Contents

About this book ..................................................................................................................................... v

System requirements and supported platforms .............................................................................. 1

Application development .................................................................................................................. 3
  UltraLite C++ application development ........................................................................................ 3
  UltraLite C++ application development using embedded SQL .................................................... 32
  UltraLite application development for Windows Mobile ............................................................. 62

Tutorial: Building a Windows application using the C++ API ....................................................... 71
  Lesson 1: Creating and connecting to a database .......................................................................... 71
  Lesson 2: Inserting data into the database ...................................................................................... 75
  Lesson 3: Selecting and listing rows from the table ....................................................................... 76
  Lesson 4: Adding synchronization to your application ................................................................. 78
  Code listing for tutorial .................................................................................................................. 79

API reference ...................................................................................................................................... 83
  UltraLite C/C++ common API reference ....................................................................................... 83
  UltraLite C/C++ API reference ..................................................................................................... 102
  UltraLite Embedded SQL API reference ..................................................................................... 217

Index .................................................................................................................................................. 261
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.
System requirements and supported platforms

Development platforms
To develop applications using UltraLite C++, you must have 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

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

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

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 approaches” [UltraLite - Database Management and Reference]

UltraLite C++ application development

Quick start guide 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 128
- “How to 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/SQLAnywhere16. 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_certificates 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:

- 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]
- “MobiLink client network protocol options” [MobiLink - Client Administration]
- “Database security” [UltraLite - Database Management and Reference]
- “Certificate Creation utility (createcert)” [SQL Anywhere Server - Database Administration]
- “-x mlsrv16 option” [MobiLink - Server Administration]
- “UltraLite Synchronization utility (ulsync)” [UltraLite - Database Management and Reference]

UltraLite database connections

UltraLite applications must connect to the database before performing operations on its data.

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 method to query the NSDocumentDirectory, for example.

Note
You can find sample code in the %SQLANYSAMP16%\UltraLite\CustDB\ directory.

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 102
- “ULDatabaseManager class [UltraLite C++]” on page 128
- “ULError class [UltraLite C++]” on page 140
Connecting to an UltraLite database

Use the ULDatabaseManager object to create or connect to an UltraLite database named sample.udb.

Prerequisites

There are no prerequisites for this task.

Task

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

   ```cpp
   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:

   ```cpp
   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.

Results

A connection to the sample.udb database is established.

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 14

**Data modification using INSERT, UPDATE and DELETE**

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

**See also**

- “ULPreparedStatement class [UltraLite C++]” on page 148

**Inserting a row in a table**

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.

**Prerequisites**

There are no prerequisites for this task.

**Task**

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();

Results

A new row is added to MyTable where the MyColumn1 value is set to a string "some value".
Deleting a row in a table

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.

Prerequisites

There are no prerequisites for this task.

Task

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();
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.

   ```c
   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.

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

Results

Row entries from MyTable are deleted where the MyColumn value in the table is equal to 7.

Updating a row in a table

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.

Prerequisites

There are no prerequisites for this task.

Task

1. Declare a ULPreparedStatement using the following code:

   ```c
   ULPreparedStatement * prepStmt;
   ```

2. Prepare a SQL statement for execution.

   The following code prepares an UPDATE statement for execution:

   ```c
   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:

   ```c
   if( prepStmt == NULL ) {
     const ULError * ulerr;
   ```
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.

   ```cpp
   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.

   ```cpp
   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.

   ```cpp
   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.

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

**Results**

In this scenario, row entries from MyTable are updated where the MyColumn value is equal to -1. The value is updated to 25.
Retrieving data using SELECT

Execute a SELECT statement to retrieve information from an UltraLite database and handle the result set that is returned.

Prerequisites

There are no prerequisites for this task.

Tasks

1. Declare the required variables using the following code:

   ```c
  _ULPreparedStatement * prepStmt;
  _ULResultSet * resultSet;
   ```

2. Prepare a SQL statement for execution.

   The following code prepares a SELECT statement for execution:

   ```c
   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:

   ```c
   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.

   ```c
   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.

   ```c
   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 be accessed after the Close method is called.

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

**Results**

The result of the SELECT statement contains a string, which is then output to the command prompt.

**See also**

- “ULPreparedStatement.ExecuteQuery method [UltraLite C++]” on page 151

### 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:

```c++
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**


### 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 158

---

**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.

**Caution**

Do not update the primary key of a row. Delete the row and add a new row instead.

**See also**

● “Data creation and modification using SQL statements” on page 6

---

**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 198

Example

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

```cpp
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:

```cpp
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:

- **UltraLite C++ application development**
- **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 insertions

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:

```c
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

Use the Update method to update a row in a table.

**Prerequisites**

There are no prerequisites for this task.
**Task**

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.
   
   ```cpp
   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.

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

4. Execute the Update.

   ```cpp
   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.

**Results**

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

**Next**

Perform additional SQL operations on the database and then execute the Commit method to commit changes to the database. executed.

**See also**

- “Transaction management” on page 20

**Find and lookup modes for searching rows**

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

```sql
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":

```c
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 198

**Access to values in 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:

```cpp
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, the following code 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.

```cpp
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:

```cpp
// 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

- “CAST function [Data type conversion]” [UltraLite - Database Management and Reference]
- “CONVERT function [Data type conversion]” [UltraLite - Database Management and Reference]

Row deletions

The steps to delete a row are simpler than to insert or update rows. There is no delete mode corresponding to the insert or update modes.

You delete a row by moving the cursor to the row you want to delete and then executing the ULTable.Delete method.
See also

- “ULResultSet.Delete method [UltraLite C++]” on page 163

Example

The following code illustrates how to delete the first row in a table:

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

Transaction management

Transactions are started implicitly by the first statement to modify the database, and must be explicitly committed or rolled back.

To commit a transaction, use the ULConnection.Commit method.

To rollback a transaction, use the ULConnection.Rollback method.

See also

- “ULConnection.Commit method [UltraLite C++]” on page 106
- “ULConnection.Rollback method [UltraLite C++]” on page 121
- “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:

```cpp
// 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 148
- “ULResultSet class [UltraLite C++]” on page 158
- “ULTable class [UltraLite C++]” on page 198
- “ULConnection class [UltraLite C++]” on page 102

**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 71
- “ULSetErrorCallback method [UltraLite Embedded SQL]” on page 244
- “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 92
- “UltraLite synchronization parameters” [UltraLite - Database Management and Reference]

Closing the UltraLite database connection

Release software resources when they are no longer being used to prevent the UltraLite database file from remaining in use for as long as the application has a connection to the database.

Prerequisites

There are no prerequisites for this task.

Task

1. Call the Close method to release resources.

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

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

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

   Use the following code when closing the application.

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

Results

The database connection is closed and resources are released.
How to 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 approach 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.

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

The linker/compiler options for libulrt.a are:

```
-L<$SQLANY16>/ultralite/linux/x86/586/lib -lulrt -|ulbase
```

Options for the engine are:

```
-L<$SQLANY16>/ultralite/linux/x86/586/lib -lulrtc -|ulbase
```

The headers command-line option is:

```
-I<$SQLANY16>/sdk/include
```

Deploying an UltraLite application for Windows Mobile (static linkage)

[ This topic has been updated for build 1823. ]

Specify appropriate creation parameters, connection parameters, synchronization parameters, protocol options, link libraries, method calls, and deployment files to ensure that your UltraLite C++ application runs successfully on Windows and Windows Mobile devices.

Prerequisites

There are no prerequisites for this task.

Task

1. Specify the following parameters:

   - When using obfuscation, set the creation parameter `obfuscate=1` while creating the database.
   - When using AES encryption, set the connection parameter `DBKEY=encryption-key` while creating or connecting to the database.

2. Set the appropriate parameter settings when using synchronization 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.

3. When using RSA 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.

For Windows Mobile, these files are located in %SQLANY16%\UltraLite\CE\Arm.50\Lib. For Windows, they are located in %SQLANY16%\UltraLite\Windows\x64\Lib\VS9 or %SQLANY16%\UltraLite\Windows\x86\Lib\VS9.

6. Call the following methods in your UltraLite application:
   - When using AES encryption, the ULDatabaseManager.EnableAesDBEncryption method.

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

<table>
<thead>
<tr>
<th>Synchronization type</th>
<th>Parameter settings</th>
</tr>
</thead>
<tbody>
<tr>
<td>TCP/IP</td>
<td>Call the EnableTcpipSynchronization method.</td>
</tr>
<tr>
<td>HTTP</td>
<td>Call the EnableHttpSynchronization method.</td>
</tr>
<tr>
<td>RSA TLS</td>
<td>Call the EnableTlsSynchronization and EnableRsaSyncEncryption methods.</td>
</tr>
<tr>
<td>RSA HTTPS</td>
<td>Call the EnableHttpsSynchronization and EnableRsaSyncEncryption methods.</td>
</tr>
</tbody>
</table>

**Results**

The UltraLite C++ application, which uses static linkage, runs successfully on the Windows desktop or Windows Mobile device that it is deployed to.
Next

Deploy an UltraLite database to the Windows desktop or Windows Mobile device that the application was deployed to, or create a new database with the deployed application.

See also
- “UltraLite application build and deployment specifications” [UltraLite - Database Management and Reference]
- “UltraLite and UltraLite Java edition database deployment techniques” [UltraLite - Database Management and Reference]

Deploying an UltraLite application for Windows Mobile (dynamic linkage)

[This topic has been updated for build 1823.]

Specify appropriate creation parameters, connection parameters, synchronization parameters, protocol options, link libraries, method calls, and deployment files to ensure that your UltraLite C++ application runs successfully on Windows and Windows Mobile devices.

Prerequisites

There are no prerequisites for this task.

Task

1. Specify the following parameters:
   - When using obfuscation, set the creation parameter `obfuscate=1` while creating the database.
   - When using AES encryption, set the connection parameter `DBKEY=encryption-key` while creating or connecting to the database.

2. Set the appropriate parameter settings when using synchronization in your UltraLite application:

<table>
<thead>
<tr>
<th>Synchronization type</th>
<th>Parameter settings</th>
</tr>
</thead>
<tbody>
<tr>
<td>TCP/IP</td>
<td>Set the Stream synchronization parameter to <code>tcpip</code>.</td>
</tr>
<tr>
<td>HTTP</td>
<td>Set the Stream synchronization parameter to <code>http</code>.</td>
</tr>
<tr>
<td>RSA TLS</td>
<td>Set the Stream synchronization parameter to <code>tls</code>.</td>
</tr>
<tr>
<td>RSA HTTPS</td>
<td>Set the Stream synchronization parameter to <code>https</code>.</td>
</tr>
</tbody>
</table>

3. When using RSA 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:

- ulbase.lib
- ulimp.lib

For Windows Mobile, these files are located in `%SQLANY16%\UltraLite\CE\Arm.50\Lib`. For Windows, they are located in `%SQLANY16%\UltraLite\Windows\x64\Lib\VS9` or `%SQLANY16%\UltraLite\Windows\x86\Lib\VS9`.

6. When linking against the `ulimp.lib` library, define the UL_USE_DLL preprocessor macro when compiling. For example, specify the following:

`-DUL_USE_DLL`

7. Call the following methods in your UltraLite application:

- When using AES encryption, the ULDatabaseManager.EnableAesDBEncryption method.

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

<table>
<thead>
<tr>
<th>Synchronization type</th>
<th>Parameter settings</th>
</tr>
</thead>
<tbody>
<tr>
<td>TCP/IP</td>
<td>Call the EnableTcpipSynchronization method.</td>
</tr>
<tr>
<td>HTTP</td>
<td>Call the EnableHttpSynchronization method.</td>
</tr>
<tr>
<td>RSA TLS</td>
<td>Call the EnableTlsSynchronization and EnableRsaSyncEncryption methods.</td>
</tr>
<tr>
<td>RSA HTTPS</td>
<td>Call the EnableHttpsSynchronization and EnableRsaSyncEncryption methods.</td>
</tr>
</tbody>
</table>

9. Deploy the following files:

- `ulrt16.dll`.
- When using ZLIB compression, `mlczlib16.dll`.
- When using RSA TLS, RSA HTTPS, or RSA E2EE, `mlcrsa16.dll`.

For Windows Mobile, the files are located in `%SQLANY16%\UltraLite\CE\Arm.50`. For Windows, the files are located in `%SQLANY16%\UltraLite\Windows\x64` or `%SQLANY16%\UltraLite\Windows\x86`.

**Results**

The UltraLite C++ application, which uses dynamic linkage, runs successfully on the Windows desktop or Windows Mobile device that it is deployed to.
Next

Deploy an UltraLite database to the Windows desktop or Windows Mobile device that the application was deployed to, or create a new database with the deployed application.

See also

- “UltraLite application build and deployment specifications” [UltraLite - Database Management and Reference]
- “UltraLite and UltraLite Java edition database deployment techniques” [UltraLite - Database Management and Reference]

Deploying an UltraLite application for Windows Mobile (UltraLite engine)

[This topic has been updated for build 1823.]

Specify appropriate creation parameters, connection parameters, synchronization parameters, protocol options, link libraries, method calls, and deployment files to ensure that your UltraLite C++ application runs successfully on Windows and Windows Mobile devices.

Prerequisites

There are no prerequisites for this task.

Task

1. Specify the following parameters:

   - When using obfuscation, set the creation parameter `obfuscate=1` while creating the database.
   - When using AES encryption, set the connection parameter `DBKEY=encryption-key` while creating or connecting to the database.

2. Set the appropriate parameter settings when using synchronization in your UltraLite application:

<table>
<thead>
<tr>
<th>Synchronization type</th>
<th>Parameter settings</th>
</tr>
</thead>
<tbody>
<tr>
<td>TCP/IP</td>
<td>Set the Stream synchronization parameter to <code>tcpip</code>.</td>
</tr>
<tr>
<td>HTTP</td>
<td>Set the Stream synchronization parameter to <code>http</code>.</td>
</tr>
<tr>
<td>RSA TLS</td>
<td>Set the Stream synchronization parameter to <code>tls</code>.</td>
</tr>
<tr>
<td>RSA HTTPS</td>
<td>Set the Stream synchronization parameter to <code>https</code>.</td>
</tr>
</tbody>
</table>

3. When using RSA 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:
   - `ulrtc.lib`
   - `ulbase.lib`

   For Windows Mobile, these files are located in `%SQLANY16%\UltraLite\CE\Arm.50\Lib`. For Windows, they are located in `%SQLANY16%\UltraLite\Windows\x64\Lib\VS9` or `%SQLANY16%\UltraLite\Windows\x86\Lib\VS9`.

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

<table>
<thead>
<tr>
<th>Synchronization type</th>
<th>Parameter settings</th>
</tr>
</thead>
<tbody>
<tr>
<td>TCP/IP</td>
<td>Call the EnableTcpipSynchronization method.</td>
</tr>
<tr>
<td>HTTP</td>
<td>Call the EnableHttpSynchronization method.</td>
</tr>
<tr>
<td>RSA TLS</td>
<td>Call the EnableTlsSynchronization and EnableRsaSyncEncryption methods.</td>
</tr>
<tr>
<td>RSA HTTPS</td>
<td>Call the EnableHttpsSynchronization and EnableRsaSyncEncryption methods.</td>
</tr>
</tbody>
</table>

7. Deploy the following files:
   - `uleng16.exe`
   - When using ZLIB compression, `mclzlib16.dll`.
   - When using RSA TLS, RSA HTTPS, or RSA E2EE `mlcrsa16.dll`.

   For Windows Mobile, the files are located in `%SQLANY16%\UltraLite\CE\Arm.50`. For Windows, the files are located in `%SQLANY16%\UltraLite\Windows\x64` or `%SQLANY16%\UltraLite\Windows\x86`.

**Results**

The UltraLite C++ application, which uses the UltraLite engine, runs successfully on the Windows desktop or Windows Mobile device that it is deployed to.

**Next**

Deploy an UltraLite database to the Windows desktop or Windows Mobile device that the application was deployed to, or create a new database with the deployed application.

**See also**

- “UltraLite application build and deployment specifications” [UltraLite - Database Management and Reference]
- “UltraLite and UltraLite Java edition database deployment techniques” [UltraLite - Database Management and Reference]
Deploying an UltraLite application for Mac OS X or iOS

Specify appropriate creation parameters, connection parameters, synchronization parameters, protocol options, method calls, and deployment files to ensure that your UltraLite application runs successfully on Mac computers, iPhones or iPads.

**Prerequisites**

There are no prerequisites for this task.

**Task**

1. For Mac OS X, add the following runtime library files to your Xcode project:
   - `/Applications/SQLAnywhere16/System/ultralite/macosx/x86_64/libulrt.a`
   - `/Applications/SQLAnywhere16/System/ultralite/macosx/x86_64/libulbase.a`

2. For iOS, to link to the UltraLite runtime library, either:

   Add `install-dir/ultralite/iphone/libulrt.a` to the **Frameworks** group in Xcode.

   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.

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

3. Add the appropriate frameworks to your Xcode project:
   - For Mac OS X, `CoreFoundation.framework`, `CoreServices.framework`, and `Security.framework`.
   - For iOS, `CFNetwork.framework` and `Security.framework`.

4. Specify the following parameters:
   - When using obfuscation, set the creation parameter `obfuscate=1` while creating the database.
   - When using AES encryption, set the connection parameter `DBKEY=encryption-key` while creating or connecting to the database.

5. Set the appropriate parameter settings when using synchronization in your UltraLite application:

<table>
<thead>
<tr>
<th>Synchronization type</th>
<th>Parameter settings</th>
</tr>
</thead>
<tbody>
<tr>
<td>TCP/IP</td>
<td>Set the <strong>Stream</strong> synchronization parameter to <code>tcpip</code>.</td>
</tr>
<tr>
<td>Synchronization type</td>
<td>Parameter settings</td>
</tr>
<tr>
<td>---------------------</td>
<td>--------------------</td>
</tr>
<tr>
<td>HTTP</td>
<td>Set the Stream synchronization parameter to <strong>http</strong>.</td>
</tr>
<tr>
<td>RSA TLS</td>
<td>Set the Stream synchronization parameter to <strong>tls</strong>.</td>
</tr>
<tr>
<td>RSA HTTPS</td>
<td>Set the Stream synchronization parameter to <strong>https</strong>.</td>
</tr>
</tbody>
</table>

6. When using RSA E2EE encryption, set the protocol option `e2ee_public_key=key-file`.

7. When using ZLIB compression, set the protocol option `compression=zlib`.

8. When using AES encryption, call the `ULDatabaseManager.EnableAesDBEncryption` method.

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

<table>
<thead>
<tr>
<th>Synchronization type</th>
<th>Parameter settings</th>
</tr>
</thead>
<tbody>
<tr>
<td>TCP/IP</td>
<td>Call the <code>EnableTcpipSynchronization</code> method.</td>
</tr>
<tr>
<td>HTTP</td>
<td>Call the <code>EnableHttpSynchronization</code> method.</td>
</tr>
<tr>
<td>RSA TLS</td>
<td>Call the <code>EnableTlsSynchronization</code> and <code>EnableRsaSyncEncryption</code> methods.</td>
</tr>
<tr>
<td>RSA HTTPS</td>
<td>Call the <code>EnableHttpsSynchronization</code> and <code>EnableRsaSyncEncryption</code> methods.</td>
</tr>
</tbody>
</table>

**Results**

The UltraLite application runs successfully on the Mac OS X desktop or iOS device that it is deployed to.

**Next**

Deploy an UltraLite database to the Mac desktop, iPhone, or iPad that the application was deployed to, or create a new database with the deployed application.

**See also**

- “UltraLite application build and deployment specifications” [UltraLite - Database Management and Reference]
- “UltraLite and UltraLite Java edition database deployment techniques” [UltraLite - Database Management and Reference]
### Deploying an UltraLite application for Linux

Specify appropriate creation parameters, connection parameters, synchronization parameters, protocol options, method calls, and deployment files to ensure that your UltraLite application runs successfully on Linux.

#### Prerequisites

There are no prerequisites for this task.

#### Task

1. Specify the following parameters:
   - When using obfuscation, set the creation parameter **obfuscate=1** while creating the database.
   - When using AES encryption, set the connection parameter **DBKEY=encryption-key** while creating or connecting to the database.

2. Set the appropriate parameter settings when using synchronization in your UltraLite application:

<table>
<thead>
<tr>
<th>Synchronization type</th>
<th>Parameter settings</th>
</tr>
</thead>
<tbody>
<tr>
<td>TCP/IP</td>
<td>Set the <strong>Stream</strong> synchronization parameter to <strong>tcpip</strong>.</td>
</tr>
<tr>
<td>HTTP</td>
<td>Set the <strong>Stream</strong> synchronization parameter to <strong>http</strong>.</td>
</tr>
<tr>
<td>RSA TLS</td>
<td>Set the <strong>Stream</strong> synchronization parameter to <strong>tls</strong>.</td>
</tr>
<tr>
<td>RSA HTTPS</td>
<td>Set the <strong>Stream</strong> synchronization parameter to <strong>https</strong>.</td>
</tr>
</tbody>
</table>

3. When using RSA 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:
   - **libulrt.a**.
   - **libulbase.a**.
   - When using RSA TLS, RSA HTTPS, or RSA E2EE, **libulrsa.a**.

   These files are located in `/opt/sqlanywhere16/ultralite/linux/x86/586/lib`.

6. When using AES encryption, call the **ULDatabaseManager.EnableAesDBEncryption** method.

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

<table>
<thead>
<tr>
<th>Synchronization type</th>
<th>Parameter settings</th>
</tr>
</thead>
<tbody>
<tr>
<td>TCP/IP</td>
<td>Call the EnableTcpipSynchronization method.</td>
</tr>
<tr>
<td>HTTP</td>
<td>Call the EnableHttpSynchronization method.</td>
</tr>
<tr>
<td>RSA TLS</td>
<td>Call the EnableTlsSynchronization and EnableRsaSyncEncryption methods.</td>
</tr>
<tr>
<td>RSA HTTPS</td>
<td>Call the EnableHttpsSynchronization and EnableRsaSyncEncryption methods.</td>
</tr>
</tbody>
</table>

Results

The UltraLite application runs successfully on the Linux computer that it is deployed to.

Next

Deploy an UltraLite database to the Linux computer that the application was deployed to, or create a new database with the deployed application.

See also

- “UltraLite application build and deployment specifications” [UltraLite - Database Management and Reference]
- “UltraLite and UltraLite Java edition database deployment techniques” [UltraLite - Database Management and Reference]

UltraLite C++ application development using embedded SQL

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

See also

- “UltraLite Embedded SQL API reference” on page 217
- “SQL preprocessor for UltraLite utility (sqlpp)” [UltraLite - Database Management and Reference]

Quick start guide to UltraLite embedded SQL application development

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:

```sql
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 60
- “UltraLite C++ application development using embedded SQL” on page 32

**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.

```c
#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:

   \[
   \text{EXEC SQL INCLUDE SQLCA;}
   \]

   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 method 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 217

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.

SQL Communications Area Initialization

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 %SQLANY16%\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 %SQLANY16%\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]

UltraLite database connection using embedded SQL

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.

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 a single SQLCA**

You can use a single SQLCA to manage multiple connections 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**

- “UltraLite connection parameters” [UltraLite - Database Management and Reference]
- “CONNECT statement [ESQL] [Interactive SQL]” [SQL Anywhere Server - SQL Reference]

**Using multiple SQLCAs to manage multiple database connections**

Use the SET SQLCA embedded SQL statement to tell the SQL preprocessor to use a specific SQLCA for database requests.

**Prerequisites**

There are no prerequisites for this task.

**Task**

1. Initialize Each SQLCA used in your program with a call to `db_init`.

2. At the top of your program or in a header file, 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.

**Results**

The state within the preprocessor changes so that any reference to the SQLCA uses the given string.

**Next**

Clean up resources when closing your application with a call to the `db_fini` method.
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:

```c
/* 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. */
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:

```c
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**
  
  ```c
  short int I;
  unsigned short int I;
  ```

- **32-bit signed integer**
  
  ```c
  long int l;
  unsigned long int l;
  ```

- **4-byte floating-point number**
  
  ```c
  float f;
  ```

- **8-byte floating-point number**
  
  ```c
  double d;
  ```

- **Packed decimal number**
  
  ```c
  DECL_DECIMAL(p,s)
  typedef struct TYPE_DECIMAL {
      char array[1];
  } TYPE_DECIMAL;
  ```

- **Null terminated, blank-padded character string**
  
  ```c
  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.
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.

```c
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.

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

- Fixed-length blank padded character string

```c
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).

```c
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.

```c
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

```c
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 */
} 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. 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:

```c
#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:

```c
#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 `%SQLANY16%\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 cannot parse C code, it assumes all host variables, no matter where they are declared, are known globally following their declaration.

```sql
// 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.

```c
#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.

```c
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

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

```c++
typedef short a_bool;
#define TRUE ((a_bool)(1==1))
#define FALSE ((a_bool)(0==1))
public class {
  long host_member;  // 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.

```sql
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 cannot 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:

<table>
<thead>
<tr>
<th>Indicator value</th>
<th>Supplying value to database</th>
<th>Receiving value from database</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Host variable value</td>
<td>Fetched a non-NULL value.</td>
</tr>
</tbody>
</table>
### 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. Indicator variables are needed to pass NULL values to the database or receive NULL results back.

### Using indicator variables when inserting NULL

An INSERT statement can include an indicator variable as follows:

```sql
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 SQL_E_NO_INDICATOR error is generated.

### See also
- “SQL Communications Area Initialization” on page 35

---

<table>
<thead>
<tr>
<th>Indicator value</th>
<th>Supplying value to database</th>
<th>Receiving value from database</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>NULL value</td>
<td>Fetched a NULL value</td>
</tr>
</tbody>
</table>
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

- “SQL Communications Area Initialization” on page 35

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.

```sql
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 */
    }
}
```
Multiple row fetching

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.

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.

A cursor can be managed using the following steps:

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.
4. 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.
5. Close the cursor, using the CLOSE statement.

The following is a simple example of cursor usage:

```c
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

<table>
<thead>
<tr>
<th>Absolute row from start</th>
<th>Absolute row from end</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>−n − 1</td>
</tr>
<tr>
<td>1</td>
<td>−n</td>
</tr>
<tr>
<td>2</td>
<td>−n + 1</td>
</tr>
<tr>
<td>3</td>
<td>−n + 2</td>
</tr>
<tr>
<td>n - 2</td>
<td>−3</td>
</tr>
<tr>
<td>n - 1</td>
<td>−2</td>
</tr>
<tr>
<td>n</td>
<td>−1</td>
</tr>
<tr>
<td>n + 1</td>
<td>0</td>
</tr>
</tbody>
</table>

Before first 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]
- “Cursor principles” [SQL Anywhere Server - Programming]
- “Tip: 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 %SQLANYSAMP16%UltraLite\esqlauth directory. The code below is taken from %SQLANYSAMP16%UltraLite\esqlauth\sample.sqc.

```sql
//embedded SQL
app() { ...

/* Declare fields */
EXEC SQL BEGIN DECLARE SECTION;
  char uid[31];
  char pwd[31];
EXEC SQL END DECLARE SECTION;
...
```
EXEC SQL END DECLARE SECTION;
db_init( &sqlca );
...
EXEC SQL CONNECT "DBA" IDENTIFIED BY "sql";
if( SQLCODE == SQLERROR ) {
    printf("Enter new user ID and password\n");
    scanf( "%s %s", uid, pwd );
    ULGrantConnectTo( &sqlca,
         UL_TEXT(uid), UL_TEXT(pwd) );
    if( SQLCODE == SQL_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 238
● “ULRevokeConnectFrom method [UltraLite Embedded SQL]” on page 242

Data encryption with 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.

The following procedure is generally used to connect to an encrypted UltraLite database:

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

2. 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.

3. 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.

Change the encryption key

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

Supply the new key as an argument of the ULChangeEncryptionKey method.

Obfuscation

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 222
- “UltraLite database creation parameters” [UltraLite - Database Management and Reference]

Synchronization setup for an embedded SQL application

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

Members of the structures in the embedded SQL API are similar to the UltraLite C++ API.

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.

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 method call to synchronize. The outline of the method is the same in each development model.
The following procedure is generally used to 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 method, supplying the structure or object as argument.

Ensure that there are no uncommitted changes when you synchronize.

See also
- “Synchronization parameter initialization” on page 54
- “UltraLite network protocol options” [UltraLite - Database Management and Reference]
- “Synchronization invocation” on page 55
- “ul_sync_info structure [UltraLite C and Embedded SQL datatypes]” on page 92
- “UltraLite synchronization parameters” [UltraLite - Database Management and Reference]

Synchronization parameter initialization

The synchronization parameters are stored in a ul_sync_info structure.

The members of the ul_sync_info structure are undefined on initialization. You must set your parameters to their initial values with a call to a special method. The synchronization parameters are defined in a structure declared in the UltraLite header file `%SQLANY16%SDK\Include\ulglobal.h`.

See also
- “ULInitSyncInfo method [UltraLite Embedded SQL]” on page 239

Example

The following example illustrates how to initialize synchronization parameters with the ULInitSyncInfo method:

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

Example

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:

```c
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 );
```
Synchronization invocation

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, 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.

To invoke synchronization, call the ULInitSyncInfo method to initialize the synchronization parameters, and then call the ULSynchronize method 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.

Committed changes and synchronization

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]

Initial data for 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:

```c
if( psqlca->sqlcode == SQLE_MOBILINK_COMMUNICATIONS_ERROR ) {
    printf( " Stream error information:\n" );
```
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:

```c
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 a parameter.

The ul_sync_status structure has the following members:

```c
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;
  }               rcv;
}
```
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  Returns a pointer to the ul_sync_info structure.

db_table_count  Returns the number of tables in the database.

table_id  Returns 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[]  Returns the name of the current table.

table_name_w2[]  Returns the 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  Returns 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 been taken.
- **UL_SYNC_STATE_CONNECTING**  The synchronization stream has been built, but not 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 has 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 the 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**  Returns the pointer to the connection's active SQLCA.
• **current_download_row_count**  Returns the number of rows that have been downloaded so far. This number includes duplicate rows that aren't included in received.inserts, received.updates, or received.deletes.

• **total_download_row_count**  Returns the total number of rows to be received in the download. This number includes duplicate rows that aren't included in received.inserts, received.updates, or received.deletes.

**Example**

The following code illustrates a simple observer function:

```c
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_SYNC_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 you synchronize 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
```
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 \%SQLANYSAMP16\%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 92
- “ul_sync_status structure [UltraLite C and Embedded SQL datatypes]” on page 96
- “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.

General build procedures

The following procedure is generally used when building 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.

You can find a makefile that uses this process in the \%SQLANYSAMP16\%UltraLite\ESQLSecurity directory.
Note
Separately licensed component required.

FIPS-certified encryption requires a separate license. All strong encryption technologies are subject to export regulations.

See “Separately licensed components” [SQL Anywhere 16 - Introduction].

See also
● “SQL preprocessor for UltraLite utility (sqlpp)” [UltraLite - Database Management and Reference]

Development tool configuration 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.

See also
● “SQL preprocessor for UltraLite utility (sqlpp)” [UltraLite - Database Management and Reference]

Running the SQL preprocessor

Incorporate the SQL preprocessor into a dependency-based build environment by adding instructions to run the it for Visual C++.

Prerequisites

There are no prerequisites for this task.

Task

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):

   "$SQLANY16%\Bin32\sqlpp.exe" -q -u $(InputPath) $(InputName).cpp

   where SQLANY16 is an environment variable that points to your SQL Anywhere installation directory.
 Application development

- 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 $(SQLANY16)\SDK\Include and any other desired include folders to your include path as a comma-separated list.

Results

The SQL preprocessor is configured for Visual C++ development.

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%\SQLAny16\ultralite\ce\arm.50\lib.\n
You can test your applications under an emulator on most Windows Mobile target platforms.

See also
- “Supported platforms” [SQL Anywhere 16 - Introduction]

CustDB sample application

CustDB is a simple sales-status application that is provided as a Visual Studio solution. It is located in the %SQLANYSAMP16%\UltraLite\ directory of your SQL Anywhere installation.

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.

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 16 - Introduction*]
- “Embedded SQL application building” on page 60

**Building the CustDB sample application**

Build the CustDB sample application to see how an application interfaces with an UltraLite database.

**Prerequisites**

There are no prerequisites for this task.

**Task**


2. Open the project file located in the `%SQLANYSAMP16%\UltraLite\CustDB` directory.

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 16 » 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 `%SQLANYSAMP16%\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`.

**Results**

The CustDB application loads.
**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:

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

**Assigning class names for applications**

Assign a distinct class name for your application if you are using MFC. 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 about the topic.

**Prerequisites**

There are no prerequisites for this task.

**Task**

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.

   ```c
   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 code to your application's InitInstance method.

```cpp
m_pMainWnd = &dlg;
```

The code ensures that if the dlg main dialog is closed, the application also closes. 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.

7. 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:

```plaintext
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.

8. Save the .rc file.

9. Add code to catch the synchronization message.

**Results**

A distinct class name for your application is created.

**See also**

- “Adding ActiveSync synchronization in the main dialog class” on page 67

**Windows Mobile synchronization**

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

- ActiveSync
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

- “ActiveSync synchronization setup” on page 66
- “TCP/IP, HTTP, or HTTPS synchronization from Windows Mobile” on page 69
- “TCP/IP, HTTP, or HTTPS synchronization from Windows Mobile” on page 69
- “UltraLite synchronization parameters” [UltraLite - Database Management and Reference]

ActiveSync synchronization setup

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]

ActiveSync synchronization setup (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 UL.Synchronize.

See also

- “ULIsSynchronizeMessage method [UltraLite Embedded SQL]” on page 239

Adding ActiveSync synchronization in the main dialog class

Catch synchronization messages in your main dialog class.

Prerequisites

You must use Microsoft Foundation Classes to develop your application.

Task

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:

   ```cpp
   static UINT WM_ULTRALITE_SYNC_MESSAGE =
       ::RegisterWindowMessage( ULAS_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 the UL.Synchronize method.
LRESULT CMyAppDlg::OnDoUltraLiteSync(
    WPARAM wParam,
    LPARAM lParam
);

The return value of this function should be 0.

**Results**

The main dialog class performs a synchronization.

**See also**

- “Assigning class names for applications” on page 64
- “ULIsSynchronizeMessage method [UltraLite Embedded SQL]” on page 239

**Adding ActiveSync synchronization in the Application class**

Catch synchronization messages in your application class.

**Prerequisites**

You must use Microsoft Foundation Classes to develop your application.

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

**Task**

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 method appears.

4. Change the PreTranslateMessage method to read as follows:

   ```cpp
   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 the DoSync method calls the ULsSynchronize method.
   ```
Results

The PreTranslateMessage method performs a synchronization.

See also

- “Assigning class names for applications” on page 64
- “ULSynchronize method [UltraLite Embedded SQL]” on page 247
- “ULIsSynchronizeMessage method [UltraLite Embedded SQL]” on page 239

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.
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 this lesson, you create an UltraLite database. You then write, compile, and run a C++ application that accesses the database you created.

Prerequisites

This lesson assumes that you have installed the required software. See “Tutorial: Building a Windows application using the C++ API” on page 71.

Task

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:

<table>
<thead>
<tr>
<th>Column name</th>
<th>Data type (size)</th>
<th>Column allows NULL values?</th>
<th>Default value</th>
<th>Primary Key</th>
</tr>
</thead>
<tbody>
<tr>
<td>cust_id</td>
<td>integer</td>
<td>No</td>
<td>autoincrement</td>
<td>ascending</td>
</tr>
<tr>
<td>cust_name</td>
<td>varchar(30)</td>
<td>No</td>
<td>None</td>
<td></td>
</tr>
</tbody>
</table>

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


8. On the Files tab, click C++ Source File.

9. Save the file as customer.cpp in your tutorial directory.

10. 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
    #define MAX_ERROR_LEN 256
    ```

11. 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.

12. 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:

```c
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_ERROR_LEN ];
                error->GetString( etext, MAX_ERROR_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.

13. 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.

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

14. 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.

```cpp
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;
}
```

15. 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:

a. Open a command prompt and change to your tutorial directory.
b. Create a makefile named `makefile`.
c. In the makefile, add directories to your include path.
   ```make
   IncludeFolders=\$(SQLANY16)\SDK\Include
   ```
d. In the makefile, add directories to your libraries path.
   ```make
   LibraryFolders=\LIBPATH:$(SQLANY16)\UltraLite\Windows\x86\Lib\vs8
   ```
e. In the makefile, add libraries to your linker options.
   ```make
   Libraries=ulimp.lib
   ```
The UltraLite runtime library is named `ulimp.lib`.
f. In the makefile, set compiler options. You must enter these options on a single line.
   ```make
   CompileOptions=/c /nologo /W3 /Od /Zi /DWIN32 /DUL_USE_DLL
   ```
g. In the makefile, add an instruction for linking the application.
   ```make
   customer.exe: customer.obj
   link /NOLOGO /DEBUG customer.obj $(LibraryFolders) $(Libraries)
   ```
h. In the makefile, add an instruction for compiling the application.
   ```make
   customer.obj: customer.cpp
   cl $(CompileOptions) $(IncludeFolders) customer.cpp
   ```
Lesson 2: Inserting data into the database

In this lesson, you add data to a database.

Prerequisites

This lesson assumes you have completed all preceding lessons. See “Lesson 1: Creating and connecting to a database” on page 71.

Task

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

```cpp
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" );
```

Results

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

Next

Proceed to “Lesson 2: Inserting data into the database” on page 75.

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 21

Lesson 2: Inserting data into the database

In this lesson, you add data to a database.

Prerequisites

This lesson assumes you have completed all preceding lessons. See “Lesson 1: Creating and connecting to a database” on page 71.

Task

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

```cpp
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.

Results

The application runs and you can insert data into the ULCustomer table.

Next

Proceed to “Lesson 3: Selecting and listing rows from the table” on page 76.

Lesson 3: Selecting and listing rows from the table

In this lesson, you retrieve rows from the table and print them on the command line.

Prerequisites

This lesson assumes you have completed all preceding lessons. See “Lesson 1: Creating and connecting to a database” on page 71.
Task

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.

   ```cpp
   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:

   ```cpp
   do_select(conn);
   ```

3. Compile your application by running `nmake`.

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

Results

A list of all the customer IDs and customer names in the ULCustomer table is outputted.
Lesson 4: Adding synchronization to your application

In this lesson, you add synchronization code to your application, start the MobiLink server, and run your application to synchronize with the consolidated database.

Prerequisites

This lesson assumes you have completed all preceding lessons. See “Lesson 1: Creating and connecting to a database” on page 71.

Context and remarks

The UltraLite database you created in the previous lessons synchronizes with the UltraLite 16 Sample database. The UltraLite 16 Sample database has a ULCustomer table whose columns include those in the customer table of your local UltraLite database.

Task

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.

```cpp
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" );
    }
}
```
2. Place the following line of code to the main method after the do_select method call:

   do_sync(conn);

3. Compile your application by running `nmake`.

4. Start the MobiLink server.

   At a command prompt, run the following command:

   `mlsrv16 -c "dsn=SQL Anywhere 16 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.

5. Run your application by typing `customer` at a command prompt.

**Results**

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.

```c
#include <tchar.h>
#include <stdio.h>
#include "ulcpp.h"
#define MAX_NAME_LEN 100
#define MAX_ERROR_LEN 256

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;
   ul_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_ERROR_LEN ];
            error->GetString( etext, MAX_ERROR_LEN );
            _tprintf( "Error %ld: %s\n", code, etext );
            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 );
}

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( "'\n"
    return false;
    }
    return true;
}
do_sync( conn );
conn->Close();
ULDatabaseManager::Fini();
return 0;
}
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"$(VCDIR)\include" \ /I"$(SQLANY16)\SDK\Include"
  sample.obj: sample.cpp
  cl $(CompileOptions) $(IncludeFolders) sample.cpp

  VCDIR is your Visual C++ directory and SQLANY16 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 62

Example
/D UNDER_CE

UL_RS_STATE enumeration
Specifies possible result set or cursor states.

Syntax
public enum UL_RS_STATE

Members

<table>
<thead>
<tr>
<th>Member name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>UL_RS_STATE_ERROR</td>
<td>Error.</td>
</tr>
<tr>
<td>UL_RS_STATE_UNPREPARED</td>
<td>Not prepared.</td>
</tr>
<tr>
<td>UL_RS_STATE_ON_ROW</td>
<td>On a valid row.</td>
</tr>
<tr>
<td>UL_RS_STATE_BEFORE_FIRST</td>
<td>Before the first row.</td>
</tr>
<tr>
<td>UL_RS_STATE_AFTER_LAST</td>
<td>After the last row.</td>
</tr>
<tr>
<td>UL_RS_STATE_COMPLETED</td>
<td>Closed.</td>
</tr>
</tbody>
</table>

ul_column_sql_type enumeration
Represents the SQL types for a column.

Syntax
public enum ul_column_sql_type
<table>
<thead>
<tr>
<th>Member name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>UL_SQLTYPE_BAD_INDEX</td>
<td>Represents that the column at the specified index does not exist.</td>
</tr>
<tr>
<td>UL_SQLTYPE_S_LONG</td>
<td>Represents that the column contains a signed long.</td>
</tr>
<tr>
<td>UL_SQLTYPE_U_LONG</td>
<td>Represents that the column contains an unsigned long.</td>
</tr>
<tr>
<td>UL_SQLTYPE_S_SHORT</td>
<td>Represents that the column contains a signed short.</td>
</tr>
<tr>
<td>UL_SQLTYPE_U_SHORT</td>
<td>Represents that the column contains an unsigned short.</td>
</tr>
<tr>
<td>UL_SQLTYPE_S_BIG</td>
<td>Represents that the column contains a signed 64-bit integer.</td>
</tr>
<tr>
<td>UL_SQLTYPE_U_BIG</td>
<td>Represents that the column contains an unsigned 64-bit integer.</td>
</tr>
<tr>
<td>UL_SQLTYPE_TINY</td>
<td>Represents that the column contains an unsigned 8-bit integer.</td>
</tr>
<tr>
<td>UL_SQLTYPE_BIT</td>
<td>Represents that the column contains a 1-bit flag.</td>
</tr>
<tr>
<td>UL_SQLTYPE_TIMESTAMP</td>
<td>Represents that the column contains timestamp information.</td>
</tr>
<tr>
<td>UL_SQLTYPE_DATE</td>
<td>Represents that the column contains date information.</td>
</tr>
<tr>
<td>UL_SQLTYPE_TIME</td>
<td>Represents that the column contains time information.</td>
</tr>
<tr>
<td>UL_SQLTYPE_DOUBLE</td>
<td>Represents that the column contains a double precision floating-point number (8 bytes).</td>
</tr>
<tr>
<td>UL_SQLTYPE_REAL</td>
<td>Represents that the column contains a single precision floating-point number (4 bytes).</td>
</tr>
<tr>
<td>UL_SQLTYPE_NUMERIC</td>
<td>Represents that the column contains exact numerical data, with specified precision and scale.</td>
</tr>
<tr>
<td>UL_SQLTYPE_BINARY</td>
<td>Represents that the column contains binary data with a specified maximum length.</td>
</tr>
<tr>
<td>UL_SQLTYPE_CHAR</td>
<td>Represents that the column contains character data with a specified length.</td>
</tr>
<tr>
<td>UL_SQLTYPE_LONGVARCHAR</td>
<td>Represents that the column contains character data with variable length.</td>
</tr>
<tr>
<td>UL_SQLTYPE_LONGBINARY</td>
<td>Represents that the column contains binary data with variable length.</td>
</tr>
<tr>
<td>UL_SQLTYPE_UUID</td>
<td>Represents that the column contains a UUID.</td>
</tr>
</tbody>
</table>
### Member name | Description
---|---
UL_SQLTYPE_ST_GEOMETRY | Represents that the column contains spatial data in the form of points.
UL_SQLTYPE_TIME-STEMP_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**
```java
public enum ul_column_storage_type
```

**Members**

<table>
<thead>
<tr>
<th>Member name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>UL_TYPE_BAD_INDEX</td>
<td>Represents an invalid value.</td>
</tr>
<tr>
<td>UL_TYPE_S_LONG</td>
<td>Represents a ul_s_long (32 bit signed int).</td>
</tr>
<tr>
<td>UL_TYPE_U_LONG</td>
<td>Represents a ul_u_long (32 bit unsigned int).</td>
</tr>
<tr>
<td>UL_TYPE_S_SHORT</td>
<td>Represents a ul_s_short (16 bit signed int).</td>
</tr>
<tr>
<td>UL_TYPE_U_SHORT</td>
<td>Represents a ul_u_short (16 bit unsigned int).</td>
</tr>
<tr>
<td>UL_TYPE_S_BIG</td>
<td>Represents a ul_s_big (64 bit signed int).</td>
</tr>
<tr>
<td>UL_TYPE_U_BIG</td>
<td>Represents a ul_u_big (64 bit unsigned int).</td>
</tr>
<tr>
<td>UL_TYPE_TINY</td>
<td>Represents a ul_byte (8 bit unsigned).</td>
</tr>
<tr>
<td>UL_TYPE_BIT</td>
<td>Represents a ul_byte (8 bit unsigned, 1 bit used).</td>
</tr>
<tr>
<td>UL_TYPE_DOUBLE</td>
<td>Represents a ul_double (double).</td>
</tr>
<tr>
<td>UL_TYPE_REAL</td>
<td>Represents a ul_real (float).</td>
</tr>
<tr>
<td>UL_TYPE_BINARY</td>
<td>Represents a ul_binary (2 byte length followed by byte array).</td>
</tr>
<tr>
<td>UL_TYPE_TIMESTAMP_STRUCT</td>
<td>Represents a DECL_DATETIME.</td>
</tr>
<tr>
<td>Member name</td>
<td>Description</td>
</tr>
<tr>
<td>------------------</td>
<td>------------------------------------</td>
</tr>
<tr>
<td>UL_TYPE_TCHAR</td>
<td>Represents a character array (string buffer).</td>
</tr>
<tr>
<td>UL_TYPE_CHAR</td>
<td>Represents a char array (string buffer).</td>
</tr>
<tr>
<td>UL_TYPE_WCHAR</td>
<td>Represents a ul_wchar (UTF16) array.</td>
</tr>
<tr>
<td>UL_TYPE_GUID</td>
<td>Represents a GUID structure.</td>
</tr>
</tbody>
</table>

**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**

```java
public enum ul_error_action
```

**Members**

<table>
<thead>
<tr>
<th>Member name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>UL_ERROR_ACTION_DEFAULT</td>
<td>Behave as if there is no error callback.</td>
</tr>
<tr>
<td>UL_ERROR_ACTION_CANCEL</td>
<td>Cancel the operation that raised the error.</td>
</tr>
<tr>
<td>UL_ERROR_ACTION_TRY_AGAIN</td>
<td>Retry the operation that raised the error.</td>
</tr>
<tr>
<td>UL_ERROR_ACTION_CONTINUE</td>
<td>Continue execution, ignoring the operation that raised the error.</td>
</tr>
</tbody>
</table>

**Remarks**

Not all actions apply to all error codes.

**ul_sync_state enumeration**

Indicates the current stage of synchronization.

**Syntax**

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

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

```java
public enum ul_validate_status_id
```

Members

<table>
<thead>
<tr>
<th>Member name</th>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>UL_VALID_NO_ERROR</td>
<td>No error occurred.</td>
<td>0</td>
</tr>
<tr>
<td>UL_VALID_START</td>
<td>Start validation.</td>
<td>1</td>
</tr>
<tr>
<td>UL_VALID_END</td>
<td>End validation.</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>Parm1 tracks the resulting sqlcode, which indicates success or failure.</td>
<td></td>
</tr>
<tr>
<td>UL_VALID_CHECKING_PAGE</td>
<td>Send a periodic status message while checking database pages.</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>Parm1 tracks a number associated with the page. The order is not defined.</td>
<td></td>
</tr>
<tr>
<td>UL_VALID_CHECKING_TABLE</td>
<td>Checking a table.</td>
<td>20</td>
</tr>
<tr>
<td></td>
<td>Parm1 tracks the table name.</td>
<td></td>
</tr>
<tr>
<td>UL_VALID_CHECKING_INDEX</td>
<td>Checking an index.</td>
<td>21</td>
</tr>
<tr>
<td></td>
<td>Parm1 stores the table name and parm2 stores the index name.</td>
<td></td>
</tr>
<tr>
<td>UL_VALID_HASH_REPORT</td>
<td>Reporting on the index hash use.</td>
<td>30</td>
</tr>
<tr>
<td></td>
<td>(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.</td>
<td></td>
</tr>
<tr>
<td>UL_VALID_REDUNDANT_INDEX</td>
<td>A redundant index was found.</td>
<td>31</td>
</tr>
<tr>
<td></td>
<td>(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.</td>
<td></td>
</tr>
</tbody>
</table>
ul_binary structure

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

Syntax

    public typedef struct ul_binary

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

<table>
<thead>
<tr>
<th>Member name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>data</td>
<td>ul_byte</td>
<td>The actual data to be set (for insert) or that was fetched (for select).</td>
</tr>
<tr>
<td>len</td>
<td>ul_length</td>
<td>The number of bytes in the value.</td>
</tr>
</tbody>
</table>

ul_error_info structure

Stores complete information about an UltraLite error.

Syntax

```c
public typedef struct ul_error_info
```

Members

<table>
<thead>
<tr>
<th>Member name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sqlcode</td>
<td>an_sql_code</td>
<td>The SQLCODE value.</td>
</tr>
<tr>
<td>sqlcount</td>
<td>ul_s_long</td>
<td>The SQLCOUNT value.</td>
</tr>
</tbody>
</table>

See also

- “ULErrorInfoString method [UltraLite Embedded SQL]” on page 232
- “ULErrorInfoGetLast method [UltraLite Embedded SQL]” on page 233
- “ULErrorInfoGetLastFromSqlca method [UltraLite Embedded SQL]” on page 231
- “ULErrorInfoParameterCount method [UltraLite Embedded SQL]” on page 232
- “ULErrorInfoParameterAt method [UltraLite Embedded SQL]” on page 231

ul_stream_error structure

Stores synchronization communication stream error information.

Syntax

```c
public typedef struct ul_stream_error
```

Members

<table>
<thead>
<tr>
<th>Member name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>error_string</td>
<td>char</td>
<td>A string with additional information, if available, for the stream_error_code value.</td>
</tr>
</tbody>
</table>
### ul_sync_info structure

Stores synchronization data.

#### Syntax

```c
public typedef struct ul_sync_info
```

#### Members

<table>
<thead>
<tr>
<th>Member name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>additionalParms</td>
<td>const char *</td>
<td>A string of name value pairs &quot;name=value;&quot; with extra parameters.</td>
</tr>
<tr>
<td>authParms</td>
<td>const char *</td>
<td>An array of authentication parameters in MobiLink events.</td>
</tr>
<tr>
<td>authStatus</td>
<td>ul_auth_status</td>
<td>The status of MobiLink user authentication. The MobiLink server provides this information to the client.</td>
</tr>
<tr>
<td>authValue</td>
<td>ul_s_long</td>
<td>The results of a custom MobiLink user authentication script. The MobiLink server provides this information to the client to determine the authentication status.</td>
</tr>
<tr>
<td>downloadOnly</td>
<td>ul_bool</td>
<td>Do not upload any changes from the UltraLite database during the current synchronization.</td>
</tr>
<tr>
<td>ignoredRows</td>
<td>ul_bool</td>
<td>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.</td>
</tr>
<tr>
<td>initVerify</td>
<td>ul_sync_info *</td>
<td>Initialize verification.</td>
</tr>
<tr>
<td>keepPartialDownload</td>
<td>ul_bool</td>
<td>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.</td>
</tr>
<tr>
<td>Member name</td>
<td>Type</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------------------</td>
<td>---------------</td>
<td>------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>new_password</td>
<td>const char *</td>
<td>A string specifying a new MobiLink password associated with the user name. This parameter is optional.</td>
</tr>
<tr>
<td>num_authParms</td>
<td>ul_byte</td>
<td>The number of authentication parameters being passed to authentication parameters in MobiLink events.</td>
</tr>
<tr>
<td>observer</td>
<td>ul_sync_observer_fn</td>
<td>A pointer to a callback function or event handler that monitors synchronization. This parameter is optional.</td>
</tr>
<tr>
<td>partial_download_retained</td>
<td>ul_bool</td>
<td>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.</td>
</tr>
<tr>
<td>password</td>
<td>const char *</td>
<td>A string specifying the existing MobiLink password associated with the user name. This parameter is optional.</td>
</tr>
<tr>
<td>ping</td>
<td>ul_bool</td>
<td>Confirm communications between the UltraLite client and the MobiLink server. When this parameter is set to true, no synchronization takes place.</td>
</tr>
<tr>
<td>publications</td>
<td>const char *</td>
<td>A comma separated list of publications indicating what data to include in the synchronization.</td>
</tr>
<tr>
<td>resume_partial_download</td>
<td>ul_bool</td>
<td>Resume a failed download. The synchronization does not upload changes; it only downloads those changes that were to be downloaded in the failed download.</td>
</tr>
<tr>
<td>send_download_ack</td>
<td>ul_bool</td>
<td>Instructs the MobiLink server whether or not the client provides download acknowledgements.</td>
</tr>
<tr>
<td>stream</td>
<td>const char *</td>
<td>The MobiLink network protocol to use for synchronization.</td>
</tr>
<tr>
<td>stream_error</td>
<td>ul_stream_error</td>
<td>The structure to hold communications error reporting information.</td>
</tr>
<tr>
<td>stream_parms</td>
<td>const char *</td>
<td>The options to configure the network protocol you selected.</td>
</tr>
<tr>
<td>upload_ok</td>
<td>ul_bool</td>
<td>The status of data uploaded to the MobiLink server. This field reports true if upload succeeded.</td>
</tr>
<tr>
<td>upload_only</td>
<td>ul_bool</td>
<td>Do not download any changes from the consolidated database during the current synchronization. This can save communication time, especially over slow communication links.</td>
</tr>
</tbody>
</table>
### 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

```c
public typedef struct ul_sync_result
```

### Members

<table>
<thead>
<tr>
<th>Member name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>auth_status</td>
<td>ul_auth_status</td>
<td>The synchronization authentication status.</td>
</tr>
<tr>
<td>auth_value</td>
<td>ul_s_long</td>
<td>The value used by the MobiLink server to determine the auth_status result.</td>
</tr>
<tr>
<td>error_status</td>
<td>ul_error_info</td>
<td>The error status from the last synchronization.</td>
</tr>
<tr>
<td>ignored_rows</td>
<td>ul_bool</td>
<td>True if uploaded rows were ignored; false otherwise.</td>
</tr>
<tr>
<td>partial_download_retained</td>
<td>ul_bool</td>
<td>The value that tells you that a partial download was retained. See keep_partial_download.</td>
</tr>
<tr>
<td>received</td>
<td>ul_sync_stats_download</td>
<td>Download statistics.</td>
</tr>
</tbody>
</table>
ul_sync_stats_upload structure
Reports the upload statistics of the synchronization stream.

Syntax
```c
public typedef struct ul_sync_stats_upload
```

Members

<table>
<thead>
<tr>
<th>Member name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sent</td>
<td>ul_sync_stats_upload</td>
<td>Upload statistics.</td>
</tr>
<tr>
<td>stream_error</td>
<td>ul_stream_error</td>
<td>The communication stream error information.</td>
</tr>
<tr>
<td>timestamp</td>
<td>SQLDATETIME</td>
<td>The time and date of the last synchronization.</td>
</tr>
<tr>
<td>upload_ok</td>
<td>ul_bool</td>
<td>True if the upload was successful; false otherwise.</td>
</tr>
</tbody>
</table>

ul_sync_stats_download structure
Reports the download statistics of the synchronization stream.

Syntax
```c
public typedef struct ul_sync_stats_download
```

Members

<table>
<thead>
<tr>
<th>Member name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>bytes</td>
<td>ul_ulong</td>
<td>The number of bytes currently received.</td>
</tr>
<tr>
<td>deletes</td>
<td>ul_ulong</td>
<td>The number of rows currently received that have been deleted.</td>
</tr>
<tr>
<td>ignored_deletes</td>
<td>ul_ulong</td>
<td>The number of rows in the current download that have been received and do not exist in the table.</td>
</tr>
<tr>
<td>ignored_updates</td>
<td>ul_ulong</td>
<td>The number of rows in the current download that have been received and are duplicates of rows that already exist in the table.</td>
</tr>
<tr>
<td>inserts</td>
<td>ul_ulong</td>
<td>The number of rows currently inserted.</td>
</tr>
<tr>
<td>truncate_deletes</td>
<td>ul_ulong</td>
<td>The number of rows that have been deleted by a truncate operation.</td>
</tr>
<tr>
<td>updates</td>
<td>ul_ulong</td>
<td>The number of rows currently received that have been updated.</td>
</tr>
</tbody>
</table>
Members

<table>
<thead>
<tr>
<th>Member name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>bytes</td>
<td>ul_u_long</td>
<td>The number of bytes currently sent.</td>
</tr>
<tr>
<td>deletes</td>
<td>ul_u_long</td>
<td>The number of deleted rows currently sent.</td>
</tr>
<tr>
<td>inserts</td>
<td>ul_u_long</td>
<td>The number of rows currently inserted.</td>
</tr>
<tr>
<td>updates</td>
<td>ul_u_long</td>
<td>The number of updated rows currently sent.</td>
</tr>
</tbody>
</table>

**ul_sync_status structure**

Returns synchronization progress monitoring data.

**Syntax**

```c
public typedef struct ul_sync_status
```

**Members**

<table>
<thead>
<tr>
<th>Member name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>current_download_row_count</td>
<td>ul_u_long</td>
<td>The number of rows that have been downloaded so far. This number includes duplicate rows that aren't included in received.inserts, received.updates, or received.deletes.</td>
</tr>
<tr>
<td>db_table_count</td>
<td>ul_u_short</td>
<td>The number of tables in database.</td>
</tr>
<tr>
<td>flags</td>
<td>ul_u_short</td>
<td>The current synchronization flags indicating additional information relating to the current state.</td>
</tr>
<tr>
<td>info</td>
<td>ul_sync_info *</td>
<td>A pointer to the ul_sync_info_a structure.</td>
</tr>
<tr>
<td>received</td>
<td>ul_sync_stats_download</td>
<td>Download statistics.</td>
</tr>
<tr>
<td>sent</td>
<td>ul_sync_stats_upload</td>
<td>Upload statistics.</td>
</tr>
<tr>
<td>sqlca</td>
<td>SQLCA *</td>
<td>The connection's active SQLCA.</td>
</tr>
<tr>
<td>state</td>
<td>ul_sync_state</td>
<td>One of the many supported states.</td>
</tr>
<tr>
<td>stop</td>
<td>ul_bool</td>
<td>A boolean that cancels synchronization. A value of true means that synchronization is canceled.</td>
</tr>
<tr>
<td>sync_table_count</td>
<td>ul_u_short</td>
<td>The number of tables being synchronized.</td>
</tr>
<tr>
<td>Member name</td>
<td>Type</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------------</td>
<td>------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>sync_table_index</td>
<td>ul_u_short</td>
<td>1 .. sync_table_count</td>
</tr>
<tr>
<td>table_id</td>
<td>ul_u_short</td>
<td>The current table ID that is being uploaded or downloaded (1-based). This number may skip values when not all tables are being synchronized, and is not necessarily increasing.</td>
</tr>
<tr>
<td>table_name</td>
<td>char</td>
<td>The name of the current table.</td>
</tr>
<tr>
<td>table_name_w2</td>
<td>ul_wchar</td>
<td>The name of the current table.</td>
</tr>
<tr>
<td>total_download_row_count</td>
<td>ul_u_long</td>
<td>The total number of rows to be received in the download. This number includes duplicate rows that aren't included in received.inserts, received.updates, or received.deletes. This field is not set until the synchronization enters the UL_SYNC_STATE_RECEIVING_TABLE state for the first table.</td>
</tr>
<tr>
<td>user_data</td>
<td>ul_void</td>
<td>User data passed in to the ULSetSynchronizationCallback method or set in the ul_sync_info structure.</td>
</tr>
</tbody>
</table>

See also
- “ul_sync_state enumeration [UltraLite C and Embedded SQL datatypes]” on page 87

### ul_validate_data structure

Stores validation status information for the callback.

**Syntax**

```c
public typedef struct ul_validate_data
```

**Members**

<table>
<thead>
<tr>
<th>Member name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>ul_u_long</td>
<td>Parameter as an integer.</td>
</tr>
<tr>
<td>parm_count</td>
<td>ul_u_short</td>
<td>The number of parameters in the structure.</td>
</tr>
<tr>
<td>parm_type</td>
<td>enumeration</td>
<td>The possible parameter types.</td>
</tr>
<tr>
<td>parms</td>
<td>struct ul_validate_data::@3</td>
<td>Array of parameters.</td>
</tr>
<tr>
<td>s</td>
<td>char</td>
<td>Parameter as a string (this is not a wide char)</td>
</tr>
<tr>
<td>Member name</td>
<td>Type</td>
<td>Description</td>
</tr>
<tr>
<td>----------------------</td>
<td>------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>status_id</td>
<td>ul_validate_status_id</td>
<td>Indicates what is being reported in the validation process.</td>
</tr>
<tr>
<td>stop</td>
<td>ul_bool</td>
<td>A boolean that cancels the validation. A value of true means that validation is canceled.</td>
</tr>
<tr>
<td>type</td>
<td>parm_type</td>
<td>Type of parameter stored.</td>
</tr>
<tr>
<td>user_data</td>
<td>ul_void *</td>
<td>User-defined data pointer passed into validation routine.</td>
</tr>
</tbody>
</table>

**ULVF_DATABASE variable**

Used to validate database.

**Syntax**

```c
#define ULVF_DATABASE
```

**Remarks**

Verify all database pages using page checksums and additional checks.

**See also**

- “ULDatabaseManager.ValidateDatabase method [UltraLite C++]” on page 137
- “ULConnection.ValidateDatabase method [UltraLite C++]” on page 127

**ULVF_EXPRESS variable**

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

**Syntax**

```c
#define ULVF_EXPRESS
```

**Remarks**

This flag modifies others specified.

**See also**

- “ULDatabaseManager.ValidateDatabase method [UltraLite C++]” on page 137
- “ULConnection.ValidateDatabase method [UltraLite C++]” on page 127

**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 137
● “ULConnection.ValidateDatabase method [UltraLite C++]” on page 127

**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 137
● “ULConnection.ValidateDatabase method [UltraLite C++]” on page 127

**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 137
● “ULConnection.ValidateDatabase method [UltraLite C++]” on page 127

**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 137
- “ULConnection.ValidateDatabase method [UltraLite C++]” on page 127

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 137
- “ULConnection.ValidateDatabase method [UltraLite C++]” on page 127

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
- “ActiveSync synchronization setup” on page 66

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 92
- “UL_SYNC_ALL variable [UltraLite C and Embedded SQL datatypes]” on page 100

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 92
- “UL_SYNC_ALL variable [UltraLite C and Embedded SQL datatypes]” on page 100

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

```c
#define UL_VALID_IS_ERROR
```

### UL_VALID_IS_INFO variable

Evaluates true if a given ul_validate_status_id is an informational status.

Syntax

```c
#define UL_VALID_IS_INFO
```

## UltraLite C/C++ API reference

The following list describes some of the commonly used API 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.

### Header file

- ulcpp.h

### See also

- “ULDatabaseManager class [UltraLite C++]” on page 128
- “ULConnection class [UltraLite C++]” on page 102
- “ULTable class [UltraLite C++]” on page 198
- “ULPreparedStatement class [UltraLite C++]” on page 148
- “ULResultSet class [UltraLite C++]” on page 158
- “ULResultSetSchema class [UltraLite C++]” on page 193

## ULConnection class

Represents a connection to an UltraLite database.

Syntax

```java
public class ULConnection
```
## Members

All members of the ULConnection class, including all inherited members.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CancelGetNotification method</td>
<td>Cancels any pending get-notification calls on all queues matching the given name.</td>
</tr>
<tr>
<td>ChangeEncryptionKey method</td>
<td>Changes the database encryption key for an UltraLite database.</td>
</tr>
<tr>
<td>Checkpoint method</td>
<td>Performs a checkpoint operation, flushing any pending committed transactions to the database.</td>
</tr>
<tr>
<td>Close method</td>
<td>Destroys this connection and any remaining associated objects.</td>
</tr>
<tr>
<td>Commit method</td>
<td>Commits the current transaction.</td>
</tr>
<tr>
<td>CountUploadRows method</td>
<td>Counts the number of rows that need to be uploaded for synchronization.</td>
</tr>
<tr>
<td>CreateNotificationQueue method</td>
<td>Creates an event notification queue for this connection.</td>
</tr>
<tr>
<td>DeclareEvent method</td>
<td>Declares an event which can then be registered for and triggered.</td>
</tr>
<tr>
<td>DestroyNotificationQueue method</td>
<td>Destroys the given event notification queue.</td>
</tr>
<tr>
<td>ExecuteScalar method</td>
<td>Executes a SQL SELECT statement directly, returning a single result.</td>
</tr>
<tr>
<td>ExecuteScalarV method</td>
<td>Executes a SQL SELECT statement string, along with a list of substitution values.</td>
</tr>
<tr>
<td>ExecuteStatement method</td>
<td>Executes a SQL statement string directly.</td>
</tr>
<tr>
<td>GetChildObjectCount method</td>
<td>Gets the number of currently open child objects on the connection.</td>
</tr>
<tr>
<td>GetDatabaseProperty method</td>
<td>Obtains the value of a database property.</td>
</tr>
<tr>
<td>GetDatabasePropertyInt method</td>
<td>Obtains the integer value of a database property.</td>
</tr>
<tr>
<td>GetDatabaseSchema method</td>
<td>Returns an object pointer used to query the schema of the database.</td>
</tr>
<tr>
<td>GetLastDownloadTime method</td>
<td>Obtains the last time a specified publication was downloaded.</td>
</tr>
<tr>
<td>GetLastError method</td>
<td>Returns the error information associated with the last call.</td>
</tr>
<tr>
<td>GetLastIdentity method</td>
<td>Gets the @@identity value.</td>
</tr>
<tr>
<td>Name</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>GetNotification method</td>
<td>Reads an event notification.</td>
</tr>
<tr>
<td>GetNotificationParameter method</td>
<td>Gets a parameter for the event notification just read by the GetNotification method.</td>
</tr>
<tr>
<td>GetSqlca method</td>
<td>Gets the communication area associated with this connection.</td>
</tr>
<tr>
<td>GetSyncResult method</td>
<td>Gets the result of the last synchronization.</td>
</tr>
<tr>
<td>GetUserPointer method</td>
<td>Gets the pointer value last set by the SetUserPointer method.</td>
</tr>
<tr>
<td>GlobalAutoincUsage method</td>
<td>Obtains the percent of the default values used in all the columns that have global autoincrement defaults.</td>
</tr>
<tr>
<td>GrantConnectTo method</td>
<td>Grants access to an UltraLite database for a new or existing user ID with the given password.</td>
</tr>
<tr>
<td>InitSyncInfo method</td>
<td>Initializes the synchronization information structure.</td>
</tr>
<tr>
<td>OpenTable method</td>
<td>Opens a table.</td>
</tr>
<tr>
<td>PrepareStatement method</td>
<td>Prepares a SQL statement.</td>
</tr>
<tr>
<td>RegisterForEvent method</td>
<td>Registers or unregisters a queue to receive notifications of an event.</td>
</tr>
<tr>
<td>ResetLastDownloadTime method</td>
<td>Resets the last download time of a publication so that the application resynchronizes previously downloaded data.</td>
</tr>
<tr>
<td>RevokeConnectFrom method</td>
<td>Revokes access from an UltraLite database for a user ID.</td>
</tr>
<tr>
<td>Rollback method</td>
<td>Rolls back the current transaction.</td>
</tr>
<tr>
<td>RollbackPartialDownload method</td>
<td>Rolls back the changes from a failed synchronization.</td>
</tr>
<tr>
<td>SendNotification method</td>
<td>Sends a notification to all queues matching the given name.</td>
</tr>
<tr>
<td>SetDatabaseOption method</td>
<td>Sets the specified database option.</td>
</tr>
<tr>
<td>SetDatabaseOptionInt method</td>
<td>Sets a database option.</td>
</tr>
<tr>
<td>SetSynchronizationCallback method</td>
<td>Sets the callback to be invoked while performing a synchronization.</td>
</tr>
<tr>
<td>SetSyncInfo method</td>
<td>Creates a synchronization profile using the given name based on the given ul_sync_info structure.</td>
</tr>
<tr>
<td>Name</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>SetUserPointer method</td>
<td>Sets an arbitrary pointer value in the connection for use by the calling application.</td>
</tr>
<tr>
<td>StartSynchronizationDelete method</td>
<td>Sets START SYNCHRONIZATION DELETE for this connection.</td>
</tr>
<tr>
<td>StopSynchronizationDelete method</td>
<td>Sets STOP SYNCHRONIZATION DELETE for this connection.</td>
</tr>
<tr>
<td>Synchronize method</td>
<td>Initiates synchronization in an UltraLite application.</td>
</tr>
<tr>
<td>SynchronizeFromProfile method</td>
<td>Synchronizes the database using the given profile and merge parameters.</td>
</tr>
<tr>
<td>TriggerEvent method</td>
<td>Triggers a user-defined event and sends notifications to all registered queues.</td>
</tr>
<tr>
<td>ValidateDatabase method</td>
<td>Validates the database on this connection.</td>
</tr>
</tbody>
</table>

**CancelGetNotification method**

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

**Syntax**

```csharp
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**

```csharp
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

```cpp
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 119

DeclareEvent method

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

Syntax

```cpp
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 126

**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 (S QE_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:

```c
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:

```c
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 86

**ExecuteScalarV method**

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

**Syntax**

```cpp
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 (SQLITE_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 86

**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 119

**GetChildObjectCount method**
Gets the number of currently open child objects on the connection.
Syntax

```c
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

```c
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

- “UltraLite database properties” [UltraLite - Database Management and Reference]

Example

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

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

**GetDatabasePropertyInt method**

Obtains the integer value of a database property.

Syntax

```c
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
   ● “UltraLite 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 140

**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 105
- “ULConnection.GetNotificationParameter method [UltraLite C++]” on page 116

### GetNotificationParameter method

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

Syntax

```csharp
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 115
**GetSqlca method**

Gets the communication area associated with this connection.

**Syntax**

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

**Returns**

A pointer to the SQLCA object for this connection.

**GetSyncResult method**

Gets the result of the last synchronization.

**Syntax**

```csharp
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 94

**GetUserPointer method**

Gets the pointer value last set by the SetUserPointer method.

**Syntax**

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

**See also**

- “ULConnection.SetUserPointer method [UltraLite C++]” on page 124

**GlobalAutoincUsage method**

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

**Syntax**

```csharp
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 121

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

```cpp
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

```cpp
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

```cpp
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

```csharp
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:

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

**RevokeConnectFrom method**

RevokeConnectFrom method

Revises access from an UltraLite database for a user ID.

**Syntax**

```cpp
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**

Rollback method

Rolls back the current transaction.

**Syntax**

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

**Returns**

True on success, otherwise false.

**RollbackPartialDownload method**

RollbackPartialDownload method

Rolls back the changes from a failed synchronization.

**Syntax**

```cpp
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**

```csharp
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 116

**SetDatabaseOption method**

Sets the specified database option.

**Syntax**

```csharp
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,

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**

```c
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**

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

**Returns**

True on success, otherwise false.

**StopSynchronizationDelete method**

Sets STOP SYNCHRONIZATION DELETE for this connection.

**Syntax**

```c
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 132
  ● “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 125
- “SYNCHRONIZE statement [UltraLite]” [UltraLite - Database Management and Reference]

---

**TriggerEvent method**

Triggers a user-defined event and sends notifications to all registered queues.

**Syntax**

```cpp
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 116
**ValidateDatabase method**

Validates the database on this connection.

**Syntax**

```csharp
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 100
- “ULVF_INDEX variable [UltraLite C and Embedded SQL datatypes]” on page 99
- “ULVF_DATABASE variable [UltraLite C and Embedded SQL datatypes]” on page 98
- “ULVF_EXPRESS variable [UltraLite C and Embedded SQL datatypes]” on page 98
- “ULVF_FULL_VALIDATE variable [UltraLite C and Embedded SQL datatypes]” on page 98
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 the ULDatabaseManager class, including all inherited members.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CreateDatabase method</td>
<td>Creates a new database.</td>
</tr>
<tr>
<td>DropDatabase method</td>
<td>Erases an existing database that is not currently running.</td>
</tr>
<tr>
<td>EnableAesDBEncryption method</td>
<td>Enables AES database encryption.</td>
</tr>
<tr>
<td>EnableAesFipsDBEncryption method</td>
<td>Enables FIPS 140-2 certified AES database encryption.</td>
</tr>
<tr>
<td>EnableAllSynchronization method</td>
<td>Enables all four types of synchronization: TCPIP, HTTP, TLS, and HTTPS.</td>
</tr>
<tr>
<td>EnableHttpsSynchronization method</td>
<td>Enables HTTPS synchronization.</td>
</tr>
<tr>
<td>EnableHttpSynchronization method</td>
<td>Enables HTTP synchronization.</td>
</tr>
<tr>
<td>EnableRsaE2ee method</td>
<td>Enables RSA end-to-end encryption.</td>
</tr>
<tr>
<td>EnableRsaFipsE2ee method</td>
<td>Enables FIPS 140-2 certified RSA end-to-end encryption.</td>
</tr>
<tr>
<td>EnableRsaFipsSyncEncryption method</td>
<td>Enables FIPS 140-2 certified RSA synchronization encryption for SSL or TLS streams.</td>
</tr>
<tr>
<td>EnableRsaSyncEncryption method</td>
<td>Enables RSA synchronization encryption.</td>
</tr>
<tr>
<td>EnableTcpipSynchronization method</td>
<td>Enables TCP/IP synchronization.</td>
</tr>
<tr>
<td>EnableTlsSynchronization method</td>
<td>Enables TLS synchronization.</td>
</tr>
<tr>
<td>EnableZlibSyncCompression method</td>
<td>Enables Zlib compression for a synchronization stream.</td>
</tr>
<tr>
<td>Fini method</td>
<td>Finalizes the UltraLite runtime.</td>
</tr>
</tbody>
</table>
### CreateDatabase method

Creates a new database.

**Syntax**

```cpp
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:

```cpp
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**

```cpp
public static bool DropDatabase(const char * params, ULError * error)
```

**Parameters**

- `params`  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**

```cpp
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.

FIPS-certified encryption requires a separate license. All strong encryption technologies are subject to
export regulations.

See “Separately licensed components” [SQL Anywhere 16 - Introduction].

See also

● “ULDatabaseManager.EnableAesDBEncryption method [UltraLite C++]” on page 130
● “UltraLite DBKEY connection parameter” [UltraLite - Database Management and Reference]

EnableAllSynchronization method

Enables all four types of synchronization: TCPIP, HTTP, TLS, and HTTPS.

Syntax

public static void EnableAllSynchronization()

Remarks

You must call this method before the Synchronize method.

When initiating synchronization, set the stream parameter to "TCPIP", "HTTP", "TLS", or "HTTPS".
Also set the network protocol certificate options if using TLS or HTTPS

See also

● “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.

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 `fips` option 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 `fips` option must be set to "yes".

See also

- "ULDatabaseManager.EnableRsaSyncEncryption method [UltraLite C++]" on page 133
- "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.

See also

- "MobiLink client network protocol options" [MobiLink - Client Administration]
EnableTcpipSynchronization method

Enables TCP/IP synchronization.

Syntax

```java
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

```java
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

```java
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**
```java
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**
```java
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**
```java
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

```java
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

```c
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 100
- “ULVF_INDEX variable [UltraLite C and Embedded SQL datatypes]” on page 99
- “ULVF_DATABASE variable [UltraLite C and Embedded SQL datatypes]” on page 98
- “ULVF_EXPRESS variable [UltraLite C and Embedded SQL datatypes]” on page 98
- “ULVF_FULL_VALIDATE variable [UltraLite C and Embedded SQL datatypes]” on page 98

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**

```java
public class ULDatabaseSchema
```

**Members**

All members of the ULDatabaseSchema class, including all inherited members.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Close method</td>
<td>Destroys this object.</td>
</tr>
<tr>
<td>GetConnection method</td>
<td>Gets the ULConnection object.</td>
</tr>
<tr>
<td>GetNextPublication method</td>
<td>Gets the name of the next publication in the database.</td>
</tr>
<tr>
<td>GetNextTable method</td>
<td>Gets the next table (schema) in the database.</td>
</tr>
<tr>
<td>GetPublicationCount method</td>
<td>Gets the number of publications in the database.</td>
</tr>
<tr>
<td>GetTableCount method</td>
<td>Returns the number of tables in the database.</td>
</tr>
<tr>
<td>GetTableSchema method</td>
<td>Returns the schema of the named table.</td>
</tr>
</tbody>
</table>

**Close method**

Destroys this object.

**Syntax**

```java
public virtual void Close()
```

**GetConnection method**

Gets the ULConnection object.

**Syntax**

```java
public virtual ULConnection * GetConnection()
```

**Returns**

The ULConnection associated with this object.
**GetNextPublication method**

Gets the name of the next publication in the database.

**Syntax**

```c
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 216

---

**GetNextTable method**

Gets the next table (schema) in the database.

**Syntax**

```c
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 216

---

**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 ULTTableSchema * GetTableSchema(const char * tableName)

Parameters
    ● **tableName**  The name of the table.

Returns
    A ULTTableSchema 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 the ULError class, including all inherited members.
<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ULError constructor</td>
<td>Constructs a ULError object.</td>
</tr>
<tr>
<td>Clear method</td>
<td>Clears the current error.</td>
</tr>
<tr>
<td>GetErrorInfo method</td>
<td>Returns a pointer to the underlying ul_error_info object.</td>
</tr>
<tr>
<td>GetParameter method</td>
<td>Copies the specified error parameter into the provided buffer.</td>
</tr>
<tr>
<td>GetParameterCount method</td>
<td>Returns the number of error parameters.</td>
</tr>
<tr>
<td>GetSQLCode method</td>
<td>Returns the SQLCODE error code for the last operation.</td>
</tr>
<tr>
<td>GetSQLCount method</td>
<td>Returns a value that depends on the last operation, and the result of that operation.</td>
</tr>
<tr>
<td>GetString method</td>
<td>Returns the description of the current error.</td>
</tr>
<tr>
<td>GetURL method</td>
<td>Returns a URL to the documentation page for this error.</td>
</tr>
<tr>
<td>IsOK method</td>
<td>Tests the error code.</td>
</tr>
</tbody>
</table>

**ULError constructor**

Constructs a ULError object.

**Syntax**

```java
public ULError()
```

**Clear method**

Clears the current error.

**Syntax**

```java
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.
GetErrorInfo() method

Returns a pointer to the underlying ul_error_info object.

Syntax

```java
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 91

GetErrorInfo() method

Returns a pointer to the underlying ul_error_info object.

Syntax

```java
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 91

GetParameter method

Copies the specified error parameter into the provided buffer.

Syntax

```java
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**

```c
public ul_u_short GetParameterCount()
```

**Returns**

The number of error parameters.

### GetSQLCode method

Returns the SQLCODE error code for the last operation.

**Syntax**

```c
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**

```c
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 144

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**
```csharp
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**
```csharp
public class ULIndexSchema
```

**Members**
All members of the ULIndexSchema class, including all inherited members.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Close method</td>
<td>Destroys this object.</td>
</tr>
<tr>
<td>GetColumnCount method</td>
<td>Gets the number of columns in the index.</td>
</tr>
<tr>
<td>GetColumnName method</td>
<td>Gets the name of the column given the position of the column in the index.</td>
</tr>
<tr>
<td>GetConnection method</td>
<td>Gets the ULConnection object.</td>
</tr>
<tr>
<td>GetIndexColumnID method</td>
<td>Gets the 1-based index column ID from its name.</td>
</tr>
<tr>
<td>GetIndexFlags method</td>
<td>Gets the index property flags bit field.</td>
</tr>
<tr>
<td>GetName method</td>
<td>Gets the name of the index.</td>
</tr>
<tr>
<td>GetReferencedIndexName method</td>
<td>Gets the associated primary index name.</td>
</tr>
<tr>
<td>GetReferencedTableName method</td>
<td>Gets the associated primary table name.</td>
</tr>
<tr>
<td>GetTableName method</td>
<td>Gets the name of the table containing this index.</td>
</tr>
<tr>
<td>IsColumnDescending method</td>
<td>Determines if the column is in descending order.</td>
</tr>
</tbody>
</table>
**Close method**
Destroys this object.

**Syntax**
```csharp
public virtual void Close()
```

**GetColumnCount method**
Gets the number of columns in the index.

**Syntax**
```csharp
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**
```csharp
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**
```csharp
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 214

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

```csharp
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

```csharp
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

```csharp
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

```csharp
public class ULPreparedStatement
```
Members

All members of the ULPreparedStatement class, including all inherited members.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AppendParameterByteChunk method</td>
<td>Sets a large binary parameter broken down into several chunks.</td>
</tr>
<tr>
<td>AppendParameterStringChunk method</td>
<td>Sets a large string parameter broken down into several chunks.</td>
</tr>
<tr>
<td>Close method</td>
<td>Destroys this object.</td>
</tr>
<tr>
<td>ExecuteQuery method</td>
<td>Executes a SQL SELECT statement as a query.</td>
</tr>
<tr>
<td>ExecuteStatement method</td>
<td>Executes a statement that does not return a result set, such as a SQL INSERT, DELETE or UPDATE statement.</td>
</tr>
<tr>
<td>GetConnection method</td>
<td>Gets the connection object.</td>
</tr>
<tr>
<td>GetParameterCount method</td>
<td>Gets the number of input parameters for this statement.</td>
</tr>
<tr>
<td>GetParameterID method</td>
<td>Get the 1-based ordinal for a parameter name.</td>
</tr>
<tr>
<td>GetParameterType method</td>
<td>Gets the storage/host variable type of a parameter.</td>
</tr>
<tr>
<td>GetPlan method</td>
<td>Gets a text-based description of the query execution plan.</td>
</tr>
<tr>
<td>GetResultSetSchema method</td>
<td>Gets the schema for the result set.</td>
</tr>
<tr>
<td>GetRowsAffectedCount method</td>
<td>Gets the number of rows affected by the last statement.</td>
</tr>
<tr>
<td>HasResultSet method</td>
<td>Determines if the SQL statement has a result set.</td>
</tr>
<tr>
<td>SetParameterBinary method</td>
<td>Sets a parameter to a ul_binary value.</td>
</tr>
<tr>
<td>SetParameterDateTime method</td>
<td>Sets a parameter to a DECL_DATETIME value.</td>
</tr>
<tr>
<td>SetParameterDouble method</td>
<td>Sets a parameter to a double value.</td>
</tr>
<tr>
<td>SetParameterFloat method</td>
<td>Sets a parameter to a float value.</td>
</tr>
<tr>
<td>SetParameterGuid method</td>
<td>Sets a parameter to a GUID value.</td>
</tr>
<tr>
<td>SetParameterInt method</td>
<td>Sets a parameter to an integer value.</td>
</tr>
<tr>
<td>SetParameterIntWithType method</td>
<td>Sets a parameter to an integer value of the specified integer type.</td>
</tr>
<tr>
<td>Name</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------------------------------</td>
<td>---------------------------------------</td>
</tr>
<tr>
<td>SetParameterNull method</td>
<td>Sets a parameter to null.</td>
</tr>
<tr>
<td>SetParameterString method</td>
<td>Sets a parameter to a string value.</td>
</tr>
</tbody>
</table>

**AppendParameterByteChunk method**

Sets a large binary parameter broken down into several chunks.

**Syntax**

```csharp
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**

```csharp
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 86

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

```cpp
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

```cpp
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**

```csharp
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**

```csharp
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**

```csharp
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**

```csharp
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**

```csharp
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**

```csharp
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**
```csharp
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**
```csharp
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 86

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 150

**ULResultSet class**

Represents a result set in an UltraLite database.

**Syntax**

```java
public class ULResultSet
```

**Derived classes**

- “ULTable class [UltraLite C++]” on page 198

**Members**

All members of the ULResultSet class, including all inherited members.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AfterLast method</td>
<td>Moves the cursor after the last row.</td>
</tr>
<tr>
<td>AppendByteChunk method</td>
<td>Appends bytes to a column.</td>
</tr>
<tr>
<td>AppendStringChunk method</td>
<td>Appends a string chunk to a column.</td>
</tr>
<tr>
<td>BeforeFirst method</td>
<td>Moves the cursor before the first row.</td>
</tr>
<tr>
<td>Close method</td>
<td>Destroys this object.</td>
</tr>
<tr>
<td>Delete method</td>
<td>Deletes the current row and moves it to the next valid row.</td>
</tr>
<tr>
<td>DeleteNamed method</td>
<td>Deletes the current row and moves it to the next valid row.</td>
</tr>
<tr>
<td>First method</td>
<td>Moves the cursor to the first row.</td>
</tr>
<tr>
<td>GetBinary method</td>
<td>Fetches a value from a column as a ul_binary value.</td>
</tr>
<tr>
<td>GetBinaryLength method</td>
<td>Gets the binary length of the value of a column.</td>
</tr>
<tr>
<td>GetByteChunk method</td>
<td>Gets a binary chunk from the column.</td>
</tr>
<tr>
<td>GetConnection method</td>
<td>Gets the connection object.</td>
</tr>
<tr>
<td>GetDateTime method</td>
<td>Fetches a value from a column as a DECL_DATETIME.</td>
</tr>
<tr>
<td>Name</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>GetDouble method</td>
<td>Fetches a value from a column as a double.</td>
</tr>
<tr>
<td>GetFloat method</td>
<td>Fetches a value from a column as a float.</td>
</tr>
<tr>
<td>GetGuid method</td>
<td>Fetches a value from a column as a GUID.</td>
</tr>
<tr>
<td>GetInt method</td>
<td>Fetches a value from a column as an integer.</td>
</tr>
<tr>
<td>GetIntWithType method</td>
<td>Fetches a value from a column as the specified integer type.</td>
</tr>
<tr>
<td>GetResultSetSchema method</td>
<td>Returns an object that can be used to get information about the result set.</td>
</tr>
<tr>
<td>GetRowCount method</td>
<td>Gets the number of rows in the table.</td>
</tr>
<tr>
<td>GetState method</td>
<td>Gets the internal state of the cursor.</td>
</tr>
<tr>
<td>GetString method</td>
<td>Fetches a value from a column as a null-terminated string.</td>
</tr>
<tr>
<td>GetStringChunk method</td>
<td>Gets a string chunk from the column.</td>
</tr>
<tr>
<td>GetStringLength method</td>
<td>Gets the string length of the value of a column.</td>
</tr>
<tr>
<td>IsNull method</td>
<td>Checks if a column is NULL.</td>
</tr>
<tr>
<td>Last method</td>
<td>Moves the cursor to the last row.</td>
</tr>
<tr>
<td>Next method</td>
<td>Moves the cursor forward one row.</td>
</tr>
<tr>
<td>Previous method</td>
<td>Moves the cursor back one row.</td>
</tr>
<tr>
<td>Relative method</td>
<td>Moves the cursor by offset rows from the current cursor position.</td>
</tr>
<tr>
<td>SetBinary method</td>
<td>Sets a column to a ul_binary value.</td>
</tr>
<tr>
<td>SetDateTime method</td>
<td>Sets a column to a DECL_DATETIME value.</td>
</tr>
<tr>
<td>SetDefault method</td>
<td>Sets a column to its default value.</td>
</tr>
<tr>
<td>SetDouble method</td>
<td>Sets a column to a double value.</td>
</tr>
<tr>
<td>SetFloat method</td>
<td>Sets a column to a float value.</td>
</tr>
<tr>
<td>SetGuid method</td>
<td>Sets a column to a GUID value.</td>
</tr>
<tr>
<td>SetInt method</td>
<td>Sets a column to an integer value.</td>
</tr>
<tr>
<td>SetIntWithType method</td>
<td>Sets a column to an integer value of the specified integer type.</td>
</tr>
</tbody>
</table>
### Name

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SetNull method</td>
<td>Sets a column to null.</td>
</tr>
<tr>
<td>SetString method</td>
<td>Sets a column to a string value.</td>
</tr>
<tr>
<td>Update method</td>
<td>Updates the current row.</td>
</tr>
<tr>
<td>UpdateBegin method</td>
<td>Selects the update mode for setting columns.</td>
</tr>
</tbody>
</table>

#### AfterLast method

Moves the cursor after the last row.

**Syntax**

```csharp
public virtual bool AfterLast()
```

**Returns**

True on success; otherwise, returns false.

#### AppendByteChunk method

Appends bytes to a column.

**Overload list**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AppendByteChunk(const char *, const ul_byte *, size_t) method</td>
<td>Appends bytes to a column.</td>
</tr>
<tr>
<td>AppendByteChunk(ul_column_num, const ul_byte *, size_t) method</td>
<td>Appends bytes to a column.</td>
</tr>
</tbody>
</table>

#### AppendByteChunk(const char *, const ul_byte *, size_t) method

Appends bytes to a column.

**Syntax**

```csharp
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 160

---

#### AppendByteChunk(ul_column_num, const ul_byte *, size_t) method

Appends bytes to a column.

**Syntax**
```cpp
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 160

---

#### AppendStringChunk method

Appends a string chunk to a column.
AppendStringChunk(const char *, const char *, size_t) method

Appends a string chunk to a column.

Syntax

```cpp
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 161

AppendStringChunk(ul_column_num, const char *, size_t) method

Appends a string chunk to a column.

Syntax

```cpp
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 161

**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**

```c
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**

```c
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**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GetBinary(const char *, p_ul_binary, size_t) method</td>
<td>Fetches a value from a column as a ul_binary value.</td>
</tr>
<tr>
<td>GetBinary(ul_column_num, p_ul_binary, size_t) method</td>
<td>Fetches a value from a column as a ul_binary value.</td>
</tr>
</tbody>
</table>

**GetBinary(const char *, p_ul_binary, size_t) method**

Fetches a value from a column as a ul_binary value.

**Syntax**

```c
public virtual bool GetBinary(
    const char * cname,
    p_ul_binary dst,
```
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

```cpp
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**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GetBinaryLength(const char *) method</td>
<td>Gets the binary length of the value of a column.</td>
</tr>
<tr>
<td>GetBinaryLength(ul_column_num) method</td>
<td>Gets the binary length of the value of a column.</td>
</tr>
</tbody>
</table>
**GetBinaryLength(const char *) method**

Gets the binary length of the value of a column.

**Syntax**

```csharp
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**

```csharp
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**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GetByteChunk(const char *, ul_byte *, size_t, size_t) method</td>
<td>Gets a binary chunk from the column.</td>
</tr>
<tr>
<td>GetByteChunk(ul_column_num, ul_byte *, size_t, size_t) method</td>
<td>Gets a binary chunk from the column.</td>
</tr>
</tbody>
</table>

**GetByteChunk(const char *, ul_byte *, size_t, size_t) method**

Gets a binary chunk from the column.

**Syntax**

```csharp
public virtual size_t GetByteChunk(
    const char * cname,
```
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 180

**GetByteChunk(ul_column_num, ul_byte *, size_t, size_t) method**

Gets a binary chunk from the column.

Syntax

```csharp
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 180

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_DATE-TIME.

Overload list

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GetDateTime(const char *, DECL_DATE-TIME *) method</td>
<td>Fetches a value from a column as a DECL_DATE-TIME.</td>
</tr>
<tr>
<td>GetDateTime(ul_column_num, DECL_DATE-TIME *) method</td>
<td>Fetches a value from a column as a DECL_DATE-TIME.</td>
</tr>
</tbody>
</table>

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.
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

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GetDouble(const char *) method</td>
<td>Fetches a value from a column as a double.</td>
</tr>
<tr>
<td>GetDouble(ul_column_num) method</td>
<td>Fetches a value from a column as a double.</td>
</tr>
</tbody>
</table>

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
```csharp
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

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GetFloat(char *) method</td>
<td>Fetches a value from a column as a float.</td>
</tr>
<tr>
<td>GetFloat(ul_column_num) method</td>
<td>Fetches a value from a column as a float.</td>
</tr>
</tbody>
</table>

GetFloat(char *) method
Fetches a value from a column as a float.

Syntax
```csharp
public virtual ul_real GetFloat(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
```csharp
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

Fetched a value from a column as a GUID.

Overload list

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GetGuid(const char *, GUID *) method</td>
<td>Fetches a value from a column as a GUID.</td>
</tr>
<tr>
<td>GetGuid(ul_column_num, GUID *) method</td>
<td>Fetches a value from a column as a GUID.</td>
</tr>
</tbody>
</table>

GetGuid(const char *, GUID *) method

Fetched a value from a column as a GUID.

Syntax

```cpp
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

Fetched a value from a column as a GUID.

Syntax

```cpp
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**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GetInt(const char *) method</td>
<td>Fetches a value from a column as an integer.</td>
</tr>
<tr>
<td>GetInt(ul_column_num) method</td>
<td>Fetches a value from a column as an integer.</td>
</tr>
</tbody>
</table>

**GetInt(const char *) method**
Fetches a value from a column as an integer.

**Syntax**
```c
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**
```c
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.

**GetIntWith Rodrigo**
Fetched from a column as the specified integer type.
Overload list

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GetIntWith_Type(const char *, ul_column_storage_type) method</td>
<td>Fetches a value from a column as the specified integer type.</td>
</tr>
<tr>
<td>GetIntWith_Type(ul_column_num, ul_column_storage_type) method</td>
<td>Fetches a value from a column as the specified integer type.</td>
</tr>
</tbody>
</table>

GetIntWith_Type(const char *, ul_column_storage_type) method

Fetches a value from a column as the specified integer type.

Syntax

```c
public virtual ul_s_big GetIntWith_Type(
    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 86

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 86

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 84

**GetString method**

Fetches a value from a column as a null-terminated string.
Overload list

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GetString(const char *, char *, size_t) method</td>
<td>Fetches a value from a column as a null-terminated string.</td>
</tr>
<tr>
<td>GetString(ul_column_num, char *, size_t) method</td>
<td>Fetches a value from a column as a null-terminated string.</td>
</tr>
</tbody>
</table>

**GetString(const char *, char *, size_t) method**

Fetches a value from a column as a null-terminated string.

**Syntax**

```c
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**

```c
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**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GetStringChunk(const char *, char *, size_t, size_t) method</td>
<td>Gets a string chunk from the column.</td>
</tr>
<tr>
<td>GetStringChunk(ul_column_num, char *, size_t, size_t) method</td>
<td>Gets a string chunk from the column.</td>
</tr>
</tbody>
</table>

**GetStringChunk(const char *, char *, size_t, size_t) method**

Gets a string chunk from the column.

**Syntax**

```c
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 180

GetStringChunk(ul_column_num, char *, size_t, size_t) method
Gets a string chunk from the column.

Syntax
```csharp
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 180

GetStringLength method
Gets the string length of the value of a column.
Overload list

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GetStringLength(const char *) method</td>
<td>Gets the string length of the value of a column.</td>
</tr>
<tr>
<td>GetStringLength(ul_column_num) method</td>
<td>Gets the string length of the value of a column.</td>
</tr>
</tbody>
</table>

**GetStringLength(const char *) method**

Gets the string length of the value of a column.

**Syntax**

```cpp
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:

```cpp
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:

```cpp
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 175

**GetStringLength(ul_column_num) method**

Gets the string length of the value of a column.

**Syntax**

```cpp
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:

```cpp
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:

```cpp
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 175

IsNull method
Checks if a column is NULL.

Overload list

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IsNull(const char *) method</td>
<td>Checks if a column is NULL.</td>
</tr>
<tr>
<td>IsNull(ul_column_num) method</td>
<td>Checks if a column is NULL.</td>
</tr>
</tbody>
</table>

IsNull(const char *) method
Checks if a column is NULL.

Syntax

```cpp
public virtual bool IsNull(const char * name)
```

Parameters
- **name**  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

```csharp
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

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SetBinary(const char *, p_ul_binary) method</td>
<td>Sets a column to a ul_binary value.</td>
</tr>
<tr>
<td>SetBinary(ul_column_num, p_ul_binary) method</td>
<td>Sets a column to a ul_binary value.</td>
</tr>
</tbody>
</table>

SetBinary(const char *, p_ul_binary) method

Sets a column to a ul_binary value.

Syntax

```csharp
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

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SetDateTime(const char *, DECL_DATETIME *)</td>
<td>Sets a column to a DECL_DATETIME value.</td>
</tr>
<tr>
<td>SetDateTime(ul_column_num, DECL_DATETIME *)</td>
<td>Sets a column to a DECL_DATETIME value.</td>
</tr>
</tbody>
</table>

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

```c
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

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SetDefault(const char *) method</td>
<td>Sets a column to its default value.</td>
</tr>
<tr>
<td>SetDefault(ul_column_num) method</td>
<td>Sets a column to its default value.</td>
</tr>
</tbody>
</table>

SetDefault(const char *) method

Sets a column to its default value.

Syntax

```c
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

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SetDouble(const char *, ul_double) method</td>
<td>Sets a column to a double value.</td>
</tr>
<tr>
<td>SetDouble(ul_column_num, ul_double) method</td>
<td>Sets a column to a double value.</td>
</tr>
</tbody>
</table>

**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**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>SetFloat(const char *, ul_real)</code> method</td>
<td>Sets a column to a float value.</td>
</tr>
<tr>
<td><code>SetFloat(ul_column_num, ul_real)</code> method</td>
<td>Sets a column to a float value.</td>
</tr>
</tbody>
</table>

**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**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SetGuid(const char *, GUID *) method</td>
<td>Sets a column to a GUID value.</td>
</tr>
<tr>
<td>SetGuid(ul_column_num, GUID *) method</td>
<td>Sets a column to a GUID value.</td>
</tr>
</tbody>
</table>

**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

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SetInt(const char *, ul_s_long) method</td>
<td>Sets a column to an integer value.</td>
</tr>
<tr>
<td>SetInt(ul_column_num, ul_s_long) method</td>
<td>Sets a column to an integer value.</td>
</tr>
</tbody>
</table>

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.

SetIntWith(Type method

Sets a column to an integer value of the specified integer type.
Overload list

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SetIntWithTyped(const char *, ul_s_big, ul_column_storage_type) method</td>
<td>Sets a column to an integer value of the specified integer type.</td>
</tr>
<tr>
<td>SetIntWithTyped(ul_column_num, ul_s_big, ul_column_storage_type) method</td>
<td>Sets a column to an integer value of the specified integer type.</td>
</tr>
</tbody>
</table>

SetIntWithTyped(const char *, ul_s_big, ul_column_storage_type) method

Sets a column to an integer value of the specified integer type.

Syntax

```cpp
public virtual bool SetIntWithTyped(
    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 86

### SetIntWithTypeId(ul_column_num, ul_s_big, ul_column_storage_type) method

Sets a column to an integer value of the specified integer type.

**Syntax**

```csharp
public virtual bool SetIntWithTypeId(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 86
**SetNull method**
Sets a column to null.

**Overload list**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SetNull(const char *) method</td>
<td>Sets a column to null.</td>
</tr>
<tr>
<td>SetNull(ul_column_num) method</td>
<td>Sets a column to null.</td>
</tr>
</tbody>
</table>

**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

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SetString(const char *, const char *, size_t) method</td>
<td>Sets a column to a string value.</td>
</tr>
<tr>
<td>SetString(ul_column_num, const char *, size_t) method</td>
<td>Sets a column to a string value.</td>
</tr>
</tbody>
</table>

SetString(const char *, const char *, size_t) method

Sets a column to a string value.

Syntax

```cpp
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 161

SetString(ul_column_num, const char *, size_t) method

Sets a column to a string value.

Syntax

```cpp
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 161

**Update method**

Updates the current row.

**Syntax**
```cpp
public virtual bool Update()
```

**Returns**
True on success, otherwise false.

**UpdateBegin method**

Selects the update mode for setting columns.

**Syntax**
```cpp
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**
```cpp
public class ULResultSetSchema
```

**Derived classes**

• “ULTableSchema class [UltraLite C++]” on page 206
Members

All members of the ULResultSetSchema class, including all inherited members.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GetColumnCount method</td>
<td>Gets the number of columns in the result set or table.</td>
</tr>
<tr>
<td>GetColumnID method</td>
<td>Gets the 1-based column ID from its name.</td>
</tr>
<tr>
<td>GetColumnName method</td>
<td>Gets the name of a column given its 1-based ID.</td>
</tr>
<tr>
<td>GetColumnPrecision method</td>
<td>Gets the precision of a numeric column.</td>
</tr>
<tr>
<td>GetColumnScale method</td>
<td>Gets the scale of a numeric column.</td>
</tr>
<tr>
<td>GetColumnSize method</td>
<td>Gets the size of the column.</td>
</tr>
<tr>
<td>GetColumnSQLType method</td>
<td>Gets the SQL type of a column.</td>
</tr>
<tr>
<td>GetColumnType method</td>
<td>Gets the storage/host variable type of a column.</td>
</tr>
<tr>
<td>GetConnection method</td>
<td>Gets the ULConnection object.</td>
</tr>
<tr>
<td>IsAliased method</td>
<td>Indicates whether the column in a result set was given an alias.</td>
</tr>
</tbody>
</table>

**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

```cpp
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 213

**GetColumnPrecision method**

Gets the precision of a numeric column.

Syntax

```cpp
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 84

**GetColumnType method**
Gets the storage/host variable type of a column.

Syntax
```csharp
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 86

**GetConnection method**
Gets the ULConnection object.

Syntax
```csharp
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
```csharp
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**
```java
public class ULTable : ULResultSet
```

**Base classes**
- “ULResultSet class [UltraLite C++]” on page 158

**Members**
All members of the ULTable class, including all inherited members.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AfterLast method</td>
<td>Moves the cursor after the last row.</td>
</tr>
<tr>
<td>AppendByteChunk method</td>
<td>Appends bytes to a column.</td>
</tr>
<tr>
<td>AppendStringChunk method</td>
<td>Appends a string chunk to a column.</td>
</tr>
<tr>
<td>BeforeFirst method</td>
<td>Moves the cursor before the first row.</td>
</tr>
<tr>
<td>Close method</td>
<td>Destroys this object.</td>
</tr>
<tr>
<td>Delete method</td>
<td>Deletes the current row and moves it to the next valid row.</td>
</tr>
<tr>
<td>DeleteAllRows method</td>
<td>Deletes all rows from a table.</td>
</tr>
<tr>
<td>DeleteNamed method</td>
<td>Deletes the current row and moves it to the next valid row.</td>
</tr>
<tr>
<td>Find method</td>
<td>Performs an exact match lookup based on the current index scanning forward</td>
</tr>
<tr>
<td></td>
<td>through the table.</td>
</tr>
<tr>
<td>FindBegin method</td>
<td>Prepares to perform a new Find call on a table by entering find mode.</td>
</tr>
<tr>
<td>FindFirst method</td>
<td>Performs an exact match lookup based on the current index scanning forward</td>
</tr>
<tr>
<td></td>
<td>through the table.</td>
</tr>
<tr>
<td>FindLast method</td>
<td>Performs an exact match lookup based on the current index scanning backward</td>
</tr>
<tr>
<td></td>
<td>through the table.</td>
</tr>
<tr>
<td>Name</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------------------</td>
<td>--------------------------------------------------</td>
</tr>
<tr>
<td>FindNext method</td>
<td>Gets the next row that exactly matches the index.</td>
</tr>
<tr>
<td>FindPrevious method</td>
<td>Gets the previous row that exactly matches the index.</td>
</tr>
<tr>
<td>First method</td>
<td>Moves the cursor to the first row.</td>
</tr>
<tr>
<td>GetBinary method</td>
<td>Fetches a value from a column as a ul_binary value.</td>
</tr>
<tr>
<td>GetBinaryLength method</td>
<td>Gets the binary length of the value of a column.</td>
</tr>
<tr>
<td>GetByteChunk method</td>
<td>Gets a binary chunk from the column.</td>
</tr>
<tr>
<td>GetConnection method</td>
<td>Gets the connection object.</td>
</tr>
<tr>
<td>GetDateTime method</td>
<td>Fetches a value from a column as a DECL_DATETIME.</td>
</tr>
<tr>
<td>GetDouble method</td>
<td>Fetches a value from a column as a double.</td>
</tr>
<tr>
<td>GetFloat method</td>
<td>Fetches a value from a column as a float.</td>
</tr>
<tr>
<td>GetGuid method</td>
<td>Fetches a value from a column as a GUID.</td>
</tr>
<tr>
<td>GetInt method</td>
<td>Fetches a value from a column as an integer.</td>
</tr>
<tr>
<td>GetIntWithType method</td>
<td>Fetches a value from a column as the specified integer type.</td>
</tr>
<tr>
<td>GetResultSetSchema method</td>
<td>Returns an object that can be used to get information about the result set.</td>
</tr>
<tr>
<td>GetRowCount method</td>
<td>Gets the number of rows in the table.</td>
</tr>
<tr>
<td>GetState method</td>
<td>Gets the internal state of the cursor.</td>
</tr>
<tr>
<td>GetString method</td>
<td>Fetches a value from a column as a null-terminated string.</td>
</tr>
<tr>
<td>GetStringChunk method</td>
<td>Gets a string chunk from the column.</td>
</tr>
<tr>
<td>GetStringLength method</td>
<td>Gets the string length of the value of a column.</td>
</tr>
<tr>
<td>GetTableSchema method</td>
<td>Returns a ULTableSchema object that can be used to get schema information about the table.</td>
</tr>
<tr>
<td>Insert method</td>
<td>Inserts a new row into the table.</td>
</tr>
<tr>
<td>InsertBegin method</td>
<td>Selects the insert mode for setting columns.</td>
</tr>
<tr>
<td>IsNull method</td>
<td>Checks if a column is NULL.</td>
</tr>
<tr>
<td>Name</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------------</td>
<td>------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Last method</td>
<td>Moves the cursor to the last row.</td>
</tr>
<tr>
<td>Lookup method</td>
<td>Performs a lookup based on the current index scanning forward through the table.</td>
</tr>
<tr>
<td>LookupBackward method</td>
<td>Performs a lookup based on the current index scanning backward through the table.</td>
</tr>
<tr>
<td>LookupBegin method</td>
<td>Prepares to perform a new lookup on a table.</td>
</tr>
<tr>
<td>LookupForward method</td>
<td>Performs a lookup based on the current index scanning forward through the table.</td>
</tr>
<tr>
<td>Next method</td>
<td>Moves the cursor forward one row.</td>
</tr>
<tr>
<td>Previous method</td>
<td>Moves the cursor back one row.</td>
</tr>
<tr>
<td>Relative method</td>
<td>Moves the cursor by offset rows from the current cursor position.</td>
</tr>
<tr>
<td>SetBinary method</td>
<td>Sets a column to a ul_binary value.</td>
</tr>
<tr>
<td>SetDateTime method</td>
<td>Sets a column to a DECL_DATETIME value.</td>
</tr>
<tr>
<td>SetDefault method</td>
<td>Sets a column to its default value.</td>
</tr>
<tr>
<td>SetDouble method</td>
<td>Sets a column to a double value.</td>
</tr>
<tr>
<td>SetFloat method</td>
<td>Sets a column to a float value.</td>
</tr>
<tr>
<td>SetGuid method</td>
<td>Sets a column to a GUID value.</td>
</tr>
<tr>
<td>SetInt method</td>
<td>Sets a column to an integer value.</td>
</tr>
<tr>
<td>SetIntWithType method</td>
<td>Sets a column to an integer value of the specified integer type.</td>
</tr>
<tr>
<td>SetNull method</td>
<td>Sets a column to null.</td>
</tr>
<tr>
<td>SetString method</td>
<td>Sets a column to a string value.</td>
</tr>
<tr>
<td>TruncateTable method</td>
<td>Truncates the table and temporarily activates STOP SYNCHRONIZATION DELETE.</td>
</tr>
<tr>
<td>Update method</td>
<td>Updates the current row.</td>
</tr>
<tr>
<td>UpdateBegin method</td>
<td>Selects the update mode for setting columns.</td>
</tr>
</tbody>
</table>
DeleteAllRows method

Deletes all rows from a table.

Syntax

```csharp
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

```csharp
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**

```csharp
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**

```csharp
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**

```csharp
public virtual bool TruncateTable()
```

**Returns**

True on success; otherwise, returns false.

**ULTableSchema class**

Represents the schema of an UltraLite table.

**Syntax**

```csharp
public class ULTableSchema : ULResultSetSchema
```

**Base classes**

- “ULResultSetSchema class [UltraLite C++]” on page 193

**Members**

All members of the ULTableSchema class, including all inherited members.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Close method</td>
<td>Destroys this object.</td>
</tr>
<tr>
<td>GetColumnCount method</td>
<td>Gets the number of columns in the result set or table.</td>
</tr>
<tr>
<td>GetColumnDefault method</td>
<td>Gets the default value for the column if it exists.</td>
</tr>
<tr>
<td>GetColumnDefaultType method</td>
<td>Gets the type of column default.</td>
</tr>
<tr>
<td>GetColumnID method</td>
<td>Gets the 1-based column ID from its name.</td>
</tr>
<tr>
<td>GetColumnName method</td>
<td>Gets the name of a column given its 1-based ID.</td>
</tr>
<tr>
<td>GetColumnPrecision method</td>
<td>Gets the precision of a numeric column.</td>
</tr>
<tr>
<td>GetColumnScale method</td>
<td>Gets the scale of a numeric column.</td>
</tr>
<tr>
<td>GetColumnSize method</td>
<td>Gets the size of the column.</td>
</tr>
<tr>
<td>GetColumnSQLType method</td>
<td>Gets the SQL type of a column.</td>
</tr>
<tr>
<td>GetColumnType method</td>
<td>Gets the storage/host variable type of a column.</td>
</tr>
</tbody>
</table>
### Close method

Destroys this object.

**Syntax**

```csharp
public virtual void Close()
```

### GetColumnDefault method

Gets the default value for the column if it exists.

**Syntax**

```csharp
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
```csharp
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 212

**GetGlobalAutoincPartitionSize method**
Gets the partition size.

Syntax
```csharp
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**

```csharp
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**

```csharp
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**

```csharp
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**

```csharp
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 216

**GetOptimalIndex method**

Determines the best index to use for searching for a column value.

Syntax

```csharp
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

```csharp
public virtual ULIndexSchema * GetPrimaryKey()
```

Returns

a ULIndexSchema object for the table's primary key.

**GetPublicationPredicate method**

Gets the publication predicate as a string.
Syntax

```cpp
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

```cpp
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 215

**InPublication method**

Checks whether the table is contained in the named publication.

Syntax

```cpp
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

```csharp
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

```csharp
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

```csharp
public enum ul_column_default_type
```

Members

<table>
<thead>
<tr>
<th>Member name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ul_column_default_none</td>
<td>The column has no default value.</td>
</tr>
<tr>
<td>ul_column_default_autoincrement</td>
<td>The column default is AUTOINCREMENT.</td>
</tr>
</tbody>
</table>
### 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 208](#)
## 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

## 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 195

## ul_index_flag enumeration

Flags (bit fields) which identify properties of an index.

### Syntax

```java
public enum ul_index_flag
```

### Members

<table>
<thead>
<tr>
<th>Member name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ul_index_flag_primary_key</td>
<td>The index is a primary key.</td>
</tr>
<tr>
<td>ul_index_flag_unique_key</td>
<td>The index is a primary key or index created for a unique constraint (nulls not allowed).</td>
</tr>
<tr>
<td>Member name</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>ul_index_flag_unique_index</td>
<td>The index was created with the UNIQUE flag (or is a primary key).</td>
</tr>
<tr>
<td>ul_index_flag_foreign_key</td>
<td>The index is a foreign key.</td>
</tr>
<tr>
<td>ul_index_flag_foreign_key_nullable</td>
<td>The foreign key allows nulls.</td>
</tr>
<tr>
<td>ul_index_flag_foreign_key_check_on_commit</td>
<td>Referential integrity checks are performed on commit (rather than on insert/update).</td>
</tr>
</tbody>
</table>

**See also**

- “ULIndexSchema.GetIndexFlags method [UltraLite C++]” on page 147

### ul_table_sync_type enumeration

Identifies a table synchronization type.

**Syntax**

```c
public enum ul_table_sync_type
```

**Members**

<table>
<thead>
<tr>
<th>Member name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ul_table_sync_on</td>
<td>All changed rows are synchronized, which is the default behavior. This initializer corresponds to the SYNCHRONIZE ON clause in a CREATE TABLE statement.</td>
</tr>
<tr>
<td>ul_table_sync_off</td>
<td>Table is never synchronized. This initializer corresponds to the SYNCHRONIZE OFF clause in a CREATE TABLE statement.</td>
</tr>
<tr>
<td>ul_table_sync_upload_all_rows</td>
<td>Always upload every row, including unchanged rows. This initializer corresponds to the SYNCHRONIZE ALL clause in a CREATE TABLE statement.</td>
</tr>
<tr>
<td>ul_table_sync_download_only</td>
<td>Changes are never uploaded. This initializer corresponds to the SYNCHRONIZE DOWNLOAD clause in a CREATE TABLE statement.</td>
</tr>
</tbody>
</table>
See also

- “ULTableSchema.GetTableSyncType method [UltraLite C++]” on page 211

**UL_BLOB_CONTINUE variable**

Used when reading data with the ULResultSet.GetStringChunk or ULResultSet.GetBytesStart methods.

**Syntax**

```c
#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 177
- “ULResultSet.GetBytesChunk method [UltraLite C++]” on page 166

**ul_index_iter_start variable**

Used by the GetNextIndex method to initialize index iteration in a table.

**Syntax**

```c
#define ul_index_iter_start
```

See also

- “ULTableSchema.GetNextIndex method [UltraLite C++]” on page 209

**ul_publication_iter_start variable**

Used by the GetNextPublication method to initialize publication iteration in a database.

**Syntax**

```c
#define ul_publication_iter_start
```

See also

- “ULDatabaseSchema.GetNextPublication method [UltraLite C++]” on page 139

**ul_table_iter_start variable**

Used by the GetNextTable method to initialize table iteration in a database.

**Syntax**

```c
#define ul_table_iter_start
```
See also

- “ULDatabaseSchema.GetNextTable method [UltraLite C++]” on page 139

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 32.

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 217

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 217

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 255

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 255

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 255

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 255

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 255

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

Syntax
```c
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 255

Example
The following example illustrates how to use the MLFileDownload method:

```c
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 255

Example

The following example illustrates how to use the MLFileUpload method:

```csharp
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.
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 255

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

```java
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

```java
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**

```java
public ul_bool ULCreateDatabase(  
    SQLCA * sqlca,  
    char const * connectparms,  
    char const * createparms,  
    void * reserved
)
```

**Parameters**

- **sqlca**  A pointer to the initialized SQLCA.

- **connectparms**  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:

```c
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**

```c
public ul_bool ULCreateNotificationQueue(
   SQLCA * sqlca,
   char const * name,
   char const * parameters
)
```

**Parameters**

● **sqlca**  A pointer to the SQLCA.
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 248

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

```java
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.
**ULEnableAesDBEncryption method**

Enables AES database encryption.

**Syntax**

```java
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**

```java
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.

FIPS-certified encryption requires a separate license. All strong encryption technologies are subject to export regulations.

See “Separately licensed components” [SQL Anywhere 16 - Introduction].
See also

- “ULEnableAesDBEncryption method [UltraLite Embedded SQL]” on page 228

**ULEnableHttpSynchronization method**

Enables HTTP synchronization.

**Syntax**

```java
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**

```java
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**

```java
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.

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 230

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.

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

- “ULEnableRsaFipsSyncEncryption method [UltraLite Embedded SQL]” on page 229

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**
```java
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**
```java
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**

```c
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**

```c
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**

```c
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

```java
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

```java
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**

```java
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.

**ULGetErrorMessage method**

Retrieve error parameter via an ordinal parameter number.

**Syntax**

```java
public size_t ULGetErrorMessage(
    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 235

ULGetErrorParameterCount method
Obtains a count of the number of error parameters.

Syntax
public ul_u_long ULGetErrorParameterCount(SQCA 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 234

ULGetIdentity method
Gets the @identity value.

Syntax
public ul_u_big ULGetIdentity(SQCA * 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(
    SQCA * sqlca,
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:

```c
DECL_DATETIME dt;
ret = ULGetLastDownloadTime( &sqlca, UL_TEXT("UL_PUB_PUB1"), &dt );
```

ULGetNotification method

Reads an event notification.

Syntax

```c
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 222
- “ULGetNotificationParameter method [UltraLite Embedded SQL]” on page 237

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 94

**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 243

**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,
        API reference
    )

238 Copyright © 2014, SAP AG or an SAP affiliate company. - SAP Sybase SQL Anywhere 16.0
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 242

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:

```c
LRESULT CALLBACK WindowProc ( HWND hwnd,
    UINT uMsg,
```
```c
  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 246

### ULLibraryVersion method
Returns the version number of the UltraLite runtime library.

**Syntax**

```c
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**

```c
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**

```c
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:

```c
ULResetLastDownloadTime( &sqlca, UL_TEXT("*"));
```

**ULRevokeConnectFrom method**

Revokes access from an UltraLite database for a user ID.

Syntax

```c
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

```c
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

```java
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 237

**ULSetDatabaseID method**

Sets the database identification number.

Syntax

```java
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 238
ULSetDatabaseOptionString method

Sets a database option from a string value.

Syntax

```java
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

```java
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

```java
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 21

ULSetSyncInfo method

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

Syntax

```java
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

```java
public ul_ret_void ULSetSynchronizationCallback(  
    SQLCA * sqlca,  
    ul_sync_observer_fn callback,  
)
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**

```java
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**

```java
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**

```java
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**

```java
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**

```java
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**

```java
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:

```java
ul_sync_info info;
ULInitSyncInfo( &info );
info.user_name = UL_TEXT( "user_name" );
```
ULSynchronizeFromProfile method
Synchronizes the database using the given profile and merge parameters.

Syntax
```java
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
```java
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 237

### ULTruncateTable method
Truncates the table and temporarily activates the STOP SYNCHRONIZATION DELETE statement.

**Syntax**
```java
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**
```java
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.
ULValidateDatabaseTableName method

Validates the database on this connection.

Syntax

```c
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
)
```

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 100
- “ULVF_INDEX variable [UltraLite C and Embedded SQL datatypes]” on page 99
- “ULVF_DATABASE variable [UltraLite C and Embedded SQL datatypes]” on page 98
- “ULVF_EXPRESS variable [UltraLite C and Embedded SQL datatypes]” on page 98
- “ULVF_FULL_VALIDATE variable [UltraLite C and Embedded SQL datatypes]” on page 98
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 100
- “ULVF_INDEX variable [UltraLite C and Embedded SQL datatypes]” on page 99
- “ULVF_DATABASE variable [UltraLite C and Embedded SQL datatypes]” on page 98
- “ULVF_EXPRESS variable [UltraLite C and Embedded SQL datatypes]” on page 98
- “ULVF_FULL_VALIDATE variable [UltraLite C and Embedded SQL datatypes]” on page 98

**ul_database_option_id enumeration**

Specifies possible database options that users can set.

Syntax

```java
public enum ul_database_option_id
```
Members

<table>
<thead>
<tr>
<th>Member name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ul_option_global_database_id</code></td>
<td>The global database ID is set using an unsigned long integer.</td>
</tr>
<tr>
<td><code>ul_option_ml_remote_id</code></td>
<td>The remote ID is set using a string.</td>
</tr>
<tr>
<td><code>ul_option_commit_flush_timeout</code></td>
<td>The database commit flush timeout is set as an integer, representing a time threshold measured in milliseconds.</td>
</tr>
<tr>
<td><code>ul_option_commit_flush_count</code></td>
<td>The database commit flush count is set as integer, representing a commit count threshold.</td>
</tr>
<tr>
<td><code>ul_option_isolation_level</code></td>
<td>The connection isolation level is set as string.</td>
</tr>
<tr>
<td></td>
<td>(read_committed/read_uncommitted)</td>
</tr>
<tr>
<td><code>ul_option_cache_allocation</code></td>
<td>Set to resize the database file cache.</td>
</tr>
<tr>
<td></td>
<td>The value is an integer in the range 0 to 100, representing the amount of cache allocated of the minimum to maximum size range.</td>
</tr>
</tbody>
</table>

Remarks

These database options are used with the ULConnection.SetDatabaseOption method.

See also

- “ULConnection.SetDatabaseOption method [UltraLite C++]” on page 122

ul_database_property_id enumeration

Specifies possible database properties that users can retrieve.

Syntax

```java
public enum ul_database_property_id
```

Members

<table>
<thead>
<tr>
<th>Member name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ul_property_date_format</code></td>
<td>Date format.</td>
</tr>
<tr>
<td></td>
<td>(date_format)</td>
</tr>
<tr>
<td><code>ul_property_date_order</code></td>
<td>Date order.</td>
</tr>
<tr>
<td></td>
<td>(date_order)</td>
</tr>
<tr>
<td>Member name</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------------------------------</td>
<td>------------------------------</td>
</tr>
<tr>
<td>ul_property_nearest_century</td>
<td>Nearest century.</td>
</tr>
<tr>
<td></td>
<td>(nearest_century)</td>
</tr>
<tr>
<td>ul_property_precision</td>
<td>Precision.</td>
</tr>
<tr>
<td></td>
<td>(precision)</td>
</tr>
<tr>
<td>ul_property_scale</td>
<td>Scale.</td>
</tr>
<tr>
<td></td>
<td>(scale)</td>
</tr>
<tr>
<td>ul_property_time_format</td>
<td>Time format.</td>
</tr>
<tr>
<td></td>
<td>(time_format)</td>
</tr>
<tr>
<td>ul_property_timestamp_format</td>
<td>Timestamp format.</td>
</tr>
<tr>
<td></td>
<td>(timestamp_format)</td>
</tr>
<tr>
<td>ul_property_timestamp_increment</td>
<td>Timestamp increment.</td>
</tr>
<tr>
<td></td>
<td>(timestamp_increment)</td>
</tr>
<tr>
<td>ul_property_name</td>
<td>Name.</td>
</tr>
<tr>
<td></td>
<td>(Name)</td>
</tr>
<tr>
<td>ul_property_file</td>
<td>File.</td>
</tr>
<tr>
<td></td>
<td>(File)</td>
</tr>
<tr>
<td>ul_property_encryption</td>
<td>Encryption.</td>
</tr>
<tr>
<td></td>
<td>(Encryption)</td>
</tr>
<tr>
<td>ul_property_global_database_id</td>
<td>Global database ID.</td>
</tr>
<tr>
<td></td>
<td>(global_database_id)</td>
</tr>
<tr>
<td>ul_property_ml_remote_id</td>
<td>Remote ID.</td>
</tr>
<tr>
<td></td>
<td>(ml_remote_id)</td>
</tr>
<tr>
<td>ul_property_char_set</td>
<td>Character set.</td>
</tr>
<tr>
<td></td>
<td>(CharSet)</td>
</tr>
<tr>
<td>Member name</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>ul_property_collation</td>
<td>collation sequence.</td>
</tr>
<tr>
<td></td>
<td>(Collation)</td>
</tr>
<tr>
<td>ul_property_page_size</td>
<td>Page size.</td>
</tr>
<tr>
<td></td>
<td>(PageSize)</td>
</tr>
<tr>
<td>ul_property_case_sensitive</td>
<td>CaseSensitive.</td>
</tr>
<tr>
<td></td>
<td>(CaseSensitive)</td>
</tr>
<tr>
<td>ul_property_conn_count</td>
<td>Connection count.</td>
</tr>
<tr>
<td></td>
<td>(ConnCount)</td>
</tr>
<tr>
<td>ul_property_max_hash_size</td>
<td>Default maximum index hash.</td>
</tr>
<tr>
<td></td>
<td>(MaxHashSize)</td>
</tr>
<tr>
<td>ul_property_checksum_level</td>
<td>Database checksum level.</td>
</tr>
<tr>
<td></td>
<td>(ChecksumLevel)</td>
</tr>
<tr>
<td>ul_property_checkpoint_count</td>
<td>Database checkpoint count.</td>
</tr>
<tr>
<td></td>
<td>(CheckpointCount)</td>
</tr>
<tr>
<td>ul_property_commit_flush_timeout</td>
<td>Database commit flush timeout.</td>
</tr>
<tr>
<td></td>
<td>(commit_flush_timeout)</td>
</tr>
<tr>
<td>ul_property_commit_flush_count</td>
<td>Database commit flush count.</td>
</tr>
<tr>
<td></td>
<td>(commit_flush_count)</td>
</tr>
<tr>
<td>ul_property_isolation_level</td>
<td>Connection isolation level.</td>
</tr>
<tr>
<td></td>
<td>(isolation_level)</td>
</tr>
<tr>
<td>ul_property_time-stamp_with_time_zone_format</td>
<td>Timestamp with time zone format.</td>
</tr>
<tr>
<td></td>
<td>(timestamp_with_time_zone_format)</td>
</tr>
<tr>
<td>ul_property_cache_allocation</td>
<td>The current database file cache size, as a percentage value of the minimum to maximum range.</td>
</tr>
</tbody>
</table>
Remarks
These properties are used with the ULConnection.GetDatabaseProperty method.

See also
- “ULConnection.GetDatabaseProperty method [UltraLite C++]” on page 113

ml_file_transfer_info structure
A structure containing the parameters to the file upload/download.

Syntax
public typedef struct ml_file_transfer_info

Members

<table>
<thead>
<tr>
<th>Member name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>auth_parms</td>
<td>const char *</td>
<td>Supplies parameters to authentication parameters in MobiLink events.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>For more information, see “Additional Parameters synchronization parameter” [UltraLite - Database Management and Reference].</td>
</tr>
<tr>
<td>auth_status</td>
<td>asa_uint16</td>
<td>Supplies parameters to authentication parameters in MobiLink events.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>For more information, see “Additional Parameters synchronization parameter” [UltraLite - Database Management and Reference].</td>
</tr>
<tr>
<td>auth_value</td>
<td>asa_uint32</td>
<td>Reports results of a custom MobiLink user authentication script.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The MobiLink server provides this information to the client.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>For more information, see “Authentication Value synchronization parameter” [UltraLite - Database Management and Reference].</td>
</tr>
<tr>
<td>enable_resume</td>
<td>bool</td>
<td>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.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>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.</td>
</tr>
<tr>
<td>Member name</td>
<td>Type</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------</td>
<td>--------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>error</td>
<td>mlf_stream_error</td>
<td>Contains information about any error that occurs.</td>
</tr>
<tr>
<td>file_auth_code</td>
<td>asa_uint16</td>
<td>Contains the return code of the optional authenticate_file_transfer script on the server.</td>
</tr>
<tr>
<td>filename</td>
<td>const char *</td>
<td>The file name to be transferred from the server running MobiLink.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MobiLink searches the username subdirectory first, before defaulting to the root directory.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>For more information, see “-ftr mlsrv16 option” [MobiLink - Server Administration].</td>
</tr>
<tr>
<td>local_filename</td>
<td>const char *</td>
<td>The local name for the downloaded file.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>If this parameter is empty, the value in file name is used.</td>
</tr>
<tr>
<td>local_path</td>
<td>const char *</td>
<td>The local path to store the downloaded file.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>If this parameter is empty (the default), the downloaded file is stored in the current directory.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>On Windows Mobile, if dest_path is empty, the file is stored in the root () directory of the device.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>On the desktop, if the dest_path is empty, the file is stored in the user's current directory.</td>
</tr>
<tr>
<td>num_auth_parms</td>
<td>asa_uint8</td>
<td>The number of authentication parameters being passed to authentication parameters in MobiLink events.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>For more information, see “Number of Authentication Parameters parameter” [UltraLite - Database Management and Reference].</td>
</tr>
<tr>
<td>observer</td>
<td>ml_file_transfer_observer_fn</td>
<td>A callback can be provided to observe file download progress through the 'observer' field.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>For more details, see description of the callback function that follows.</td>
</tr>
<tr>
<td>password</td>
<td>const char *</td>
<td>The password for the MobiLink user name.</td>
</tr>
<tr>
<td>remote_key</td>
<td>const char *</td>
<td>The MobiLink remote key.</td>
</tr>
<tr>
<td>Member name</td>
<td>Type</td>
<td>Description</td>
</tr>
<tr>
<td>--------------</td>
<td>----------</td>
<td>----------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>stream</td>
<td>const char *</td>
<td>The protocol can be one of: TCPIP, TLS, HTTP, or HTTPS. This field is required.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>For more information, see “Stream Type synchronization parameter” [UltraLite - Database Management and Reference].</td>
</tr>
<tr>
<td>stream_parms</td>
<td>const char *</td>
<td>The protocol options for a given stream. For more information, see “UltraLite network protocol options” [UltraLite - Database Management and Reference].</td>
</tr>
<tr>
<td>transferred_file</td>
<td>asa_uint16</td>
<td>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.</td>
</tr>
<tr>
<td>user_data</td>
<td>void *</td>
<td>The application-specific information made available to the synchronization observer. For more information, see “User Data synchronization parameter” [UltraLite - Database Management and Reference].</td>
</tr>
<tr>
<td>username</td>
<td>const char *</td>
<td>The MobiLink user name. This field is required.</td>
</tr>
<tr>
<td>version</td>
<td>const char *</td>
<td>The MobiLink script version. This field is required.</td>
</tr>
</tbody>
</table>

**ml_file_transfer_status structure**

A structure containing status/progress information while the file upload/download is in progress.

**Syntax**

```plaintext
public typedef struct ml_file_transfer_status
```

**Members**

<table>
<thead>
<tr>
<th>Member name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>bytes_transferred</td>
<td>asa_uint64</td>
<td>Indicates how much of the file has been downloaded so far, including previous synchronizations, if the download is resumed.</td>
</tr>
</tbody>
</table>
### 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

<table>
<thead>
<tr>
<th>Member name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>error_string</td>
<td>char</td>
<td>The localized description for the system_error_code value, if available from the system, or additional information for the stream_error_code value.</td>
</tr>
<tr>
<td>stream_error_code</td>
<td>ss_error_code</td>
<td>The specific stream error. For a list of possible values, see the ss_error_code enumeration in the %SQLANY16%\SDK\Include\sserror.h header file.</td>
</tr>
<tr>
<td>system_error_code</td>
<td>asa_int32</td>
<td>A system-specific error code.</td>
</tr>
</tbody>
</table>
mlft_stream_error_w structure

A structure containing status/progress information while the file upload or download is in progress.

Syntax

```c
public typedef struct mlft_stream_error_w
```

Members

<table>
<thead>
<tr>
<th>Member name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>error_string</td>
<td>wchar_t</td>
<td>The localized description for the system_error_code value, if available from the system, or additional information for the stream_error_code value.</td>
</tr>
<tr>
<td>stream_error_code</td>
<td>ss_error_code</td>
<td>The specific stream error. For a list of possible values, see the ss_error_code enumeration in the <code>%SQLANY12%\SDK\Include\sserror.h</code> header file.</td>
</tr>
<tr>
<td>system_error_code</td>
<td>asa_int32</td>
<td>A system-specific error code.</td>
</tr>
</tbody>
</table>

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 258
- “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

```c
#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, 83

### 16-bit signed integer UltraLite embedded SQL data type
- about, 39

### 32-bit signed integer UltraLite embedded SQL data type
- about, 39

### 4-byte floating-point UltraLite embedded SQL data type
- about, 39

### 8-byte floating-point UltraLite embedded SQL data type
- about, 39

## A

### accessing data
- UltraLite C++ API, 14

### accessing schema information
- UltraLite C++ about, 20

### ActiveSync
- class names, 64
- UltraLite MFC requirements, 67
- UltraLite synchronization for Windows Mobile, 66
- UltraLite versions for Windows Mobile, 66
- UltraLite Windows Mobile applications, 66
- WindowProc function, 66

### AES encryption algorithm
- UltraLite embedded SQL databases, 52

### AfterLast method
- ULResultSet class [UltraLite C++ API], 160

### Android
- disconnecting from a database, 22

### AppendByteChunk method
- ULResultSet class [UltraLite C++ API], 160

### AppendParameterByteChunk method
- ULPStatement class [UltraLite C++ API], 150

### AppendParameterStringChunk method
- ULPStatement class [UltraLite C++ API], 150

### AppendStringChunk method
- ULResultSet class [UltraLite C++ API], 161

### applications
- building UltraLite embedded SQL, 60
- compiling UltraLite embedded SQL, 60
- developing for iOS, 3
- preprocessing UltraLite embedded SQL, 60
- writing UltraLite embedded SQL, 32

### architectures
- UltraLite C/C++, 102
- AutoCommit mode
- UltraLite C++ development, 20

## B

### BeforeFirst method
- ULResultSet class [UltraLite C++ API], 163

### binary UltraLite embedded SQL data type
- about, 40

### BlackBerry
- disconnecting from a database, 22

### build processes
- embedded SQL applications, 60
- UltraLite embedded SQL applications, 60

### building
- UltraLite embedded SQL applications, 60

### C

### CancelGetNotification method
- ULResultSet class [UltraLite C++ API], 163

### canceling monitoring
- in UltraLite embedded SQL, 56

### canceling synchronization
- in UltraLite embedded SQL, 56

### casting
- UltraLite C++ API data types in, 19

### ChangeEncryptionKey method
- ULConnection class [UltraLite C++ API], 105

### character string UltraLite embedded SQL data type
- fixed length, 40
- variable length, 40

### Checkpoint method
- ULConnection class [UltraLite C++ API], 106

### class names
- ActiveSync synchronization, 64

### Clear method
- ULError class [UltraLite C++ API], 141

### Close method
DeclareEvent method
ULConnection class [UltraLite C++ API], 108

declaring
UltraLite host variables, 38
Delete method
ULResultSet class [UltraLite C++ API], 163
DeleteAllRows method
ULTable class [UltraLite C++ API], 201
DeleteNamed method
ULResultSet class [UltraLite C++ API], 164
deleting
UltraLite C++ API table rows, 19
dependencies
UltraLite embedded SQL, 61
deploying
in-process version of UltraLite, 23
UltraLite for Windows Mobile, 23, 25, 27
DestroyNotificationQueue method
ULConnection class [UltraLite C++ API], 109
development
UltraLite C++, 3
development platforms
UltraLite C++, 1
development process
UltraLite embedded SQL, 32
development tools
UltraLite embedded SQL, 61
directives
UltraLite applications, 83
DML
UltraLite C++, 7
DropDatabase method
ULDatabaseManager class [UltraLite C++ API], 130
DT_LONGBINARY UltraLite embedded SQL data
type
about, 41
DT_LONGVARCHAR UltraLite embedded SQL data
type
about, 41
dynamic libraries
UltraLite C++ applications, 23, 25

E
E2EE private key
iPhone, 4
Mac OS X, 4
e2ee_public_key
iPhone, 4
Mac OS X, 4
emulator
UltraLite for Windows Mobile, 23
EnableAesDBEncryption method
ULDatabaseManager class [UltraLite C++ API], 130
EnableAesFipsDBEncryption method
ULDatabaseManager class [UltraLite C++ API], 131
EnableAllSynchronization method
ULDatabaseManager class [UltraLite C++ API], 131
EnableHttpsSynchronization method
ULDatabaseManager class [UltraLite C++ API], 131
EnableHttpSynchronization method
ULDatabaseManager class [UltraLite C++ API], 132
EnableRsaE2ee method
ULDatabaseManager class [UltraLite C++ API], 132
EnableRsaFipsE2ee method
ULDatabaseManager class [UltraLite C++ API], 132
EnableRsaFipsSyncEncryption method
ULDatabaseManager class [UltraLite C++ API], 133
EnableRsaSyncEncryption method
ULDatabaseManager class [UltraLite C++ API], 133
EnableTcpipSynchronization method
ULDatabaseManager class [UltraLite C++ API], 133
EnableTlsSynchronization method
ULDatabaseManager class [UltraLite C++ API], 134
EnableZlibSyncCompression method
ULDatabaseManager class [UltraLite C++ API], 134
encryption
changing keys in UltraLite embedded SQL, 53
iPhone, 4
Mac OS X, 4
PEM encoded X509 certificate, 4
UltraLite databases using embedded SQL, 52
UltraLite embedded SQL databases, 52
Index

end-to-end encryption
  iPhone, 4
  Mac OS X, 4
environment variables
  INCLUDE, 71
error handling
  UltraLite C++, 21
errors
  UltraLite C++ API handling, 21
  UltraLite codes, 36
  UltraLite embedded SQL communications errors, 55
  UltraLite SQLCODE, 36
  UltraLite sqlcode SQLCA field, 36
EXEC SQL
  UltraLite embedded SQL development, 35
ExecuteQuery method
  ULPreparedStatement class [UltraLite C++ API], 151
ExecuteScalar method
  ULConnection class [UltraLite C++ API], 109
ExecuteScalarV method
  ULConnection class [UltraLite C++ API], 111
ExecuteStatement method
  ULConnection class [UltraLite C++ API], 112
  ULPreparedStatement class [UltraLite C++ API], 151

F
FETCH statement
  UltraLite embedded SQL multi-row queries, 49
  UltraLite embedded SQL single-row queries, 48
fetching
  UltraLite embedded SQL, 48
Find method
  ULTable class [UltraLite C++ API], 201
find methods
  UltraLite C++, 17
find mode
  UltraLite C++, 15
FindBegin method
  ULTable class [UltraLite C++ API], 201
FindFirst method
  ULTable class [UltraLite C++ API], 202
FindLast method
  ULTable class [UltraLite C++ API], 202
FindNext method
  UltraLite class [UltraLite C++ API], 203
FindPrevious method
  ULTable class [UltraLite C++ API], 203
Fini method
  ULDatabaseManager class [UltraLite C++ API], 135
First method
  ULResultSet class [UltraLite C++ API], 164
functions
  UltraLite embedded SQL, 217

G
GetBinary method
  ULResultSet class [UltraLite C++ API], 164
GetBinaryLength method
  ULResultSet class [UltraLite C++ API], 165
GetByteChunk method
  ULResultSet class [UltraLite C++ API], 166
GetChildObjectCount method
  ULConnection class [UltraLite C++ API], 112
GetColumnCount method
  ULIndexSchema class [UltraLite C++ API], 146
  ULResultSetSchema class [UltraLite C++ API], 194
GetColumnDefault method
  ULTableSchema class [UltraLite C++ API], 207
GetColumnDefaultType method
  ULTableSchema class [UltraLite C++ API], 208
GetColumnID method
  ULResultSetSchema class [UltraLite C++ API], 194
GetColumnName method
  ULIndexSchema class [UltraLite C++ API], 146
  ULResultSetSchema class [UltraLite C++ API], 195
GetColumnPrecision method
  ULResultSetSchema class [UltraLite C++ API], 195
GetColumnScale method
  ULResultSetSchema class [UltraLite C++ API], 196
GetColumnSize method
  ULResultSetSchema class [UltraLite C++ API], 196
GetColumnSQLType method
  ULResultSetSchema class [UltraLite C++ API], 196
GetColumnType method
ULResultSetSchema class [UltraLite C++ API], 197

GetConnection method
ULDatabaseSchema class [UltraLite C++ API], 138
ULIndexSchema class [UltraLite C++ API], 146
ULPreparedStatement class [UltraLite C++ API], 151
ULResultSet class [UltraLite C++ API], 168
ULResultSetSchema class [UltraLite C++ API], 197

GetDatabaseProperty method
ULConnection class [UltraLite C++ API], 113
GetDatabasePropertyInt method
ULConnection class [UltraLite C++ API], 113

GetDatabaseSchema method
ULConnection class [UltraLite C++ API], 114

GetDateTime method
ULResultSet class [UltraLite C++ API], 168
GetDouble method
ULResultSet class [UltraLite C++ API], 169

GetErrorInfo method
ULError class [UltraLite C++ API], 141

GetFloat method
ULResultSet class [UltraLite C++ API], 170

GetGlobalAutoincPartitionSize method
ULTableSchema class [UltraLite C++ API], 208

GetGuid method
ULResultSet class [UltraLite C++ API], 171

GetIndexColumnID method
ULIndexSchema class [UltraLite C++ API], 146

GetIndexCount method
ULTableSchema class [UltraLite C++ API], 209

GetIndexFlags method
ULIndexSchema class [UltraLite C++ API], 147

GetIndexSchema method
ULTableSchema class [UltraLite C++ API], 209

GetInt method
ULResultSet class [UltraLite C++ API], 172

GetIntWithType method
ULResultSet class [UltraLite C++ API], 172

GetLastDownloadTime method
ULConnection class [UltraLite C++ API], 114

GetLastError method
ULConnection class [UltraLite C++ API], 115

GetLastIdentity method
ULConnection class [UltraLite C++ API], 115

GetName method
ULIndexSchema class [UltraLite C++ API], 147
ULTableSchema class [UltraLite C++ API], 209

GetNextIndex method
ULTableSchema class [UltraLite C++ API], 209

GetNextPublication method
ULDatabaseSchema class [UltraLite C++ API], 139

GetNextTable method
ULDatabaseSchema class [UltraLite C++ API], 139

GetNotification method
ULConnection class [UltraLite C++ API], 115

GetNotificationParameter method
ULConnection class [UltraLite C++ API], 116

GetOptimalIndex method
ULTableSchema class [UltraLite C++ API], 210

GetParameter method
ULError class [UltraLite C++ API], 142

GetParameterCount method
ULError class [UltraLite C++ API], 143
ULTableSchema class [UltraLite C++ API], 143

GetParameterID method
ULTableSchema class [UltraLite C++ API], 152

GetParameterType method
ULTableSchema class [UltraLite C++ API], 152

GetPlan method
ULTableSchema class [UltraLite C++ API], 152

GetPrimaryKey method
ULTableSchema class [UltraLite C++ API], 210

GetPublicationCount method
ULDatabaseSchema class [UltraLite C++ API], 139

GetPublicationPredicate method
ULTableSchema class [UltraLite C++ API], 148

GetResultSetSchema method
ULTableSchema class [UltraLite C++ API], 153

GetReferencedIndexName method
ULTableSchema class [UltraLite C++ API], 147

GetReferencedTableName method
ULTableSchema class [UltraLite C++ API], 147

GetRowCount method
ULTableSchema class [UltraLite C++ API], 174
ULResultSet class [UltraLite C++ API], 175
GetRowsAffectedCount method
ULPreparedStatement class [UltraLite C++ API], 153
GetSQLca method
ULConnection class [UltraLite C++ API], 117
GetSQLCode method
ULError class [UltraLite C++ API], 143
GetSQLCount method
ULError class [UltraLite C++ API], 143
GetState method
ULResultSet class [UltraLite C++ API], 175
GetString method
ULError class [UltraLite C++ API], 144
ULResultSet class [UltraLite C++ API], 175
GetStringChunk method
ULResultSet class [UltraLite C++ API], 177
GetStringLength method
ULResultSet class [UltraLite C++ API], 178
GetSyncResult method
ULConnection class [UltraLite C++ API], 117
GetTableCount method
ULDatabaseSchema class [UltraLite C++ API], 140
GetTableName method
ULIndexSchema class [UltraLite C++ API], 148
GetTableSchema method
ULDatabaseSchema class [UltraLite C++ API], 140
ULTable class [UltraLite C++ API], 203
GetTableSyncType method
ULTableSchema class [UltraLite C++ API], 211
GetURL method
ULError class [UltraLite C++ API], 144
getUserPointer method
ULConnection class [UltraLite C++ API], 117
GlobalAutoincUsage method
ULConnection class [UltraLite C++ API], 117
GrantConnectTo method
ULConnection class [UltraLite C++ API], 118

H
HasResultSet method
ULPreparedStatement class [UltraLite C++ API], 154
host variables
UltraLite embedded SQL, 38

 UltraLite embedded SQL expressions, 43
UltraLite scope, 42
UltraLite usage, 42

import libraries
  UltraLite C++, 23, 25, 27
INCLUDE statement
  UltraLite SQLCA, 35
indexes
  UltraLite C++ API schema information in, 21
indicator variables
  UltraLite embedded SQL, 46
  UltraLite NULL, 47
Init method
  ULDatabaseManager class [UltraLite C++ API], 135
InitSyncInfo method
  ULConnection class [UltraLite C++ API], 118
InPublication method
  ULTableSchema class [UltraLite C++ API], 211
Insert method
  ULTable class [UltraLite C++ API], 203
insert mode
  UltraLite C++, 15
InsertBegin method
  ULTable class [UltraLite C++ API], 204
inserting
  UltraLite C++ API table rows, 16
installing
  UltraLite Windows Mobile applications, 62
iOS
about UltraLite C++ applications, 3
creating and connecting to a database, 5
database schemas, 20
developing applications, 3
handling errors, 21
UltraLite C++ development, 23
iPhone
build settings for UltraLite C++, 3
include files for UltraLite C++, 3
UltraLite runtime libraries for, 29
IsAliased method
  ULResultSetSchema class [UltraLite C++ API], 197
IsColumnDescending method
  ULIndexSchema class [UltraLite C++ API], 148
IsColumnInIndex method
ULTableSchema class [UltraLite C++ API], 212
IsColumnNullable method
ULTableSchema class [UltraLite C++ API], 212
IsNull method
ULResultSet class [UltraLite C++ API], 180
IsOK method
ULError class [UltraLite C++ API], 145

L
Last method
ULResultSet class [UltraLite C++ API], 181
libeay32.dll
UltraLite C++ development, 23, 25, 27
libraries
UltraLite applications for iOS, 23, 29
UltraLite applications for Linux, 23, 31
UltraLite applications for Mac OS X, 23, 29
UltraLite applications for Windows Mobile, 23, 25, 27
UltraLite compiling and linking in C++, 23, 25, 27
UltraLite DLL for Windows Mobile, 23, 27
UltraLite linking example in C++, 71
library functions
UltraLite embedded SQL, 217
libulbase.a
UltraLite C++ development, 23, 29, 31
libulsrs.a
UltraLite C++ development, 23, 31
libulrt.a
UltraLite C++ development, 23, 29, 31
libulrt.lib
UltraLite C++ development, 23
linking
UltraLite applications for Windows Mobile, 23, 25, 27
UltraLite C++ applications, 23, 25, 27
Linux
UltraLite runtime libraries for, 23
Lookup method
ULTable class [UltraLite C++ API], 204
lookup methods
UltraLite C++, 17
lookup mode
UltraLite C++, 15
LookupBackward method
ULTable class [UltraLite C++ API], 204
LookupBegin method
ULTable class [UltraLite C++ API], 205
LookupForward method
ULTable class [UltraLite C++ API], 205

M
Mac OS X
about UltraLite C++ applications, 3
include files for UltraLite C++, 3
UltraLite C++ development, 23
UltraLite runtime libraries for, 29
macros
UL_USE_DLL, 83
UltraLite applications, 83
makefiles
UltraLite embedded SQL, 61
managing
UltraLite C++ transactions, 20
MFC
UltraLite applications ActiveSync requirements, 67
MFC applications
UltraLite for Windows Mobile, 64
ml_file_transfer_info structure [UltraLite Embedded SQL API]
description, 255
ml_file_transfer_status structure [UltraLite Embedded SQL API]
description, 257
mlcrsa16.dll
UltraLite C++ development, 23, 25, 27
mlcrsafips16.dll
UltraLite C++ development, 23, 25, 27
mlczlib16.dll
UltraLite C++ development, 23, 25, 27
MLFileDownload method [UltraLite Embedded SQL API]
description, 220
mlfiletransfer.h header file
UltraLite Embedded SQL API, 217
MLFileUpload method [UltraLite Embedded SQL API]
description, 220
MLFiniFileTransferInfo method [UltraLite Embedded SQL API]
description, 221
mlf_stream_error structure [UltraLite Embedded SQL API]
Index

description, 258
mflt_stream_error_w structure [UltraLite Embedded SQL API]
  description, 259
MLFTEnableRsaE2ee method [UltraLite Embedded SQL API]
  description, 218
MLFTEnableRsaEncryption method [UltraLite Embedded SQL API]
  description, 218
MLFTEnableRsaFipsE2ee method [UltraLite Embedded SQL API]
  description, 219
MLFTEnableRsaFipsEncryption method [UltraLite Embedded SQL API]
  description, 219
MLFTEnableZlibCompression method [UltraLite Embedded SQL API]
  description, 219
MLInitFileTransferInfo method [UltraLite Embedded SQL API]
  description, 222
modes
  UltraLite C++, 15
msvcr90.dll
  UltraLite C++ development, 23
multi-row queries
  UltraLite cursors, 49
multi-threaded applications
  UltraLite C++, 5
  UltraLite embedded SQL, 37

N
namespaces
  UltraLite C++ example, 71
navigating
  UltraLite C++ API, 14
  navigating SQL result sets
    UltraLite C++, 13
network protocols
  UltraLite for Windows Mobile, 69
Next method
  ULResultSet class [UltraLite C++ API], 181
next method
  UltraLite C++ data retrieval example, 12
NULL
  UltraLite indicator variables, 46

Null terminated string UltraLite embedded SQL data type
  about, 39
null-terminated TCHAR character string UltraLite SQL data type
  about, 40
null-terminated UNICODE character string UltraLite SQL data type
  about, 40
null-terminated WCHAR character string UltraLite SQL data type
  about, 40
null-terminated wide character string UltraLite SQL data type
  about, 40

O
obfuscating
  UltraLite embedded SQL databases, 52
obfuscation
  UltraLite databases using embedded SQL, 53
  UltraLite embedded SQL databases, 52
observer synchronization parameter
  UltraLite embedded SQL example, 59
offsets
  UltraLite C++ relative, 14
OPEN statement
  UltraLite embedded SQL, 49
OpenConnection method
  ULDatabaseManager class [UltraLite C++ API], 135
OpenTable method
  ULConnection class [UltraLite C++ API], 119

P
packed decimal UltraLite embedded SQL data type
  about, 39
PEM encoded X509 certificate
  iPhone, 4
  Mac OS X, 4
performance
  UltraLite using DLL for economical memory use, 23, 27
  UltraLite using INSERT statements, 55
permissions
  UltraLite embedded SQL, 35
persistent storage
UltraLite for Windows Mobile, 64
platform requirements
UltraLite for Windows Mobile, 62
platforms
supported in UltraLite C++, 1
prepared statements
UltraLite C++, 7
PrepareStatement method
ULConnection class [UltraLite C++ API], 119
preprocessing
UltraLite embedded SQL applications, 60
 UltraLite embedded SQL development tool
settings, 61
Previous method
ULResultSet class [UltraLite C++ API], 181
previous method
UltraLite C++ data retrieval example, 12
program structure
UltraLite embedded SQL, 35
protocols
UltraLite for Windows Mobile, 69

Q
queries
UltraLite embedded SQL multi-row queries, 49
UltraLite embedded SQL single-row queries, 48

R
RegisterForEvent method
ULConnection class [UltraLite C++ API], 119
Relative method
ULResultSet class [UltraLite C++ API], 182
relative offset
UltraLite C++ API, 14
ResetLastDownloadTime method
ULConnection class [UltraLite C++ API], 120
result set schemas
UltraLite C++, 13
result sets
UltraLite C++ API schema information in , 20
UltraLite C++ navigation of, 13
RevokeConnectFrom method
ULConnection class [UltraLite C++ API], 121
Rollback method
ULConnection class [UltraLite C++ API], 121
UltraLite C++ transactions, 20
RollbackPartialDownload method
ULConnection class [UltraLite C++ API], 121
rollbacks
UltraLite C++ transactions, 20
rows
accessing in UltraLite C++ API tutorial, 71
UltraLite C++ table access of current, 18
UltraLite C++ table navigation, 14
UltraLite deletions with C++ API, 19
UltraLite insertions with C++ API, 16
UltraLite updates with C++ API, 16
runtime libraries
UltraLite applications for Windows Mobile, 23, 25,
27
UltraLite C++, 23, 25, 27
UltraLite for C++, 23, 25, 27, 29, 31
runtime library
Windows Mobile, 83

S
sample applications
UltraLite building for Windows Mobile, 62
schemas
UltraLite C++ API access, 20
UltraLite C++ API schema information in , 20
scrolling
UltraLite C++ API, 14
searching
UltraLite rows with C++, 17
security
changing the encryption key in UltraLite embedded
SQL, 53
obfuscation in UltraLite embedded SQL, 52
UltraLite database encryption, 52
SELECT statement
UltraLite C++ data retrieval example, 12
UltraLite embedded SQL single row, 48
selecting data from database tables
UltraLite C++, 12
SendNotification method
ULConnection class [UltraLite C++ API], 122
SET CONNECTION statement
multiple connections in UltraLite embedded SQL,
37
SetBinary method
ULResultSet class [UltraLite C++ API], 182
SetDatabaseOption method
ULConnection class [UltraLite C++ API], 122
ULConnection class [UltraLite C++ API], 124
string UltraLite embedded SQL data type
about, 39
fixed length, 40
variable length, 40
strong encryption
UltraLite embedded SQL, 52
supported platforms
UltraLite C++, 1
synchronization
adding to UltraLite embedded SQL applications, 53
canceling in UltraLite embedded SQL, 56
for UltraLite for Windows Mobile menu control, 69
initial in UltraLite embedded SQL, 55
invoking in UltraLite embedded SQL, 55
monitoring in UltraLite embedded SQL, 56
UltraLite C++ API tutorial, 71
UltraLite embedded SQL, 53
UltraLite embedded SQL committing changes in , 55
UltraLite embedded SQL example, 54
UltraLite for Windows Mobile introduction, 65
synchronization errors
UltraLite embedded SQL communications errors , 55
Synchronize method
ULConnection class [UltraLite C++ API], 125
SynchronizeFromProfile method
ULConnection class [UltraLite C++ API], 125
synchronizing
UltraLite C++, 22

tables
UltraLite C++ API schema information in , 21
target platforms
UltraLite C++, 1
threads
UltraLite C++ API multi-threaded applications, 5
UltraLite embedded SQL, 37
timestamp structure UltraLite embedded SQL data type
about, 40
tips
UltraLite development, 55
transaction processing
UltraLite C++ management, 20
transactions
committing in UltraLite with embedded SQL, 55
UltraLite C++ management, 20
TriggerEvent method
ULConnection class [UltraLite C++ API], 126
troubleshooting
UltraLite C++ handling errors, 21
UltraLite development, 55
UltraLite synchronization with embedded SQL, 55
UltraLite using reference expressions in SQL preprocessor, 43
TruncateTable method
ULTable class [UltraLite C++ API], 206
truncation
UltraLite FETCH, 47
tutorials
UltraLite C++ API, 71

U
ul_binary structure [UltraLite C and Embedded SQL datatypes API]
description, 90
ul_column_default_type enumeration [UltraLite C++ API]
description, 212
ul_column_name_type enumeration [UltraLite C++ API]
description, 213
ul_column_sql_type enumeration [UltraLite C and Embedded SQL datatypes API]
description, 84
ul_column_storage_type enumeration [UltraLite C and Embedded SQL datatypes API]
description, 86
ul_database_option_id enumeration [UltraLite Embedded SQL API]
description, 251
ul_database_property_id enumeration [UltraLite Embedded SQL API]
description, 252
ul_error_action enumeration [UltraLite C and Embedded SQL datatypes API]
description, 87
ul_error_info structure [UltraLite C and Embedded SQL datatypes API]
Index

description, 91
ul_index_flag enumeration [UltraLite C++ API]
description, 214
UL_RS_STATE enumeration [UltraLite C and
Embedded SQL datatypes API]
description, 84
ul_stream_error structure [UltraLite C and Embedded
SQL datatypes API]
description, 91
ul_sync_info structure
about, 54
ul_sync_info structure [UltraLite C and Embedded
SQL datatypes API]
description, 91
ul_sync_result structure [UltraLite C and Embedded
SQL datatypes API]
description, 92
ul_sync_state enumeration [UltraLite C and
Embedded SQL datatypes API]
description, 87
ul_sync_stats_download structure [UltraLite C and
Embedded SQL datatypes API]
description, 95
ul_sync_stats_upload structure [UltraLite C and
Embedded SQL datatypes API]
description, 95
ul_sync_status structure
UltraLite embedded SQL, 56
ul_sync_status structure [UltraLite C and Embedded
SQL datatypes API]
description, 96
ul_table_sync_type enumeration [UltraLite C++ API]
description, 215
UL_USE_DLL macro
about, 83
ul_validate_data structure [UltraLite C and Embedded
SQL datatypes API]
description, 97
ul_validate_status_id enumeration [UltraLite C and
Embedded SQL datatypes API]
description, 89
ULActiveSyncStream function
Windows Mobile usage, 66
ulbase.lib
UltraLite C++ development, 23, 25, 27
ULCancelGetNotification method [UltraLite
Embedded SQL API]
description, 222
ULChangeEncryptionKey function [UL ESQl]
using, 53
ULChangeEncryptionKey method [UltraLite
Embedded SQL API]
description, 222
ULCheckpoint method [UltraLite Embedded SQL
API]
description, 223
ULConnection class [UltraLite C++ API]
CancelGetNotification method, 105
ChangeEncryptionKey method, 105
Checkpoint method, 106
Close method, 106
Commit method, 106
CountUploadRows method, 107
CreateNotificationQueue method, 108
DeclareEvent method, 108
description, 102
DestroyNotificationQueue method, 109
ExecuteScalar method, 109
ExecuteScalarV method, 111
ExecuteStatement method, 112
GetChildObjectCount method, 112
GetDatabaseProperty method, 113
GetDatabasePropertyInt method, 113
GetDatabaseSchema method, 114
GetLastDownloadTime method, 114
GetLastError method, 115
GetLastIdentity method, 115
GetNotification method, 115
GetNotificationParameter method, 116
GetSqlca method, 117
GetSyncResult method, 117
GetUserPointer method, 117
GlobalAutoincUsage method, 117
GrantConnectTo method, 118
InitSyncInfo method, 118
OpenTable method, 119
PrepareStatement method, 119
RegisterForEvent method, 119
ResetLastDownloadTime method, 120
RevokeConnectFrom method, 121
Rollback method, 121
RollbackPartialDownload method, 121
SendNotification method, 122
SetDatabaseOption method, 122
SetDatabaseOptionInt method, 123
SetSynchronizationCallback method, 123
SetSyncInfo method, 123
SetUserPointer method, 124
StartSynchronizationDelete method, 124
StopSynchronizationDelete method, 124
Synchronize method, 125
SynchronizeFromProfile method, 125
TriggerEvent method, 126
ValidateDatabase method, 127
ULCountUploadRows method [UltraLite Embedded SQL API]
    description, 223
ulcpp.h header file
    UltraLite C/C++ API reference, 102
ULCreateDatabase method [UltraLite Embedded SQL API]
    description, 224
ULCreateNotificationQueue method [UltraLite Embedded SQL API]
    description, 225
ULDatabaseManager class [UltraLite C++ API]
    CreateDatabase method, 129
description, 128
DropDatabase method, 130
EnableAesDBEncryption method, 130
EnableAesFipsDBEncryption method, 131
EnableAllSynchronization method, 131
EnableHttpsSynchronization method, 131
EnableHttpSynchronization method, 132
EnableRsaE2ee method, 132
EnableRsaFipsE2ee method, 132
EnableRsaFipsSyncEncryption method, 133
EnableRsaSyncEncryption method, 133
EnableTcpipSynchronization method, 134
EnableTlsSynchronization method, 134
EnableZlibSyncCompression method, 134
Fini method, 135
Init method, 135
OpenConnection method, 135
SetErrorCallback method, 136
ValidateDatabase method, 137
ULDatabaseSchema class [UltraLite C++ API]
    Close method, 138
description, 138
GetConnection method, 138
GetNextPublication method, 139
GetNextTable method, 139
GetPublicationCount method, 139
GetTableCount method, 140
GetTableSchema method, 140
ULDatabaseSchema object
    UltraLite C++ development, 20
ULDeclareEvent method [UltraLite Embedded SQL API]
    description, 226
ULDeleteAllRows method [UltraLite Embedded SQL API]
    description, 226
ULDestroyNotificationQueue method [UltraLite Embedded SQL API]
    description, 227
ULEnableAesDBEncryption method [UltraLite Embedded SQL API]
    description, 228
ULEnableAesFipsDBEncryption method [UltraLite Embedded SQL API]
    description, 228
ULEnableHttpSynchronization method [UltraLite Embedded SQL API]
    description, 229
ULEnableRsaE2ee method [UltraLite Embedded SQL API]
    description, 229
ULEnableRsaFipsE2ee method [UltraLite Embedded SQL API]
    description, 229
ULEnableRsaFipsSyncEncryption method [UltraLite Embedded SQL API]
    description, 229
ULEnableRsaSyncEncryption method [UltraLite Embedded SQL API]
    description, 230
ULEnableTcpipSynchronization method [UltraLite Embedded SQL API]
    description, 230
ULEnableZlibSyncCompression method [UltraLite Embedded SQL API]
    description, 231
uleng16.exe
    UltraLite C++ development, 23, 27
ULError class [UltraLite C++ API]
    Clear method, 141
description, 140
GetErrorInfo method, 141
GetParameter method, 142
GetParameterCount method, 143
GetSQLCode method, 143
Index

GetSQLCount method, 143
GetString method, 144
GetURL method, 144
IsOK method, 145
ULError constructor, 141
ULError constructor
    ULError class [UltraLite C++ API], 141
ULErrorInfoInitFromSqlca method [UltraLite
Embedded SQL API]
    description, 231
ULErrorInfoParameterAt method [UltraLite
Embedded SQL API]
    description, 231
ULErrorInfoParameterCount method [UltraLite
Embedded SQL API]
    description, 232
ULErrorInfoString method [UltraLite Embedded SQL
API]
    description, 232
ULErrorInfoURL method [UltraLite Embedded SQL
API]
    description, 233
ulfips16.dll
    UltraLite C++ development, 23, 25, 27
ULGetDatabaseID method [UltraLite Embedded SQL
API]
    description, 233
ULGetDatabaseProperty method [UltraLite
Embedded SQL API]
    description, 234
ULGetErrorParameter method [UltraLite Embedded
SQL API]
    description, 234
ULGetErrorParameterCount method [UltraLite
Embedded SQL API]
    description, 235
ULGetIdentity method [UltraLite Embedded SQL
API]
    description, 235
ULGetLastDownloadTime method [UltraLite
Embedded SQL API]
    description, 235
ULGetNotification method [UltraLite Embedded SQL
API]
    description, 236
ULGetNotificationParameter method [UltraLite
Embedded SQL API]
    description, 237
ULGetSyncResult method [UltraLite Embedded SQL
API]
    description, 237
ULGlobalAutoincUsage method [UltraLite Embedded
SQL API]
    description, 238
ULGrantConnectTo method [UltraLite Embedded
SQL API]
    description, 238
ulimp.lib
    UltraLite C++ development, 23, 25
ULIndexSchema class [UltraLite C++ API]
    Close method, 146
    description, 145
    GetColumnCount method, 146
    GetColumnName method, 146
    GetConnection method, 146
    GetIndexColumnID method, 146
    GetIndexFlags method, 147
    GetName method, 147
    GetReferencedIndexName method, 147
    GetReferencedTableName method, 148
    GetTableName method, 148
    IsColumnDescending method, 148
ULIndexSchema object
    UltraLite C++ development, 21
ULInitSyncInfo function [UL ESQL]
    about, 54
ULInitSyncInfo method [UltraLite Embedded SQL
API]
    description, 239
ULIsSynchronizeMessage function [UL ESQL]
    ActiveSync usage, 66
ULIsSynchronizeMessage method [UltraLite
Embedded SQL API]
    description, 239
ULLibraryVersion method [UltraLite Embedded SQL
API]
    description, 240
ULPreparedStatement class
    UltraLite C++, 7
ULPreparedStatement class [UltraLite C++ API]
    AppendParameterByteChunk method, 150
    AppendParameterStringChunk method, 150
    Close method, 151
    description, 148
    ExecuteQuery method, 151
    ExecuteStatement method, 151
GetConnection method, 151
GetParameterCount method, 152
GetParameterID method, 152
GetParameterType method, 152
GetPlan method, 152
GetResultSetSchema method, 153
HasResultSet method, 154
SetParameterBinary method, 154
SetParameterDateTime method, 154
SetParameterDouble method, 155
SetParameterFloat method, 155
SetParameterGuid method, 155
SetParameterInt method, 156
SetParameterIntWithType method, 156
SetParameterNull method, 157
SetParameterString method, 157

ulprotos.h header file
UltraLite Embedded SQL API, 217
ULRegisterForEvent method [UltraLite Embedded SQL API]
   description, 240
ULResetLastDownloadTime method [UltraLite Embedded SQL API]
   description, 241
ULResultSet class [UltraLite C++ API]
   AfterLast method, 160
   AppendByteChunk method, 160
   AppendStringChunk method, 161
   BeforeFirst method, 163
   Close method, 163
   Delete method, 163
   DeleteNamed method, 164
   description, 158
   First method, 164
   GetBinary method, 164
   GetBinaryLength method, 165
   GetByteChunk method, 166
   GetConnection method, 168
   GetDateTime method, 168
   GetDouble method, 169
   GetFloat method, 170
   GetGuid method, 171
   GetInt method, 172
   GetIntWithType method, 172
   GetResultSetSchema method, 174
   GetRowCount method, 175
   GetState method, 175
   GetString method, 175
   GetStringChunk method, 177
   GetStringLength method, 178
   IsNull method, 180
   Last method, 181
   Next method, 181
   Previous method, 181
   Relative method, 182
   SetBinary method, 182
   SetDate method, 182
   SetDate method, 183
   SetDefault method, 184
   SetDouble method, 185
   SetFloat method, 186
   SetGuid method, 187
   SetInt method, 188
   SetInt method, 188
   SetIntWithTypeInfo method, 188
   SetNull method, 191
   SetString method, 191
   Update method, 193
   UpdateBegin method, 193
ULResultSetSchema class [UltraLite C++ API]
   description, 193
   GetColumnCount method, 194
   GetColumnID method, 194
   GetColumnName method, 195
   GetColumnPrecision method, 195
   GetColumnScale method, 196
   GetColumnSize method, 196
   GetColumnType method, 197
   GetConnection method, 197
   IsAliased method, 197
ULResultSetSchema object
   UltraLite C++, 20
ULRevokeConnectFrom method [UltraLite Embedded SQL API]
   description, 242
ULRollbackPartialDownload method [UltraLite Embedded SQL API]
   description, 242
ulrsa.lib
   UltraLite C++ development, 23
ULRSAvLibraryVersion method [UltraLite Embedded SQL API]
   description, 240
ulrt.lib
   UltraLite C++ development, 23
ulrt16.dll
ul_sync_status structure, 96
ul_validate_data structure, 97
ul_validate_status_id enumeration, 89
UltraLite C++
  accessing schema information, 20
data modification, 7
data modification with SQL, 6
data retrieval, 12
development, 3, 23, 25, 27, 29, 31
iPhone build settings, 3
iPhone include files, 3
Mac OS X include files, 3
quick start, 3
synchronizing data, 22
transaction processing, 20
UltraLite C++ API
  ul_column_default_type enumeration, 212
  ul_column_name_type enumeration, 213
  ul_index_flag enumeration, 214
  ul_table_sync_type enumeration, 215
ULConnection class, 102
ULDatabaseManager class, 128
ULDatabaseSchema class, 138
ULError class, 140
ULIndexSchema class, 145
ULPreparedStatement class, 148
ULResultSet class, 158
ULResultSetSchema class, 193
ULTable class, 198
ULTableSchema class, 206
UltraLite C/C++
  about, 3
  application tutorial, 71
  architecture, 102
  building a Windows application, 71
  INCLUDE environment variable, 71
  obfuscating UltraLite databases, 53
  supported platforms, 1
  tutorials, 71
UltraLite C/C++ API reference
  ulcpp.h header file, 102
UltraLite C/C++ Common API
  alphabetical listing, 83
UltraLite databases
  connecting in UltraLite C++, 5
  encrypting in embedded SQL, 52
  UltraLite C++ API information access, 20
  UltraLite for Windows Mobile, 64
UltraLite embedded SQL
  authorization, 35
  building CustDB application, 62
cursors, 49
developing applications, 32
fetching data, 48
functions, 217
host variables, 38
quick start, 32
synchronization, 53
using, 217
UltraLite Embedded SQL API
  db_fini method, 217
  db_init method, 217
  ml_file_transfer_info structure, 255
  ml_file_transfer_status structure, 257
  MLFileDownload method, 220
  MLFileUpload method, 220
  MLFiniFileTransferInfo method, 221
  mlft_stream_error structure, 258
  mlft_stream_error_w structure, 259
  MLFTEnableRsaE2ee method, 218
  MLFTEnableRsaEncryption method, 218
  MLFTEnableRsaFipsE2ee method, 219
  MLFTEnableRsaFipsEncryption method, 219
  MLFTEnableZlibCompression method, 219
  MLInitFileTransferInfo method, 222
  ul_database_option_id enumeration, 251
  ul_database_property_id enumeration, 252
  ULCancelGetNotification method, 222
  ULChangeEncryptionKey method, 222
  ULCheckpoint method, 223
  ULCountUploadRows method, 223
  ULCreateDatabase method, 224
  ULCreateNotificationRows method, 225
  ULDeclareEvent method, 226
  ULDeleteAllRows method, 226
  ULDeleteNotificationQueue method, 225
  ULEnableAesDBEncryption method, 228
  ULEnableAesFipsDBEncryption method, 228
  ULEnableHttpSynchronization method, 229
  ULEnableRsaE2ee method, 229
  ULEnableRsaFipsE2ee method, 229
  ULEnableRsaFipsSyncEncryption method, 229
  ULEnableRsaSyncEncryption method, 230
  ULEnableTcpipSynchronization method, 230
  ULEnableZlibSyncCompression method, 231
  ULErrorInfoInitFromSqlca method, 231
UltraLite platform requirements, 62
UltraLite synchronization menu control, 69
winsock.lib
UltraLite Windows Mobile applications, 62

x

x.509 certificate
iPhone, 4
Mac OS X, 4