA.

Remarks

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

ULEnableRsaE2ee method

Enables RSA end-to-end encryption.

Syntax

```
public ul_ret_void ULEnableRsaE2ee(SQLCA * sqlca)
```

Parameters

- **sqlca** A pointer to the SQLCA.

ULEnableRsaFipsE2ee method

Enables FIPS 140-2 certified RSA end-to-end encryption.

Syntax

```
public ul_ret_void ULEnableRsaFipsE2ee(SQLCA * sqlca)
```

Parameters

- **sqlca** A pointer to the SQLCA.

ULEnableRsaFipsSyncEncryption method

Enables RSA FIPS encryption for SSL or TLS streams.

Syntax

```
public ul_ret_void ULEnableRsaFipsSyncEncryption(SQLCA * sqlca)
```

Parameters

- **sqlca** A pointer to the SQLCA.

Remarks

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

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

See also

- "ULEnableRsaSyncEncryption method [UltraLite Embedded SQL]" on page 253
- "ULEnableEccSyncEncryption method [UltraLite Embedded SQL]" on page 251

ULEnableRsaSyncEncryption method

Enables RSA encryption for SSL or TLS streams.

Syntax

```
public ul_ret_void ULEnableRsaSyncEncryption(SQLCA * sqlca)
```

Parameters

- **sqlca** A pointer to the SQLCA.

Remarks

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.

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

See also

- "ULEnableEccSyncEncryption method [UltraLite Embedded SQL]" on page 251
- "ULEnableRsaFipsSyncEncryption method [UltraLite Embedded SQL]" on page 252

ULEnableTcpipSynchronization method

Enables TCP/IP synchronization.

Syntax

```
public ul_ret_void ULEnableTcpipSynchronization(SQLCA * sqlca)
```

Parameters

- **sqlca** A pointer to the SQLCA.

Remarks

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

ULEnableZlibSyncCompression method

Enables ZLIB compression for a synchronization stream.

Syntax

```
public ul_ret_void ULEnableZlibSyncCompression(SQLCA * sqlca)
```

Parameters

- **sqlca** A pointer to the initialized SQLCA.

Remarks

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

ULErrorInfoInitFromSqlca method

Copies the error information from the SQLCA to the ul_error_info object.

Syntax

```
public void ULErrorInfoInitFromSqlca(
     ul_error_info * errinf,
    SQLCA const * sqlca
)
```

Parameters

- **sqlca** A pointer to the SQLCA.
- **errinf** The ul_error_info object.

ULErrorInfoParameterAt method

Retrieves an error parameter by ordinal.

Syntax

```
public size_t ULErrorInfoParameterAt(
     ul_error_info const * errinf,
    ul_u_short parmNo,
```



```
    char * buffer,
    size_t bufferSize
)
```

Parameters

- **errinf** The ul_error_info object.
- **parmNo** The 1-based parameter ordinal.
- **buffer** The buffer to receive parameter string.
- **bufferSize** The size of the buffer.

Returns

The size, in bytes, required to store the parameter, or zero if the ordinal isn't valid. If the return value is larger than the bufferSize value, the parameter was truncated.

ULErrorInfoParameterCount method

Retrieves the number of error parameters.

Syntax

```
public ul_u_short ULErrorInfoParameterCount(
     ul_error_info const * errinf
)
```

Parameters

- **errinf** The ul_error_info object.

Returns

The number of error parameters.

ULErrorInfoString method

Retrieves a description of the error.

Syntax

```
public size_t ULErrorInfoString(
     ul_error_info const * errinf,
    char * buffer,
    size_t bufferSize
)
```

Parameters

- **errinf** The ul_error_info object.

- **buffer** The buffer to receive the error description.
- **bufferSize** The size, in bytes, of the buffer.

Returns

The size, in bytes, required to store the string. If the return value is larger than the len value, the string was truncated.

ULErrorInfoURL method

Retrieves a URL to the documentation page for this error.

Syntax

```
public size_t ULErrorInfoURL(
     ul_error_info const * errinf,
    char * buffer,
    size_t bufferSize,
    char const * reserved
)
```

Parameters

- **errinf** The ul_error_info object.
- **buffer** The buffer to receive the URL.
- **bufferSize** The size, in bytes, of the buffer.
- **reserved** Reserved for future use.

Returns

The size, in bytes, required to store the URL. If the return value is larger than the len value, the URL was truncated.

ULGetDatabaseID method

Gets the current database ID used for global autoincrement.

Syntax

```
public ul_u_long ULGetDatabaseID(SQLCA * sqlca)
```

Parameters

- **sqlca** A pointer to the SQLCA.

Returns

The value set by the last call to the SetDatabaseID method, or UL_INVALID_DATABASE_ID if the ID was never set.

ULGetDatabaseProperty method

Obtains the value of a database property.

Syntax

```
public 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.

ULGetErrorParameter method

Retrieve error parameter via an ordinal parameter number.

Syntax

```
public size_t ULGetErrorParameter(
     SQLCA const * sqlca,
    ul_u_long parm_num,
    char * buffer,
    size_t size
)
```

Parameters

- **sqlca** A pointer to the SQLCA.
- **parm_num** The ordinal parameter number.
- **buffer** A pointer to a buffer that contains the error parameter.

- **size** The size, in bytes, of the buffer.

Returns

This method returns the number of characters copied to the supplied buffer.

See also

- "ULGetErrorParameterCount method [UltraLite Embedded SQL]" on page 258

ULGetErrorParameterCount method

Obtains a count of the number of error parameters.

Syntax

```
public ul_u_long ULGetErrorParameterCount(SQLCA const * sqlca)
```

Parameters

- **sqlca** A pointer to the SQLCA.

Returns

The number of error parameters. Unless the result is zero, values from 1 through this result can be used to call the ULGetErrorParameter method to retrieve the corresponding error parameter value.

See also

- "ULGetErrorParameter method [UltraLite Embedded SQL]" on page 257

ULGetIdentity method

Gets the @identity value.

Syntax

```
public ul_u_big ULGetIdentity(SQLCA * sqlca)
```

Parameters

- **sqlca** A pointer to the SQLCA.

Returns

The last value inserted into an autoincrement or global autoincrement column.

ULGetLastDownloadTime method

Obtains the last time a specified publication was downloaded.

Syntax

```
public ul_bool ULGetLastDownloadTime(
     SQLCA * sqlca,
    char const * 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, the value of January 1, 1990 indicates that the publication has yet to be synchronized.

Returns

True when the value is successfully populated by the last download time of the publication specified by the pub_name value; Otherwise, returns false.

Remarks

The following call populates the dt structure with the date and time that the UL_PUB_PUB1 publication was downloaded:

```
DECL_DATETIME dt;
ret = ULGetLastDownloadTime( &sqlca, UL_TEXT("UL_PUB_PUB1"), &dt );
```

ULGetNotification method

Reads an event notification.

Syntax

```
public ul_bool ULGetNotification(
     SQLCA * sqlca,
    char const * queue_name,
    char * event_name_buf,
    ul_length event_name_buf_len,
    ul_u_long wait_ms
)
```

Parameters

- **sqlca** A pointer to the SQLCA.
- **queue_name** The queue to read or NULL for the default connection queue.
- **event_name_buf** A buffer to hold the name of the event.
- **event_name_buf_len** The size of the buffer in bytes.
- **wait_ms** The time, in milliseconds, to wait (block) before returning.

Returns

True on success; otherwise, returns false.

Remarks

This call blocks until a notification is received or until the given wait period expires. Pass UL_READ_WAIT_INFINITE to the wait_ms parameter to wait indefinitely. To cancel a wait, send another notification to the given queue or use the ULCancelGetNotification method. After reading a notification, use the ULGetNotificationParameter method to retrieve additional parameters by name.

See also

- "ULCancelGetNotification method [UltraLite Embedded SQL]" on page 244
- "ULGetNotificationParameter method [UltraLite Embedded SQL]" on page 260

ULGetNotificationParameter method

Gets a parameter for the event notification just read by the ULGetNotification method.

Syntax

```
public ul_bool ULGetNotificationParameter(
     SQLCA * sqlca,
    char const * queue_name,
    char const * parameter_name,
    char * value_buf,
    ul_length value_buf_len
)
```

Parameters

- **sqlca** A pointer to the SQLCA.
- **queue_name** The queue to read or NULL for default connection queue.
- **parameter_name** The name of the parameter to read (or "*").
- **value_buf** A buffer to hold the parameter value.
- **value_buf_len** The size of the buffer in bytes.

Returns

True on success; otherwise, returns false.

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.

ULGetSyncResult method

Gets the result of the last synchronization.

Syntax

```
public ul_bool ULGetSyncResult(
     SQLCA * sqlca,
    ul_sync_result * sync_result
)
```

Parameters

- **sqlca** A pointer to the SQLCA.
- **sync_result** A pointer to the ul_sync_result structure that holds the synchronization results.

Returns

True on success; otherwise, returns false.

See also

- "ul_sync_result structure [UltraLite C and Embedded SQL datatypes]" on page 112

ULGlobalAutoincUsage method

Obtains the percent of the default values used in all the columns that have global autoincrement defaults.

Syntax

```
public ul_u_short ULGlobalAutoincUsage(SQLCA * sqlca)
```

Parameters

- **sqlca** A pointer to the SQLCA.

Returns

The percent of the global autoincrement values used by the counter.

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 method [UltraLite Embedded SQL]" on page 267

ULGrantConnectTo method

Grants access to an UltraLite database for a new or existing user ID with the given password.

Syntax

```
public ul_ret_void ULGrantConnectTo(
     SQLCA * sqlca,
    char const * uid,
    char const * pwd
)
```

Parameters

- **sqlca** A pointer to the SQLCA.
- **uid** A character array that holds the user ID.
- **pwd** A character array that holds the password for the user ID.

Remarks

This method updates the password for an existing user when you specify an existing user ID.

See also

- "ULRevokeConnectFrom method [UltraLite Embedded SQL]" on page 265

ULInitSyncInfo method

Initializes the synchronization information structure.

Syntax

```
public ul_ret_void ULInitSyncInfo(ul_sync_info * info)
```

Parameters

- **info** A synchronization structure.

ULIsSynchronizeMessage method

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.

Syntax

```
public ul_bool ULIsSynchronizeMessage(ul_u_long number)
```


Remarks

When the processing of a synchronization message is complete, the ULSignalSyncIsComplete method should be called.

You should include a call to this method in the WindowProc function of your application. This applies to Windows Mobile for ActiveSync.

The following code snippet illustrates how to use the ULIsSynchronizeMessage method 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

- "ULSignalSyncIsComplete method [UltraLite Embedded SQL]" on page 269

ULLibraryVersion method

Returns the version number of the UltraLite runtime library.

Syntax

```
public char const * ULLibraryVersion(void)
```

Returns

The version number of the UltraLite runtime library.

ULRSALibraryVersion method

Returns the version number of the RSA encryption library.

Syntax

```
public char const * ULRSALibraryVersion(void)
```

Returns

The version number of the RSA encryption library.

ULRegisterForEvent method

Registers or unregisters a queue to receive notifications of an event.

Syntax

```
public ul_bool ULRegisterForEvent(
     SQLCA * sqlca,
    char const * event_name,
    char const * object_name,
    char const * queue_name,
    ul_bool register_not_unreg
)
```

Parameters

- **sqlca** A pointer to the SQLCA.
- **event_name** The system- or user-defined event to register for.
- **object_name** The object to which the event applies, such as a table name.
- **queue_name** The connection queue name. NULL denotes the 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 you to specify an object name to which the event applies. For example, the TableModified event can specify the table name. Unlike the ULSendNotification method, only the specific queue registered receives notifications of the event. Other queues with the same name on different connections do not receive notifications, unless they are also explicitly 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.

ULResetLastDownloadTime method

Resets the last download time of a publication so that the application resynchronizes previously downloaded data.

Syntax

```
public ul_ret_void ULResetLastDownloadTime(
     SQLCA * sqlca,
    char const * 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 assigns all tables except tables marked as "no sync". A string containing just an asterisk ("*") assigns all publications. Some tables may not be part of any publication and are not included if the pub_list string is "*".

Remarks

The following method call resets the last download time for all tables:

```
ULResetLastDownloadTime( &sqlca, UL_TEXT("*") );
```

ULRevokeConnectFrom method

Revokes access from an UltraLite database for a user ID.

Syntax

```
public ul_ret_void ULRevokeConnectFrom(SQLCA * sqlca, char const * uid)
```

Parameters

- **sqlca** A pointer to the SQLCA.
- **uid** A character array holding the user ID to be excluded from database access.

ULRollbackPartialDownload method

Rolls back the changes from a failed synchronization.

Syntax

```
public ul_ret_void ULRollbackPartialDownload(SQLCA * sqlca)
```

Parameters

- **sqlca** A pointer to the SQLCA.

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), the ULRollbackPartialDownload method rolls back the failed download transaction.

ULSendNotification method

Sends a notification to all queues matching the given name.

Syntax

```
public ul_u_long ULSendNotification(
     SQLCA * sqlca,
    char const * queue_name,
    char const * event_name,
    char const * parameters
)
```

Parameters

- **sqlca** A pointer to the SQLCA.
- **queue_name** The connection queue name. NULL indicates the default connection queue.
- **event_name** The system or user-defined event to register for.
- **parameters** Currently unused. Set to 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 value specifies a semicolon delimited name=value pairs option list. After the notification is read, the parameter values are read with the ULGetNotificationParameter method.

See also

- "ULGetNotificationParameter method [UltraLite Embedded SQL]" on page 260

ULSetDatabaseID method

Sets the database identification number.

Syntax

```
public ul_ret_void ULSetDatabaseID(SQLCA * sqlca, ul_u_long value)
```

Parameters

- **sqlca** A pointer to the SQLCA.
- **value** A positive integer that uniquely identifies a particular database in a replication or synchronization setup.

See also

- "ULGlobalAutoincUsage method [UltraLite Embedded SQL]" on page 261

ULSetDatabaseOptionString method

Sets a database option from a string value.

Syntax

```
public void ULSetDatabaseOptionString(
     SQLCA * sqlca,
    ul_database_option_id id,
    char const * 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 method

Sets a numeric database option.

Syntax

```
public 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.

ULSetErrorCallback method

Sets the callback to be invoked when an error occurs.

Syntax

```
public ul_ret_void ULSetErrorCallback(
     SQLCA * sqlca,
    ul_error_callback_fn_a callback,
    ul_void * user_data,
    char * buffer,
    size_t len
)
```

Parameters

- **sqlca** A pointer to the SQLCA.
- **callback** The callback function.
- **user_data** User context information passed to the callback.
- **buffer** A user-supplied buffer that contains the error parameters when the callback is invoked.
- **len** The size, in bytes, of the buffer.

See also

- "Error handling" on page 22

ULSetSyncInfo method

Creates a synchronization profile using the given name based on the given ul_sync_info structure.

Syntax

```
public ul_bool ULSetSyncInfo(
     SQLCA * sqlca,
    char const * profile_name,
    ul_sync_info * sync_info
)
```

Parameters

- **sqlca** A pointer to the SQLCA.

- **profile_name** The name of the synchronization profile.
- **sync_info** A pointer to the ul_sync_info structure that holds the synchronization parameters.

Returns

True on success; otherwise, returns false.

Remarks

The synchronization profile replaces any previous profile with the same name. The named profile is deleted by specifying a null pointer for the structure.

ULSetSynchronizationCallback method

Sets the callback to be invoked while performing a synchronization.

Syntax

```
public ul_ret_void ULSetSynchronizationCallback(
     SQLCA * sqlca,
    ul_sync_observer_fn callback,
    ul_void * user_data
)
```

Parameters

- **sqlca** A pointer to the SQLCA.
- **callback** The callback.
- **user_data** User context information that is passed to the callback.

ULSignalSyncIsComplete method

Indicates that processing a synchronization message is complete.

Syntax

```
public ul_ret_void ULSignalSyncIsComplete()
```

Remarks

Applications that are registered with the ActiveSync provider need to call this method in their WNDPROC when processing a synchronization message is complete.

ULStartSynchronizationDelete method

Sets START SYNCHRONIZATION DELETE for this connection.

Syntax

```
public ul_ret_void ULStartSynchronizationDelete(SQLCA * sqlca)
```

Parameters

- **sqlca** A pointer to the SQLCA.

Returns

True on success; otherwise, returns false.

ULStaticFini method

Performs finalization of the UltraLite runtime for embedded SQL applications.

Syntax

```
public void ULStaticFini()
```

Remarks

This method should be called once and only once per application, after which no other UltraLite method should be called.

ULStaticInit method

Performs initialization of the UltraLite runtime for embedded SQL applications.

Syntax

```
public void ULStaticInit()
```

Remarks

This method should be called once and only once per application, before any other UltraLite methods have been called.

ULStopSynchronizationDelete method

Sets STOP SYNCHRONIZATION DELETE for this connection.

Syntax

```
public ul_bool ULStopSynchronizationDelete(SQLCA * sqlca)
```

Parameters

- **sqlca** A pointer to the SQLCA

Returns

True on success; otherwise, returns false.

ULSynchronize method

Initiates synchronization in an UltraLite application.

Syntax

```
public ul_ret_void ULSynchronize(SQLCA * sqlca, ul_sync_info * info)
```

Parameters

- **sqlca** A pointer to the SQLCA.
- **info** A pointer to the ul_sync_info structure that holds the synchronization parameters.

Remarks

For TCP/IP or HTTP synchronization, the ULSynchronize method 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 method.

The following example demonstrates database synchronization:

```
ul_sync_info info;
ULInitSyncInfo( &info );
info.user_name = UL_TEXT( "user_name" );
info.version = UL_TEXT( "test" );
ULSynchronize( &sqlca, &info );
```

ULSynchronizeFromProfile method

Synchronizes the database using the given profile and merge parameters.

Syntax

```
public ul_ret_void ULSynchronizeFromProfile(
     SQLCA * sqlca,
    char const * profile_name,
    char const * merge_parms,
    ul_sync_observer_fn observer,
    ul_void * user_data
)
```

Parameters

- **sqlca** A pointer to the SQLCA.
- **profile_name** The name of the profile to synchronize.
- **merge_parms** Merge parameters for the synchronization.

- **observer** Observer callback to send status updates to.
- **user_data** User context data passed to callback.

Remarks

This method is identical to executing the SYNCHRONIZE statement.

See also

- "SYNCHRONIZE statement [UltraLite]" [*UltraLite - Database Management and Reference*]

ULTriggerEvent method

Trigger a user-defined event (and send notification to all registered queues).

Syntax

```
public ul_u_long ULTriggerEvent(
     SQLCA * sqlca,
    char const * event_name,
    char const * parameters
)
```

Parameters

- **sqlca** A pointer for the SQLCA.
- **event_name** The system or user-defined event to register for.
- **parameters** Currently unused. Set to 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 the ULGetNotificationParameter method.

See also

- "ULGetNotificationParameter method [UltraLite Embedded SQL]" on page 260

ULTruncateTable method

Truncates the table and temporarily activates the STOP SYNCHRONIZATION DELETE statement.

Syntax

```
public ul_ret_void ULTruncateTable(SQLCA * sqlca, ul_table_num number)
```


Parameters

- **sqlca** A pointer to the SQLCA.
- **number** The ID of the table to truncate.

Returns

True on success; otherwise, returns false.

ULValidateDatabase method

Validates the database on this connection.

Syntax

```
public ul_bool ULValidateDatabase(
     SQLCA * sqlca,
    char const * start_parms,
    ul_table_num table_id,
    ul_u_short flags,
    ul_validate_callback_fn callback_fn,
    void * user_data
)
```

Parameters

- **sqlca** A pointer to the SQLCA.
- **start_parms** The parameter used to start the database.
- **table_id** The ID of a specific table to validate.
- **flags** Flags controlling the type of validation.
- **callback_fn** The function to receive validation progress information.
- **user_data** User data to send back to the caller via the callback.

Returns

True on success; otherwise, returns false.

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.

The flags parameter is combination of the following values:

- ULVF_TABLE
- ULVF_INDEX

- ULVF_DATABASE
- ULVF_EXPRESS
- ULVF_FULL_VALIDATE

See also

- "ULVF_TABLE variable [UltraLite C and Embedded SQL datatypes]" on page 117
- "ULVF_INDEX variable [UltraLite C and Embedded SQL datatypes]" on page 117
- "ULVF_DATABASE variable [UltraLite C and Embedded SQL datatypes]" on page 115
- "ULVF_EXPRESS variable [UltraLite C and Embedded SQL datatypes]" on page 115
- "ULVF_FULL_VALIDATE variable [UltraLite C and Embedded SQL datatypes]" on page 116

ULValidateDatabaseTableName method

Validates the database on this connection.

Syntax

```
public ul_bool ULValidateDatabaseTableName(
     SQLCA * sqlca,
    char const * start_parms,
    char const * table_name,
    ul_u_short flags,
    ul_validate_callback_fn callback_fn,
    void * user_data
)
```

Parameters

- **sqlca** A pointer to the SQLCA.
- **start_parms** The parameter used to start the database.
- **table_name** The name of a specific table to validate.
- **flags** Flags controlling the type of validation.
- **callback_fn** The function to receive validation progress information.
- **user_data** User data to send back to the caller via the callback.

Returns

True on success; otherwise, returns false.

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.

The flags parameter is combination of the following values:

- ULVF_TABLE
- ULVF_INDEX
- ULVF_DATABASE
- ULVF_EXPRESS
- ULVF_FULL_VALIDATE

See also

- "ULVF_TABLE variable [UltraLite C and Embedded SQL datatypes]" on page 117
- "ULVF_INDEX variable [UltraLite C and Embedded SQL datatypes]" on page 117
- "ULVF_DATABASE variable [UltraLite C and Embedded SQL datatypes]" on page 115
- "ULVF_EXPRESS variable [UltraLite C and Embedded SQL datatypes]" on page 115
- "ULVF_FULL_VALIDATE variable [UltraLite C and Embedded SQL datatypes]" on page 116

ul_database_option_id enumeration

Specifies possible database options that users can set.

Syntax

```
public enum ul_database_option_id
```

Members

Member name	Description
ul_option_global_database_id	The global database ID is set using an unsigned long integer.
ul_option_ml_remote_id	The remote ID is set using a string.
ul_option_commit_flush_timeout	The database commit flush timeout is set as an integer, representing a time threshold measured in milliseconds.
ul_option_commit_flush_count	The database commit flush count is set as integer, representing a commit count threshold.
ul_option_isolation_level	The connection isolation level is set as string. (read_committed/read_uncommitted)

Member name	Description
ul_option_cache_allocation	Set to resize the database file cache. The value is an integer in the range 0 to 100, representing the amount of cache allocated of the minimum to maximum size range.

Remarks

These database options are used with the ULConnection.SetDatabaseOption method.

See also

- "ULConnection.SetDatabaseOption method [UltraLite C++]" on page 140

ul_database_property_id enumeration

Specifies possible database properties that users can retrieve.

Syntax

```
public enum ul_database_property_id
```

Members

Member name	Description
ul_property_date_format	Date format. (date_format)
ul_property_date_order	Date order. (date_order)
ul_property_nearest_century	Nearest century. (nearest_century)
ul_property_precision	Precision. (precision)
ul_property_scale	Scale. (scale)

Member name	Description
ul_property_time_format	Time format. (time_format)
ul_property_timestamp_format	Timestamp format. (timestamp_format)
ul_property_timestamp_increment	Timestamp increment. (timestamp_increment)
ul_property_name	Name. (Name)
ul_property_file	File. (File)
ul_property_encryption	Encryption. (Encryption)
ul_property_global_database_id	Global database ID. (global_database_id)
ul_property_ml_remote_id	Remote ID. (ml_remote_id)
ul_property_char_set	Character set. (CharSet)
ul_property_collation	collation sequence. (Collation)
ul_property_page_size	Page size. (PageSize)
ul_property_case_sensitive	CaseSensitive. (CaseSensitive)

Member name	Description
ul_property_conn_count	Connection count. (ConnCount)
ul_property_max_hash_size	Default maximum index hash. (MaxHashSize)
ul_property_checksum_level	Database checksum level. (ChecksumLevel)
ul_property_checkpoint_count	Database checkpoint count. (CheckpointCount)
ul_property_commit_flush_timeout	Database commit flush timeout. (commit_flush_timeout)
ul_property_commit_flush_count	Database commit flush count. (commit_flush_count)
ul_property_isolation_level	Connection isolation level. (isolation_level)
ul_property_time-stamp_with_time_zone_format	Timestamp with time zone format. (timestamp_with_time_zone_format)
ul_property_cache_allocation	The current database file cache size, as a percentage value of the minimum to maximum range.

Remarks

These properties are used with the ULConnection.GetDatabaseProperty method.

See also

- "ULConnection.GetDatabaseProperty method [UltraLite C++]" on page 130

ml_file_transfer_info structure

A structure containing the parameters to the file upload/download.

Syntax

```
public typedef struct ml_file_transfer_info
```

Members

Member name	Type	Description
auth_parms	const char *	Supplies parameters to authentication parameters in MobiLink events. For more information, see "Additional Parameters synchronization parameter" [*UltraLite - Database Management and Reference*].
auth_status	asa_uint16	Supplies parameters to authentication parameters in MobiLink events. For more information, see "Additional Parameters synchronization parameter" [*UltraLite - Database Management and Reference*].
auth_value	asa_uint32	Reports results of a custom MobiLink user authentication script. The MobiLink server provides this information to the client. For more information, see "Authentication Value synchronization parameter" [*UltraLite - Database Management and Reference*].
enable_resume	bool	If set to true, the MLFileDownload method resumes a previous download that was interrupted due to a communications error or if 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 default is true.
error	mlft_stream_error	Contains information about any error that occurs.
file_auth_code	asa_uint16	Contains the return code of the optional authenticate_file_transfer script on the server.

Member name	Type	Description
filename	const char *	The file name to be transferred from the server running MobiLink. MobiLink searches the username subdirectory first, before defaulting to the root directory. For more information, see "-ftr mlsrv12 option" [*MobiLink - Server Administration*].
local_filename	const char *	The local name for the downloaded file. If this parameter is empty, the value in file name is used.
local_path	const char *	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.
num_auth_parms	asa_uint8	The number of authentication parameters being passed to authentication parameters in MobiLink events. For more information, see "Number of Authentication Parameters parameter" [*UltraLite - Database Management and Reference*].
observer	ml_file_transfer_observer_fn	A callback can be provided to observe file download progress through the 'observer' field. For more details, see description of the callback function that follows.
password	const char *	The password for the MobiLink user name.
remote_key	const char *	The MobiLink remote key.
stream	const char *	The protocol can be one of: TCPIP, TLS, HTTP, or HTTPS. This field is required. For more information, see "Stream Type synchronization parameter" [*UltraLite - Database Management and Reference*].

Member name	Type	Description
stream_parms	const char *	The protocol options for a given stream. For more information, see "Network protocol options for UltraLite synchronization streams" [*UltraLite - Database Management and Reference*].
transfered_file	asa_uint16	1 if the file was successfully transferred, and 0 if an error occurs. An error occurs if the file is already up-to-date when MLFileUpload is invoked. In this case, the function returns true rather than false.
user_data	void *	The application-specific information made available to the synchronization observer. For more information, see "User Data synchronization parameter" [*UltraLite - Database Management and Reference*].
username	const char *	The MobiLink user name. This field is required.
version	const char *	The MobiLink script version. This field is required.

ml_file_transfer_status structure

A structure containing status/progress information while the file upload/download is in progress.

Syntax

```
public typedef struct ml_file_transfer_status
```

Members

Member name	Type	Description
bytes_transfered	asa_uint64	Indicates how much of the file has been downloaded so far, including previous synchronizations, if the download is resumed.
file_size	asa_uint64	Indicates the total size, in bytes, of the file being downloaded.

Member name	Type	Description
flags	asa_uint16	Provides additional information. The MLFT_STATUS_FLAG_IS_BLOCKING value is set when the MLFileDownload method is blocking on a network call and the download status has not changed since the last time the observer method was called.
info	ml_file_transfer_info *	Points to the information object passed to the MLFileDownload method. You can access the user_data parameter through this pointer.
resumed_at_size	asa_uint64	Used with download resumption and indicates at what point the current download resumed.
stop	asa_uint8	Set to true to cancel the current download. You can resume the download in a subsequent call to the MLFileDownload method, but only if you have set the enable_resume parameter.

mlft_stream_error structure

A structure containing status/progress information while the file upload or download is in progress.

Syntax

```
public typedef struct mlft_stream_error
```

Members

Member name	Type	Description
error_string	char	A string with additional information, if available, for the stream_error_code value.
stream_error_code	ss_error_code	The specific stream error. For a list of possible values, see the ss_error_code enumeration in the *%SQLANY12%\SDK\Include\sserror.h* header file.
system_error_code	asa_int32	A system-specific error code.

See also

- "MobiLink communication error messages sorted by error code" [*Error Messages*]

mlft_stream_error_w structure

A structure containing status/progress information while the file upload or download is in progress.

Syntax

```
public typedef struct mlft_stream_error_w
```

Members

Member name	Type	Description
error_string	wchar_t	A string with additional information, if available, for the stream_error_code value.
stream_error_code	ss_error_code	The specific stream error. For a list of possible values, see the ss_error_code enumeration in the *%SQLANY12%\SDK\Include\sserror.h* header file.
system_error_code	asa_int32	A system-specific error code.

Remarks

Note
This structure prototype is used internally when you refer to the mlft_stream_error structure and #define the UNICODE macro on Win32 platforms. Typically, you would not reference this structure directly when creating an UltraLite application.

See also

- "mlft_stream_error structure [UltraLite Embedded SQL]" on page 282
- "MobiLink communication error messages sorted by error code" [*Error Messages*]

MLFT_STATUS_FLAG_IS_BLOCKING variable

Defines a bit set in the ml_file_transfer_status.flags field to indicate that the file transfer is blocked awaiting a response from the MobiLink server.

Syntax

```
#define MLFT_STATUS_FLAG_IS_BLOCKING
```

Remarks

Identical file transfer progress messages are generated periodically while this is the case.

Index

Symbols

#define
  UltraLite applications, 101
16-bit signed integer UltraLite embedded SQL data type
  about, 33
32-bit signed integer UltraLite embedded SQL data type
  about, 33
4-byte floating-point UltraLite embedded SQL data type
  about, 33
8-byte floating-point UltraLite embedded SQL data type
  about, 33

A

accessing data
  UltraLite C++ API, 15
accessing schema information
  UltraLite C++ about, 21
ActiveSync
  class names, 58
  UltraLite MFC requirements, 61
  UltraLite synchronization for Windows Mobile, 60
  UltraLite versions for Windows Mobile, 60
  UltraLite Windows Mobile applications, 60
  WindowProc function, 61
AES encryption algorithm
  UltraLite embedded SQL databases, 46
AfterLast method
  ULResultSet class [UltraLite C++ API], 180
Android
  disconnecting from a database, 23
AppendByteChunk method
  ULResultSet class [UltraLite C++ API], 180
AppendParameterByteChunk method
  ULPreparedStatement class [UltraLite C++ API], 169
AppendParameterStringChunk method
  ULPreparedStatement class [UltraLite C++ API], 170
AppendStringChunk method
  ULResultSet class [UltraLite C++ API], 181
applications
  building UltraLite embedded SQL, 54
  compiling UltraLite embedded SQL, 54
  developing for iOS, 5
  preprocessing UltraLite embedded SQL, 54
  writing UltraLite embedded SQL, 27
architectures
  UltraLite C/C++, 1
AutoCommit mode
  UltraLite C++ development, 21

B

BeforeFirst method
  ULResultSet class [UltraLite C++ API], 183
binary UltraLite embedded SQL data type
  about, 34
BlackBerry
  disconnecting from a database, 23
build processes
  embedded SQL applications, 54
  UltraLite embedded SQL applications, 54
building
  UltraLite embedded SQL applications, 54

C

CancelGetNotification method
  ULConnection class [UltraLite C++ API], 122
canceling monitoring
  in UltraLite embedded SQL, 50
canceling synchronization
  in UltraLite embedded SQL, 50
casting
  UltraLite C++ API data types in, 20
ChangeEncryptionKey method
  ULConnection class [UltraLite C++ API], 122
changeEncryptionKey method
  UltraLite embedded SQL, 47
character string UltraLite embedded SQL data type
  fixed length, 34
  variable length, 34
Checkpoint method
  ULConnection class [UltraLite C++ API], 123
class names
  ActiveSync synchronization, 58
Clear method
  ULError class [UltraLite C++ API], 160
Close method

ULConnection class [UltraLite C++ API], 123
ULDatabaseSchema class [UltraLite C++ API], 157
ULIndexSchema class [UltraLite C++ API], 165
ULPreparedStatement class [UltraLite C++ API], 170
ULResultSet class [UltraLite C++ API], 183
ULTableSchema class [UltraLite C++ API], 228
CLOSE statement
UltraLite embedded SQL, 43
columns
UltraLite C++ API modification of values , 20
UltraLite C++ API retrieval of values , 19
Commit method
ULConnection class [UltraLite C++ API], 123
UltraLite C++ transactions, 21
committing
UltraLite C++ transactions, 21
UltraLite changes with embedded SQL, 49
communications errors
UltraLite embedded SQL, 50
compile options
UltraLite applications for Windows Mobile, 26
compiler directives
UltraLite applications, 101
UNDER_CE, 102
compiler options
UltraLite C++ development, 26
compilers
UltraLite applications for Windows Mobile, 26
compiling
UltraLite applications for Windows Mobile, 26
UltraLite embedded SQL applications, 54
configuring
development tools for UltraLite embedded SQL, 55
CONNECT statement
UltraLite embedded SQL, 31
connecting
UltraLite databases, 7
Connection object
UltraLite C++, 7
connections
UltraLite embedded SQL, 31
CountUploadRows method
ULConnection class [UltraLite C++ API], 124
CreateDatabase method
ULDatabaseManager class [UltraLite C++ API], 147
CreateNotificationQueue method
ULConnection class [UltraLite C++ API], 125
cursors
UltraLite fetching multiple rows, 43
UltraLite order of rows, 44
UltraLite positioning, 44
UltraLite positioning after updates, 45
UltraLite repositioning, 45
CustDB application
UltraLite building for Windows Mobile, 57

D

data manipulation
UltraLite C++ API, 15
data modification
UltraLite C++ with SQL, 8
data types
UltraLite C++ API accessing and casting, 19
UltraLite embedded SQL, 32
database files
UltraLite encrypting and obfuscating (embedded SQL), 46
UltraLite for Windows Mobile, 58
database schemas
UltraLite C++ API access, 21
DatabaseManager object
UltraLite C++, 7
db_fini method [UltraLite Embedded SQL API]
description, 238
db_init method [UltraLite Embedded SQL API]
description, 238
decimal UltraLite embedded SQL data type
about, 33
DECL_BINARY macro
UltraLite embedded SQL, 33
DECL_DATETIME macro
UltraLite embedded SQL, 33
DECL_DECIMAL macro
UltraLite embedded SQL, 33
DECL_FIXCHAR macro
UltraLite embedded SQL, 33
DECL_VARCHAR macro
UltraLite embedded SQL, 33
declaration section
UltraLite embedded SQL declaration of, 32
DECLARE statement
UltraLite embedded SQL, 43

DeclareEvent method
ULConnection class [UltraLite C++ API], 125
declaring
UltraLite host variables, 32
Delete method
ULResultSet class [UltraLite C++ API], 183
DeleteAllRows method
ULTable class [UltraLite C++ API], 221
DeleteNamed method
ULResultSet class [UltraLite C++ API], 183
deleting
UltraLite C++ API table rows, 20
dependencies
UltraLite embedded SQL, 55
deploying
in-process version of UltraLite, 26
UltraLite for Windows Mobile, 26
DestroyNotificationQueue method
ULConnection class [UltraLite C++ API], 126
development
UltraLite C++, 5
development platforms
UltraLite C++, 1
development process
UltraLite embedded SQL, 2
development tools
UltraLite embedded SQL, 55
directives
UltraLite applications, 101
DML
UltraLite C++, 9
DropDatabase method
ULDatabaseManager class [UltraLite C++ API], 148
DT_LONGBINARY UltraLite embedded SQL data type
about, 35
DT_LONGVARCHAR UltraLite embedded SQL data type
about, 35
dynamic libraries
UltraLite C++ applications, 26

E

E2EE private key
iPhone, 6
Mac OS X, 6
e2ee_public_key
iPhone, 6
Mac OS X, 6
emulator
UltraLite for Windows Mobile, 26
EnableAesDBEncryption method
ULDatabaseManager class [UltraLite C++ API], 148
EnableAesFipsDBEncryption method
ULDatabaseManager class [UltraLite C++ API], 148
EnableEccE2ee method
ULDatabaseManager class [UltraLite C++ API], 149
EnableEccSyncEncryption method
ULDatabaseManager class [UltraLite C++ API], 149
EnableHttpsSynchronization method
ULDatabaseManager class [UltraLite C++ API], 150
EnableHttpSynchronization method
ULDatabaseManager class [UltraLite C++ API], 150
EnableRsaE2ee method
ULDatabaseManager class [UltraLite C++ API], 151
EnableRsaFipsE2ee method
ULDatabaseManager class [UltraLite C++ API], 151
EnableRsaFipsSyncEncryption method
ULDatabaseManager class [UltraLite C++ API], 151
EnableRsaSyncEncryption method
ULDatabaseManager class [UltraLite C++ API], 152
EnableTcpipSynchronization method
ULDatabaseManager class [UltraLite C++ API], 152
EnableTlsSynchronization method
ULDatabaseManager class [UltraLite C++ API], 153
EnableZlibSyncCompression method
ULDatabaseManager class [UltraLite C++ API], 153
encryption
changing keys in UltraLite embedded SQL, 47
iPhone, 6
Mac OS X, 6

PEM encoded X509 certificate, 6
UltraLite databases using embedded SQL, 46
UltraLite embedded SQL databases, 46
end-to-end encryption
iPhone, 6
Mac OS X, 6
environment variables
INCLUDE, 65
error handling
UltraLite C++, 22
errors
UltraLite C++ API handling , 22
UltraLite codes, 30
UltraLite embedded SQL communications errors, 50
UltraLite SQLCODE, 30
UltraLite sqlcode SQLCA field, 30
EXEC SQL
UltraLite embedded SQL development, 29
ExecuteQuery method
ULPreparedStatement class [UltraLite C++ API], 170
ExecuteScalar method
ULConnection class [UltraLite C++ API], 126
ExecuteScalarV method
ULConnection class [UltraLite C++ API], 128
ExecuteStatement method
ULConnection class [UltraLite C++ API], 129
ULPreparedStatement class [UltraLite C++ API], 171

F

FETCH statement
UltraLite embedded SQL multi-row queries, 43
UltraLite embedded SQL single-row queries, 42
fetching
UltraLite embedded SQL, 42
Find method
ULTable class [UltraLite C++ API], 221
find methods
UltraLite C++, 18
find mode
UltraLite C++, 16
FindBegin method
ULTable class [UltraLite C++ API], 222
FindFirst method
ULTable class [UltraLite C++ API], 222
FindLast method
ULTable class [UltraLite C++ API], 223
FindNext method
ULTable class [UltraLite C++ API], 223
FindPrevious method
ULTable class [UltraLite C++ API], 223
Fini method
ULDatabaseManager class [UltraLite C++ API], 153
First method
ULResultSet class [UltraLite C++ API], 184
functions
UltraLite embedded SQL, 237

G

GetBinary method
ULResultSet class [UltraLite C++ API], 184
GetBinaryLength method
ULResultSet class [UltraLite C++ API], 185
GetByteChunk method
ULResultSet class [UltraLite C++ API], 186
GetChildObjectCount method
ULConnection class [UltraLite C++ API], 130
GetColumnCount method
ULIndexSchema class [UltraLite C++ API], 165
ULResultSetSchema class [UltraLite C++ API], 214
GetColumnDefault method
ULTableSchema class [UltraLite C++ API], 228
GetColumnDefaultType method
ULTableSchema class [UltraLite C++ API], 228
GetColumnID method
ULResultSetSchema class [UltraLite C++ API], 214
GetColumnName method
ULIndexSchema class [UltraLite C++ API], 165
ULResultSetSchema class [UltraLite C++ API], 215
GetColumnPrecision method
ULResultSetSchema class [UltraLite C++ API], 215
GetColumnScale method
ULResultSetSchema class [UltraLite C++ API], 216
GetColumnSize method
ULResultSetSchema class [UltraLite C++ API], 216

GetColumnSQLType method
ULResultSetSchema class [UltraLite C++ API], 217
GetColumnType method
ULResultSetSchema class [UltraLite C++ API], 217
GetConnection method
ULDatabaseSchema class [UltraLite C++ API], 157
ULIndexSchema class [UltraLite C++ API], 166
ULPreparedStatement class [UltraLite C++ API], 171
ULResultSet class [UltraLite C++ API], 188
ULResultSetSchema class [UltraLite C++ API], 217
GetDatabaseProperty method
ULConnection class [UltraLite C++ API], 130
GetDatabasePropertyInt method
ULConnection class [UltraLite C++ API], 130
GetDatabaseSchema method
ULConnection class [UltraLite C++ API], 131
GetDateTime method
ULResultSet class [UltraLite C++ API], 188
GetDouble method
ULResultSet class [UltraLite C++ API], 189
GetErrorInfo method
ULError class [UltraLite C++ API], 160
GetFloat method
ULResultSet class [UltraLite C++ API], 190
GetGlobalAutoincPartitionSize method
ULTableSchema class [UltraLite C++ API], 229
GetGuid method
ULResultSet class [UltraLite C++ API], 191
GetIndexColumnID method
ULIndexSchema class [UltraLite C++ API], 166
GetIndexCount method
ULTableSchema class [UltraLite C++ API], 229
GetIndexFlags method
ULIndexSchema class [UltraLite C++ API], 166
GetIndexSchema method
ULTableSchema class [UltraLite C++ API], 229
GetInt method
ULResultSet class [UltraLite C++ API], 192
GetIntWithType method
ULResultSet class [UltraLite C++ API], 193
GetLastDownloadTime method
ULConnection class [UltraLite C++ API], 131
GetLastError method
ULConnection class [UltraLite C++ API], 132
GetLastIdentity method
ULConnection class [UltraLite C++ API], 132
GetName method
ULIndexSchema class [UltraLite C++ API], 166
ULTableSchema class [UltraLite C++ API], 230
GetNextIndex method
ULTableSchema class [UltraLite C++ API], 230
GetNextPublication method
ULDatabaseSchema class [UltraLite C++ API], 157
GetNextTable method
ULDatabaseSchema class [UltraLite C++ API], 158
GetNotification method
ULConnection class [UltraLite C++ API], 133
GetNotificationParameter method
ULConnection class [UltraLite C++ API], 133
GetOptimalIndex method
ULTableSchema class [UltraLite C++ API], 230
GetParameter method
ULError class [UltraLite C++ API], 161
GetParameterCount method
ULError class [UltraLite C++ API], 162
ULPreparedStatement class [UltraLite C++ API], 171
GetParameterID method
ULPreparedStatement class [UltraLite C++ API], 171
GetParameterType method
ULPreparedStatement class [UltraLite C++ API], 172
GetPlan method
ULPreparedStatement class [UltraLite C++ API], 172
GetPrimaryKey method
ULTableSchema class [UltraLite C++ API], 231
GetPublicationCount method
ULDatabaseSchema class [UltraLite C++ API], 158
GetPublicationPredicate method
ULTableSchema class [UltraLite C++ API], 231
GetReferencedIndexName method
ULIndexSchema class [UltraLite C++ API], 167
GetReferencedTableName method
ULIndexSchema class [UltraLite C++ API], 167
GetResultSetSchema method

ULPreparedStatement class [UltraLite C++ API], 173
ULResultSet class [UltraLite C++ API], 195
GetRowCount method
ULResultSet class [UltraLite C++ API], 195
GetRowsAffectedCount method
ULPreparedStatement class [UltraLite C++ API], 173
GetSqlca method
ULConnection class [UltraLite C++ API], 134
GetSQLCode method
ULError class [UltraLite C++ API], 162
GetSQLCount method
ULError class [UltraLite C++ API], 162
GetState method
ULResultSet class [UltraLite C++ API], 195
GetString method
ULError class [UltraLite C++ API], 163
ULResultSet class [UltraLite C++ API], 196
GetStringChunk method
ULResultSet class [UltraLite C++ API], 197
GetStringLength method
ULResultSet class [UltraLite C++ API], 199
GetSyncResult method
ULConnection class [UltraLite C++ API], 134
GetTableCount method
ULDatabaseSchema class [UltraLite C++ API], 159
GetTableName method
ULIndexSchema class [UltraLite C++ API], 167
GetTableSchema method
ULDatabaseSchema class [UltraLite C++ API], 159
ULTable class [UltraLite C++ API], 224
GetTableSyncType method
ULTableSchema class [UltraLite C++ API], 231
GetURL method
ULError class [UltraLite C++ API], 163
GetUserPointer method
ULConnection class [UltraLite C++ API], 134
GlobalAutoincUsage method
ULConnection class [UltraLite C++ API], 135
GrantConnectTo method
ULConnection class [UltraLite C++ API], 135

H

HasResultSet method
ULPreparedStatement class [UltraLite C++ API], 173
host variables
UltraLite embedded SQL, 32
UltraLite embedded SQL expressions, 37
UltraLite scope, 36
UltraLite usage, 36

I

import libraries
UltraLite C++, 26
INCLUDE statement
UltraLite SQLCA, 29
indexes
UltraLite C++ API schema information in , 22
indicator variables
UltraLite embedded SQL, 40
UltraLite NULL, 41
Init method
ULDatabaseManager class [UltraLite C++ API], 154
InitSyncInfo method
ULConnection class [UltraLite C++ API], 136
InPublication method
ULTableSchema class [UltraLite C++ API], 232
Insert method
ULTable class [UltraLite C++ API], 224
insert mode
UltraLite C++, 16
InsertBegin method
ULTable class [UltraLite C++ API], 224
inserting
UltraLite C++ API table rows, 17
installing
UltraLite Windows Mobile applications, 56
iOS
about UltraLite C++ applications, 1
creating and connecting to a database, 7
database schemas, 21
developing applications, 5
handling errors, 22
UltraLite C++ development, 26
iPhone
access databases, 78
adding data to databases, 81
adding databases to applications, 78
build settings for UltraLite C++, 5

compiling UltraLite library for, 76
creating an application project, 76
data model class, 78
DataAccess singleton, 78
deleting data from databases, 87
displaying data, 85
frameworks, 78
include files for UltraLite C++, 5
interface builder, 81
MobiLink synchronization model, 88
ODBC data source, 88
open connection, 78
root view controller setup, 81
swipe-to-delete, 87
synchronization, 88
thread management, 93
tutorial, 75
UltraLite runtime libraries for, 26
view controller, 81
IsAliased method
ULResultSetSchema class [UltraLite C++ API], 218
IsColumnDescending method
ULIndexSchema class [UltraLite C++ API], 168
IsColumnInIndex method
ULTableSchema class [UltraLite C++ API], 232
IsColumnNullable method
ULTableSchema class [UltraLite C++ API], 233
IsNull method
ULResultSet class [UltraLite C++ API], 200
IsOK method
ULError class [UltraLite C++ API], 164

L

Last method
ULResultSet class [UltraLite C++ API], 201
libraries
UltraLite applications for Windows Mobile, 26
UltraLite compiling and linking in C++, 26
UltraLite DLL for Windows Mobile, 26
UltraLite linking example in C++, 66
library functions
UltraLite embedded SQL, 237
libulrt.lib
UltraLite C++ development, 25
linking
UltraLite applications for Windows Mobile, 26
UltraLite C++ applications, 26
Linux
UltraLite runtime libraries for, 25
Lookup method
ULTable class [UltraLite C++ API], 224
lookup methods
UltraLite C++, 18
lookup mode
UltraLite C++, 16
LookupBackward method
ULTable class [UltraLite C++ API], 225
LookupBegin method
ULTable class [UltraLite C++ API], 225
LookupForward method
ULTable class [UltraLite C++ API], 226

M

Mac OS X
about UltraLite C++ applications, 1
include files for UltraLite C++, 5
UltraLite C++ development, 26
UltraLite runtime libraries for, 26
macros
UL_USE_DLL, 101
UltraLite applications, 101
makefiles
UltraLite embedded SQL, 55
managing
UltraLite C++ transactions, 21
MFC
UltraLite applications ActiveSync requirements, 61
MFC applications
UltraLite for Windows Mobile, 58
ml_file_transfer_info structure [UltraLite Embedded SQL API]
description, 278
ml_file_transfer_status structure [UltraLite Embedded SQL API]
description, 281
MLFileDownload method [UltraLite Embedded SQL API]
description, 241
mlfiletransfer.h header file
UltraLite Embedded SQL API, 237
MLFileUpload method [UltraLite Embedded SQL API]
description, 242

MLFiniFileTransferInfo method [UltraLite Embedded SQL API]
description, 243
mlft_stream_error structure [UltraLite Embedded SQL API]
description, 282
mlft_stream_error_w structure [UltraLite Embedded SQL API]
description, 283
MLFTEnableEccE2ee method [UltraLite Embedded SQL API]
description, 239
MLFTEnableEccEncryption method [UltraLite Embedded SQL API]
description, 239
MLFTEnableRsaE2ee method [UltraLite Embedded SQL API]
description, 240
MLFTEnableRsaEncryption method [UltraLite Embedded SQL API]
description, 240
MLFTEnableRsaFipsE2ee method [UltraLite Embedded SQL API]
description, 240
MLFTEnableRsaFipsEncryption method [UltraLite Embedded SQL API]
description, 241
MLFTEnableZlibCompression method [UltraLite Embedded SQL API]
description, 241
MLInitFileTransferInfo method [UltraLite Embedded SQL API]
description, 243
MobiLink synchronization
iPhone, 88
modes
UltraLite C++, 16
multi-row queries
UltraLite cursors, 43
multi-threaded applications
UltraLite C++, 8
UltraLite embedded SQL, 31

N

namespaces
UltraLite C++ example, 66
navigating
UltraLite C++ API, 15
navigating SQL result sets
UltraLite C++, 14
network protocols
UltraLite for Windows Mobile, 63
Next method
ULResultSet class [UltraLite C++ API], 201
next method
UltraLite C++ data retrieval example, 13
NULL
UltraLite indicator variables, 40
Null terminated string UltraLite embedded SQL data type
about, 33
null-terminated TCHAR character string UltraLite SQL data type
about, 34
null-terminated UNICODE character string UltraLite SQL data type
about, 34
null-terminated WCHAR character string UltraLite SQL data type
about, 34
null-terminated wide character string UltraLite SQL data type
about, 34

O

obfuscating
UltraLite embedded SQL databases, 46
obfuscation
UltraLite databases using embedded SQL, 47
UltraLite embedded SQL databases, 46
observer synchronization parameter
UltraLite embedded SQL example, 53
ODBC data source
iPhone, 88
offsets
UltraLite C++ relative, 15
OPEN statement
UltraLite embedded SQL, 43
OpenConnection method
ULDatabaseManager class [UltraLite C++ API], 154
OpenTable method
ULConnection class [UltraLite C++ API], 136

P

packed decimal UltraLite embedded SQL data type
about, 33
PEM encoded X509 certificate
iPhone, 6
Mac OS X, 6
performance
UltraLite using DLL for economical memory use, 26
UltraLite using INSERT statements , 50
permissions
UltraLite embedded SQL, 29
persistent storage
UltraLite for Windows Mobile, 58
platform requirements
UltraLite for Windows Mobile, 56
platforms
supported in UltraLite C++, 1
prepared statements
UltraLite C++, 9
PrepareStatement method
ULConnection class [UltraLite C++ API], 136
preprocessing
UltraLite embedded SQL applications, 54
UltraLite embedded SQL development tool settings, 55
Previous method
ULResultSet class [UltraLite C++ API], 201
previous method
UltraLite C++ data retrieval example, 13
program structure
UltraLite embedded SQL, 29
protocols
UltraLite for Windows Mobile, 63

Q

queries
UltraLite embedded SQL multi-row queries, 43
UltraLite embedded SQL single-row queries, 42

R

RegisterForEvent method
ULConnection class [UltraLite C++ API], 137
Relative method
ULResultSet class [UltraLite C++ API], 202
relative offset
UltraLite C++ API, 15
ResetLastDownloadTime method
ULConnection class [UltraLite C++ API], 138
result set schemas
UltraLite C++, 14
result sets
UltraLite C++ API schema information in , 21
UltraLite C++ navigation of, 14
RevokeConnectFrom method
ULConnection class [UltraLite C++ API], 138
Rollback method
ULConnection class [UltraLite C++ API], 138
UltraLite C++ transactions, 21
RollbackPartialDownload method
ULConnection class [UltraLite C++ API], 139
rollbacks
UltraLite C++ transactions, 21
rows
accessing in UltraLite C++ API tutorial, 70
UltraLite C++ table access of current, 19
UltraLite C++ table navigation, 15
UltraLite deletions with C++ API, 20
UltraLite insertions with C++ API, 17
UltraLite updates with C++ API, 17
runtime libraries
UltraLite applications for Windows Mobile, 26
UltraLite C++, 26
UltraLite for C++, 26
runtime library
Windows Mobile, 101

S

sample applications
UltraLite building for Windows Mobile, 57
schemas
UltraLite C++ API access, 21
UltraLite C++ API schema information in , 21
scrolling
UltraLite C++ API, 15
searching
UltraLite rows with C++, 18
security
changing the encryption key in UltraLite embedded SQL, 47
obfuscation in UltraLite embedded SQL, 46
UltraLite database encryption, 46
SELECT statement
UltraLite C++ data retrieval example, 13

UltraLite embedded SQL single row, 42
selecting data from database tables
UltraLite C++, 13
SendNotification method
ULConnection class [UltraLite C++ API], 139
SET CONNECTION statement
multiple connections in UltraLite embedded SQL, 31
SetBinary method
ULResultSet class [UltraLite C++ API], 202
SetDatabaseOption method
ULConnection class [UltraLite C++ API], 140
SetDatabaseOptionInt method
ULConnection class [UltraLite C++ API], 140
SetDateTime method
ULResultSet class [UltraLite C++ API], 203
SetDefault method
ULResultSet class [UltraLite C++ API], 204
SetDouble method
ULResultSet class [UltraLite C++ API], 205
SetErrorCallback method
ULDatabaseManager class [UltraLite C++ API], 155
SetFloat method
ULResultSet class [UltraLite C++ API], 206
SetGuid method
ULResultSet class [UltraLite C++ API], 207
SetInt method
ULResultSet class [UltraLite C++ API], 208
SetIntWithType method
ULResultSet class [UltraLite C++ API], 209
SetNull method
ULResultSet class [UltraLite C++ API], 211
SetParameterBinary method
ULPreparedStatement class [UltraLite C++ API], 173
SetParameterDateTime method
ULPreparedStatement class [UltraLite C++ API], 174
SetParameterDouble method
ULPreparedStatement class [UltraLite C++ API], 174
SetParameterFloat method
ULPreparedStatement class [UltraLite C++ API], 175
SetParameterGuid method
ULPreparedStatement class [UltraLite C++ API], 175
SetParameterInt method
ULPreparedStatement class [UltraLite C++ API], 175
SetParameterIntWithType method
ULPreparedStatement class [UltraLite C++ API], 176
SetParameterNull method
ULPreparedStatement class [UltraLite C++ API], 177
SetParameterString method
ULPreparedStatement class [UltraLite C++ API], 177
SetString method
ULResultSet class [UltraLite C++ API], 212
SetSynchronizationCallback method
ULConnection class [UltraLite C++ API], 141
SetSyncInfo method
ULConnection class [UltraLite C++ API], 141
SetUserPointer method
ULConnection class [UltraLite C++ API], 141
simple encryption
UltraLite database simple encryption, 47
SQL Communications Area
UltraLite embedded SQL, 29
SQL preprocessor utility (sqlpp)
UltraLite embedded SQL applications, 54
SQLCA
UltraLite embedded SQL, 29
UltraLite embedded SQL multiple SQLCA, 31
UltraLite fields, 30
sqlcabc SQLCA field
UltraLite embedded SQL, 30
sqlcaid SQLCA field
UltraLite embedded SQL, 30
SQLCODE
UltraLite C++ error handling, 22
sqlcode SQLCA field
UltraLite embedded SQL, 30
sqlerrd SQLCA field
UltraLite embedded SQL, 30
sqlerrmc SQLCA field
UltraLite embedded SQL, 30
sqlerrml SQLCA field
UltraLite embedded SQL, 30
sqlerrp SQLCA field
UltraLite embedded SQL, 30
sqlpp utility
UltraLite embedded SQL applications, 54

UltraLite usage, 54
sqlstate SQLCA field
UltraLite embedded SQL, 30
sqlwarn SQLCA field
UltraLite embedded SQL, 30
StartSynchronizationDelete method
ULConnection class [UltraLite C++ API], 142
static libraries
UltraLite C++ applications, 26
StopSynchronizationDelete method
ULConnection class [UltraLite C++ API], 142
string UltraLite embedded SQL data type
about, 33
fixed length, 34
variable length, 34
strong encryption
UltraLite embedded SQL, 46
supported platforms
UltraLite C++, 1
swipe-to-delete
iPhone, 87
synchronization
adding to UltraLite embedded SQL applications, 48
canceling in UltraLite embedded SQL, 50
for UltraLite for Windows Mobile menu control, 63
initial in UltraLite embedded SQL, 50
invoking in UltraLite embedded SQL, 49
iPhone, 88
monitoring in UltraLite embedded SQL, 50
UltraLite C++ API tutorial, 71
UltraLite embedded SQL, 48
UltraLite embedded SQL committing changes in , 49
UltraLite embedded SQL example, 49
UltraLite for Windows Mobile introduction, 60
synchronization errors
UltraLite embedded SQL communications errors , 50
Synchronize method
ULConnection class [UltraLite C++ API], 142
SynchronizeFromProfile method
ULConnection class [UltraLite C++ API], 143
synchronizing
iPhone, 88
UltraLite C++, 23

T

tables
UltraLite C++ API schema information in , 21
target platforms
UltraLite C++, 1
threads
UltraLite C++ API multi-threaded applications, 8
UltraLite embedded SQL, 31
timestamp structure UltraLite embedded SQL data type
about, 34
tips
UltraLite development, 50
transaction processing
UltraLite C++ management, 21
transactions
committing in UltraLite with embedded SQL, 49
UltraLite C++ management, 21
TriggerEvent method
ULConnection class [UltraLite C++ API], 144
troubleshooting
UltraLite C++ handling errors, 22
UltraLite development, 50
UltraLite synchronization with embedded SQL, 49
UltraLite using reference expressions in SQL preprocessor, 37
TruncateTable method
ULTable class [UltraLite C++ API], 226
truncation
UltraLite FETCH, 41
tutorials
building an iPhone application, 75
UltraLite C++ API, 65
UltraLite C/C++ API, 75

U

ul_binary structure [UltraLite C and Embedded SQL datatypes API]
description, 109
ul_column_default_type enumeration [UltraLite C++ API]
description, 233
ul_column_name_type enumeration [UltraLite C++ API]
description, 234
ul_column_sql_type enumeration [UltraLite C and Embedded SQL datatypes API]

description, 102
ul_column_storage_type enumeration [UltraLite C and Embedded SQL datatypes API]
description, 104
ul_database_option_id enumeration [UltraLite Embedded SQL API]
description, 275
ul_database_property_id enumeration [UltraLite Embedded SQL API]
description, 276
ul_error_action enumeration [UltraLite C and Embedded SQL datatypes API]
description, 105
ul_error_info structure [UltraLite C and Embedded SQL datatypes API]
description, 109
ul_index_flag enumeration [UltraLite C++ API]
description, 235
UL_RS_STATE enumeration [UltraLite C and Embedded SQL datatypes API]
description, 102
ul_stream_error structure [UltraLite C and Embedded SQL datatypes API]
description, 109
ul_sync_info structure
about, 48
ul_sync_info structure [UltraLite C and Embedded SQL datatypes API]
description, 110
ul_sync_result structure [UltraLite C and Embedded SQL datatypes API]
description, 112
ul_sync_state enumeration [UltraLite C and Embedded SQL datatypes API]
description, 105
ul_sync_stats structure [UltraLite C and Embedded SQL datatypes API]
description, 113
ul_sync_status structure
UltraLite embedded SQL, 51
ul_sync_status structure [UltraLite C and Embedded SQL datatypes API]
description, 113
ul_table_sync_type enumeration [UltraLite C++ API]
description, 236
UL_USE_DLL macro
about, 101
ul_validate_data structure [UltraLite C and Embedded SQL datatypes API]
description, 114
ul_validate_status_id enumeration [UltraLite C and Embedded SQL datatypes API]
description, 107
ULActiveSyncStream function
Windows Mobile usage, 60
ulbase.lib
UltraLite C++ development, 26
ULCancelGetNotification method [UltraLite Embedded SQL API]
description, 244
ULChangeEncryptionKey function [UL ESQL]
using, 47
ULChangeEncryptionKey method [UltraLite Embedded SQL API]
description, 244
ULCheckpoint method [UltraLite Embedded SQL API]
description, 244
ULConnection class [UltraLite C++ API]
CancelGetNotification method, 122
ChangeEncryptionKey method, 122
Checkpoint method, 123
Close method, 123
Commit method, 123
CountUploadRows method, 124
CreateNotificationQueue method, 125
DeclareEvent method, 125
description, 119
DestroyNotificationQueue method, 126
ExecuteScalar method, 126
ExecuteScalarV method, 128
ExecuteStatement method, 129
GetChildObjectCount method, 130
GetDatabaseProperty method, 130
GetDatabasePropertyInt method, 130
GetDatabaseSchema method, 131
GetLastDownloadTime method, 131
GetLastError method, 132
GetLastIdentity method, 132
GetNotification method, 133
GetNotificationParameter method, 133
GetSqlca method, 134
GetSyncResult method, 134
GetUserPointer method, 134
GlobalAutoincUsage method, 135

GrantConnectTo method, 135
InitSyncInfo method, 136
OpenTable method, 136
PrepareStatement method, 136
RegisterForEvent method, 137
ResetLastDownloadTime method, 138
RevokeConnectFrom method, 138
Rollback method, 138
RollbackPartialDownload method, 139
SendNotification method, 139
SetDatabaseOption method, 140
SetDatabaseOptionInt method, 140
SetSynchronizationCallback method, 141
SetSyncInfo method, 141
SetUserPointer method, 141
StartSynchronizationDelete method, 142
StopSynchronizationDelete method, 142
Synchronize method, 142
SynchronizeFromProfile method, 143
TriggerEvent method, 144
ValidateDatabase method, 144
ULCountUploadRows method [UltraLite Embedded SQL API]
description, 245
ulcpp.h header file
UltraLite C/C++ API reference, 119
ULCreateDatabase method [UltraLite Embedded SQL API]
description, 246
ULCreateNotificationQueue method [UltraLite Embedded SQL API]
description, 247
ULDatabaseManager class [UltraLite C++ API]
CreateDatabase method, 147
description, 145
DropDatabase method, 148
EnableAesDBEncryption method, 148
EnableAesFipsDBEncryption method, 148
EnableEccE2ee method, 149
EnableEccSyncEncryption method, 149
EnableHttpsSynchronization method, 150
EnableHttpSynchronization method, 150
EnableRsaE2ee method, 151
EnableRsaFipsE2ee method, 151
EnableRsaFipsSyncEncryption method, 151
EnableRsaSyncEncryption method, 152
EnableTcpipSynchronization method, 152
EnableTlsSynchronization method, 153
EnableZlibSyncCompression method, 153
Fini method, 153
Init method, 154
OpenConnection method, 154
SetErrorCallback method, 155
ValidateDatabase method, 155
ULDatabaseSchema class [UltraLite C++ API]
Close method, 157
description, 156
GetConnection method, 157
GetNextPublication method, 157
GetNextTable method, 158
GetPublicationCount method, 158
GetTableCount method, 159
GetTableSchema method, 159
ULDatabaseSchema object
UltraLite C++ development, 21
ULDeclareEvent method [UltraLite Embedded SQL API]
description, 247
ULDeleteAllRows method [UltraLite Embedded SQL API]
description, 248
ULDestroyNotificationQueue method [UltraLite Embedded SQL API]
description, 249
ulecc.lib
UltraLite C++ development, 26
ULECCLibraryVersion method [UltraLite Embedded SQL API]
description, 249
ULEnableAesDBEncryption method [UltraLite Embedded SQL API]
description, 249
ULEnableAesFipsDBEncryption method [UltraLite Embedded SQL API]
description, 250
ULEnableEccE2ee method [UltraLite Embedded SQL API]
description, 250
ULEnableEccSyncEncryption method [UltraLite Embedded SQL API]
description, 251
ULEnableHttpSynchronization method [UltraLite Embedded SQL API]
description, 251
ULEnableRsaE2ee method [UltraLite Embedded SQL API]

description, 252
ULEnableRsaFipsE2ee method [UltraLite Embedded SQL API]
description, 252
ULEnableRsaFipsSyncEncryption method [UltraLite Embedded SQL API]
description, 252
ULEnableRsaSyncEncryption method [UltraLite Embedded SQL API]
description, 253
ULEnableTcpipSynchronization method [UltraLite Embedded SQL API]
description, 253
ULEnableZlibSyncCompression method [UltraLite Embedded SQL API]
description, 254
ULError class [UltraLite C++ API]
Clear method, 160
description, 159
GetErrorInfo method, 160
GetParameter method, 161
GetParameterCount method, 162
GetSQLCode method, 162
GetSQLCount method, 162
GetString method, 163
GetURL method, 163
IsOK method, 164
ULError constructor, 160
ULError constructor
ULError class [UltraLite C++ API], 160
ULErrorInfoInitFromSqlca method [UltraLite Embedded SQL API]
description, 254
ULErrorInfoParameterAt method [UltraLite Embedded SQL API]
description, 254
ULErrorInfoParameterCount method [UltraLite Embedded SQL API]
description, 255
ULErrorInfoString method [UltraLite Embedded SQL API]
description, 255
ULErrorInfoURL method [UltraLite Embedded SQL API]
description, 256
ulfips.lib
UltraLite C++ development, 26
ULGetDatabaseID method [UltraLite Embedded SQL API]
description, 256
ULGetDatabaseProperty method [UltraLite Embedded SQL API]
description, 257
ULGetErrorParameter method [UltraLite Embedded SQL API]
description, 257
ULGetErrorParameterCount method [UltraLite Embedded SQL API]
description, 258
ULGetIdentity method [UltraLite Embedded SQL API]
description, 258
ULGetLastDownloadTime method [UltraLite Embedded SQL API]
description, 258
ULGetNotification method [UltraLite Embedded SQL API]
description, 259
ULGetNotificationParameter method [UltraLite Embedded SQL API]
description, 260
ULGetSyncResult method [UltraLite Embedded SQL API]
description, 261
ULGlobalAutoincUsage method [UltraLite Embedded SQL API]
description, 261
ULGrantConnectTo method [UltraLite Embedded SQL API]
description, 262
ulimp.lib
UltraLite C++ development, 26
ULIndexSchema class [UltraLite C++ API]
Close method, 165
description, 164
GetColumnCount method, 165
GetColumnName method, 165
GetConnection method, 166
GetIndexColumnID method, 166
GetIndexFlags method, 166
GetName method, 166
GetReferencedIndexName method, 167
GetReferencedTableName method, 167
GetTableName method, 167
IsColumnDescending method, 168

ULIndexSchema object
UltraLite C++ development, 22
ULInitSyncInfo function [UL ESQL]
about, 48
ULInitSyncInfo method [UltraLite Embedded SQL API]
description, 262
ULIsSynchronizeMessage function [UL ESQL]
ActiveSync usage, 60
ULIsSynchronizeMessage method [UltraLite Embedded SQL API]
description, 262
ULLibraryVersion method [UltraLite Embedded SQL API]
description, 263
ULPreparedStatement class
UltraLite C++, 9
ULPreparedStatement class [UltraLite C++ API]
AppendParameterByteChunk method, 169
AppendParameterStringChunk method, 170
Close method, 170
description, 168
ExecuteQuery method, 170
ExecuteStatement method, 171
GetConnection method, 171
GetParameterCount method, 171
GetParameterID method, 171
GetParameterType method, 172
GetPlan method, 172
GetResultSetSchema method, 173
GetRowsAffectedCount method, 173
HasResultSet method, 173
SetParameterBinary method, 173
SetParameterDateTime method, 174
SetParameterDouble method, 174
SetParameterFloat method, 175
SetParameterGuid method, 175
SetParameterInt method, 175
SetParameterIntWithType method, 176
SetParameterNull method, 177
SetParameterString method, 177
ulprotos.h header file
UltraLite Embedded SQL API, 237
ULRegisterForEvent method [UltraLite Embedded SQL API]
description, 264
ULResetLastDownloadTime method [UltraLite Embedded SQL API]
description, 265
ULResultSet class [UltraLite C++ API]
AfterLast method, 180
AppendByteChunk method, 180
AppendStringChunk method, 181
BeforeFirst method, 183
Close method, 183
Delete method, 183
DeleteNamed method, 183
description, 177
First method, 184
GetBinary method, 184
GetBinaryLength method, 185
GetByteChunk method, 186
GetConnection method, 188
GetDateTime method, 188
GetDouble method, 189
GetFloat method, 190
GetGuid method, 191
GetInt method, 192
GetIntWithType method, 193
GetResultSetSchema method, 195
GetRowCount method, 195
GetState method, 195
GetString method, 196
GetStringChunk method, 197
GetStringLength method, 199
IsNull method, 200
Last method, 201
Next method, 201
Previous method, 201
Relative method, 202
SetBinary method, 202
SetDateTime method, 203
SetDefault method, 204
SetDouble method, 205
SetFloat method, 206
SetGuid method, 207
SetInt method, 208
SetIntWithType method, 209
SetNull method, 211
SetString method, 212
Update method, 213
UpdateBegin method, 213
ULResultSetSchema class [UltraLite C++ API]
description, 213
GetColumnCount method, 214
GetColumnID method, 214

GetColumnName method, 215
GetColumnPrecision method, 215
GetColumnScale method, 216
GetColumnSize method, 216
GetColumnSQLType method, 217
GetColumnType method, 217
GetConnection method, 217
IsAliased method, 218
ULResultSetSchema object
UltraLite C++, 21
ULRevokeConnectFrom method [UltraLite Embedded SQL API]
description, 265
ULRollbackPartialDownload method [UltraLite Embedded SQL API]
description, 265
ulrsa.lib
UltraLite C++ development, 26
ULRSALibraryVersion method [UltraLite Embedded SQL API]
description, 263
ulrt.lib
UltraLite C++ development, 26
ulrt12.dll
UltraLite C++ development, 26
ulrtc.lib
UltraLite C++ development, 26
ULSendNotification method [UltraLite Embedded SQL API]
description, 266
ULSetDatabaseID method [UltraLite Embedded SQL API]
description, 267
ULSetDatabaseOptionString method [UltraLite Embedded SQL API]
description, 267
ULSetDatabaseOptionULong method [UltraLite Embedded SQL API]
description, 267
ULSetErrorCallback method [UltraLite Embedded SQL API]
description, 268
ULSetSynchronizationCallback method [UltraLite Embedded SQL API]
description, 269
ULSetSyncInfo method [UltraLite Embedded SQL API]
description, 268
ULSignalSyncIsComplete method [UltraLite Embedded SQL API]
description, 269
ULStartSynchronizationDelete method [UltraLite Embedded SQL API]
description, 269
ULStaticFini method [UltraLite Embedded SQL API]
description, 270
ULStaticInit method [UltraLite Embedded SQL API]
description, 270
ULStopSynchronizationDelete method [UltraLite Embedded SQL API]
description, 270
ULSynchronize method [UltraLite Embedded SQL API]
description, 271
ULSynchronizeFromProfile method [UltraLite Embedded SQL API]
description, 271
ULTable class
UltraLite C++ introduction, 15
ULTable class [UltraLite C++ API]
DeleteAllRows method, 221
description, 218
Find method, 221
FindBegin method, 222
FindFirst method, 222
FindLast method, 223
FindNext method, 223
FindPrevious method, 223
GetTableSchema method, 224
Insert method, 224
InsertBegin method, 224
Lookup method, 224
LookupBackward method, 225
LookupBegin method, 225
LookupForward method, 226
TruncateTable method, 226
ULTable object
UltraLite C++ data retrieval example, 13
ULTableSchema class [UltraLite C++ API]
Close method, 228
description, 226
GetColumnDefault method, 228
GetColumnDefaultType method, 228
GetGlobalAutoincPartitionSize method, 229
GetIndexCount method, 229
GetIndexSchema method, 229

GetName method, 230
GetNextIndex method, 230
GetOptimalIndex method, 230
GetPrimaryKey method, 231
GetPublicationPredicate method, 231
GetTableSyncType method, 231
InPublication method, 232
IsColumnInIndex method, 232
IsColumnNullable method, 233
ULTableSchema object
UltraLite C++ development, 21
UltraLite
compiling library for iPhone, 76
creating an iPhone application project, 76
iPhone frameworks, 78
UltraLite C and Embedded SQL datatypes API
ul_binary structure, 109
ul_column_sql_type enumeration, 102
ul_column_storage_type enumeration, 104
ul_error_action enumeration, 105
ul_error_info structure, 109
UL_RS_STATE enumeration, 102
ul_stream_error structure, 109
ul_sync_info structure, 110
ul_sync_result structure, 112
ul_sync_state enumeration, 105
ul_sync_stats structure, 113
ul_sync_status structure, 113
ul_validate_data structure, 114
ul_validate_status_id enumeration, 107
UltraLite C++
accessing schema information, 21
data modification, 9
data modification with SQL, 8
data retrieval, 13
development, 5
iPhone build settings, 5
iPhone include files, 5
Mac OS X include files, 5
quick start, 5
synchronizing data, 23
transaction processing, 21
UltraLite C++ API
ul_column_default_type enumeration, 233
ul_column_name_type enumeration, 234
ul_index_flag enumeration, 235
ul_table_sync_type enumeration, 236
ULConnection class, 119
ULDatabaseManager class, 145
ULDatabaseSchema class, 156
ULError class, 159
ULIndexSchema class, 164
ULPreparedStatement class, 168
ULResultSet class, 177
ULResultSetSchema class, 213
ULTable class, 218
ULTableSchema class, 226
UltraLite C/C++
about, 1
application tutorial, 65
architecture, 1
building a Windows application, 65
building an iPhone application, 75
INCLUDE environment variable, 65
iPhone tutorial, 75
obfuscating UltraLite databases, 47
supported platforms, 1
tutorials, 65, 75
UltraLite C/C++ API reference
ulcpp.h header file, 119
UltraLite C/C++ Common API
alphabetical listing, 101
UltraLite databases
connecting in UltraLite C++, 7
encrypting in embedded SQL, 46
UltraLite C++ API information access , 21
UltraLite for Windows Mobile, 58
UltraLite embedded SQL
authorization, 29
building CustDB application, 57
cursors, 43
developing applications, 27
fetching data, 42
functions, 237
host variables, 32
synchronization, 48
using, 237
UltraLite Embedded SQL API
db_fini method, 238
db_init method, 238
ml_file_transfer_info structure, 278
ml_file_transfer_status structure, 281
MLFileDownload method, 241
MLFileUpload method, 242
MLFiniFileTransferInfo method, 243
mlft_stream_error structure, 282

mlft_stream_error_w structure, 283
MLFTEnableEccE2ee method, 239
MLFTEnableEccEncryption method, 239
MLFTEnableRsaE2ee method, 240
MLFTEnableRsaEncryption method, 240
MLFTEnableRsaFipsE2ee method, 240
MLFTEnableRsaFipsEncryption method, 241
MLFTEnableZlibCompression method, 241
MLInitFileTransferInfo method, 243
ul_database_option_id enumeration, 275
ul_database_property_id enumeration, 276
ULCancelGetNotification method, 244
ULChangeEncryptionKey method, 244
ULCheckpoint method, 244
ULCountUploadRows method, 245
ULCreateDatabase method, 246
ULCreateNotificationQueue method, 247
ULDeclareEvent method, 247
ULDeleteAllRows method, 248
ULDestroyNotificationQueue method, 249
ULECCLibraryVersion method, 249
ULEnableAesDBEncryption method, 249
ULEnableAesFipsDBEncryption method, 250
ULEnableEccE2ee method, 250
ULEnableEccSyncEncryption method, 251
ULEnableHttpSynchronization method, 251
ULEnableRsaE2ee method, 252
ULEnableRsaFipsE2ee method, 252
ULEnableRsaFipsSyncEncryption method, 252
ULEnableRsaSyncEncryption method, 253
ULEnableTcpipSynchronization method, 253
ULEnableZlibSyncCompression method, 254
ULErrorInfoInitFromSqlca method, 254
ULErrorInfoParameterAt method, 254
ULErrorInfoParameterCount method, 255
ULErrorInfoString method, 255
ULErrorInfoURL method, 256
ULGetDatabaseID method, 256
ULGetDatabaseProperty method, 257
ULGetErrorParameter method, 257
ULGetErrorParameterCount method, 258
ULGetIdentity method, 258
ULGetLastDownloadTime method, 258
ULGetNotification method, 259
ULGetNotificationParameter method, 260
ULGetSyncResult method, 261
ULGlobalAutoincUsage method, 261
ULGrantConnectTo method, 262
ULInitSyncInfo method, 262
ULIsSynchronizeMessage method, 262
ULLibraryVersion method, 263
ulprotos.h header file, 237
ULRegisterForEvent method, 264
ULResetLastDownloadTime method, 265
ULRevokeConnectFrom method, 265
ULRollbackPartialDownload method, 265
ULRSALibraryVersion method, 263
ULSendNotification method, 266
ULSetDatabaseID method, 267
ULSetDatabaseOptionString method, 267
ULSetDatabaseOptionULong method, 267
ULSetErrorCallback method, 268
ULSetSynchronizationCallback method, 269
ULSetSyncInfo method, 268
ULSignalSyncIsComplete method, 269
ULStartSynchronizationDelete method, 269
ULStaticFini method, 270
ULStaticInit method, 270
ULStopSynchronizationDelete method, 270
ULSynchronize method, 271
ULSynchronizeFromProfile method, 271
ULTriggerEvent method, 272
ULTruncateTable method, 272
ULValidateDatabase method, 273
ULValidateDatabaseTableName method, 274
UltraLite runtime
deploying Windows Mobile libraries, 26
UltraLite C++ libraries, 26
UltraLite runtime libraries
iPhone, 26
Linux, 25
Mac OS X, 26
ULTriggerEvent method [UltraLite Embedded SQL API]
description, 272
ULTruncateTable method [UltraLite Embedded SQL API]
description, 272
ULValidateDatabase method [UltraLite Embedded SQL API]
description, 273
ULValidateDatabaseTableName method [UltraLite Embedded SQL API]
description, 274
uncommitted transactions
UltraLite embedded SQL, 49

UNDER_CE compiler directive
about, 102
Update method
ULResultSet class [UltraLite C++ API], 213
update mode
UltraLite C++, 16
UpdateBegin method
ULResultSet class [UltraLite C++ API], 213
updating
UltraLite C++ API table rows, 17
user authentication
UltraLite embedded SQL applications, 45

V

ValidateDatabase method
ULConnection class [UltraLite C++ API], 144
ULDatabaseManager class [UltraLite C++ API], 155
values
UltraLite C++ API accessing, 19
Visual C++
UltraLite for Windows Mobile development, 56

W

WindowProc function
ActiveSync usage, 61
Windows Mobile
UltraLite application development overview, 56
UltraLite application synchronization, 60
UltraLite class names, 58
UltraLite platform requirements, 56
UltraLite synchronization menu control, 63
winsock.lib
UltraLite Windows Mobile applications, 56

X

x509 certificate
iPhone, 6
Mac OS X, 6