UltraLite™ Embedded SQL™ User's Guide

Part number: DC50028-01-0900-01

Last modified: June 2003

Contents

About This Manual

Subject	This manual describes how to develop UltraLite database applications for handheld, mobile, or embedded devices in C/C++ using embedded SQL.
Audience	This manual is intended for all application developers writing UltraLite embedded SQL programs. Familiarity is assumed with the C/C++ programming language, with relational databases in general, and Adaptive Server Anywhere in particular.

SQL Anywhere Studio documentation

This book is part of the SQL Anywhere documentation set. This section describes the books in the documentation set and how you can use them.

The SQL Anywhere Studio documentation

The SQL Anywhere Studio documentation is available in a variety of forms: in an online form that combines all books in one large help file; as separate PDF files for each book; and as printed books that you can purchase. The documentation consists of the following books:

- **Introducing SQL Anywhere Studio** This book provides an overview of the SQL Anywhere Studio database management and synchronization technologies. It includes tutorials to introduce you to each of the pieces that make up SQL Anywhere Studio.
- **What's New in SQL Anywhere Studio** This book is for users of previous versions of the software. It lists new features in this and previous releases of the product and describes upgrade procedures.
- **Adaptive Server Anywhere Getting Started** This book is for people new to relational databases or new to Adaptive Server Anywhere. It provides a quick start to using the Adaptive Server Anywhere database-management system and introductory material on designing, building, and working with databases.
- **Adaptive Server Anywhere Database Administration Guide** This book covers material related to running, managing, and configuring databases and database servers.
- **Adaptive Server Anywhere SQL User's Guide** This book describes how to design and create databases; how to import, export, and modify data; how to retrieve data; and how to build stored procedures and triggers.
- **Adaptive Server Anywhere SQL Reference Manual** This book provides a complete reference for the SQL language used by Adaptive Server Anywhere. It also describes the Adaptive Server Anywhere system tables and procedures.
- **Adaptive Server Anywhere Programming Guide** This book describes how to build and deploy database applications using the C, C++, and Java programming languages. Users of tools such as Visual Basic and PowerBuilder can use the programming interfaces provided by those tools. It also describes the Adaptive Server Anywhere ADO.NET data provider.

- **Adaptive Server Anywhere Error Messages** This book provides a complete listing of Adaptive Server Anywhere error messages together with diagnostic information.
- **SQL Anywhere Studio Security Guide** This book provides information about security features in Adaptive Server Anywhere databases. Adaptive Server Anywhere 7.0 was awarded a TCSEC (Trusted Computer System Evaluation Criteria) C2 security rating from the U.S. Government. This book may be of interest to those who wish to run the current version of Adaptive Server Anywhere in a manner equivalent to the C2-certified environment.
- **MobiLink Synchronization User's Guide** This book describes how to use the MobiLink data synchronization system for mobile computing, which enables sharing of data between a single Oracle, Sybase, Microsoft or IBM database and many Adaptive Server Anywhere or UltraLite databases.
- **MobiLink Synchronization Reference** This book is a reference guide to MobiLink command line options, synchronization scripts, SQL statements, stored procedures, utilities, system tables, and error messages.
- **iAnywhere Solutions ODBC Drivers** This book describes how to set up ODBC drivers to access consolidated databases other than Adaptive Server Anywhere from the MobiLink synchronization server and from Adaptive Server Anywhere remote data access.
- **SQL Remote User's Guide** This book describes all aspects of the SQL Remote data replication system for mobile computing, which enables sharing of data between a single Adaptive Server Anywhere or Adaptive Server Enterprise database and many Adaptive Server Anywhere databases using an indirect link such as e-mail or file transfer.
- **SQL Anywhere Studio Help** This book includes the context-sensitive help for Sybase Central, Interactive SQL, and other graphical tools. It is not included in the printed documentation set.
- **UltraLite Database User's Guide** This book is intended for all UltraLite developers. It introduces the UltraLite database system and provides information common to all UltraLite programming interfaces.
- **UltraLite Interface Guides** A separate book is provided for each UltraLite programming interface. Some of these interfaces are provided as UltraLite components for rapid application development, and others are provided as static interfaces for C, C++, and Java development.

In addition to this documentation set, PowerDesigner and InfoMaker include their own online documentation.

Documentation formats

SQL Anywhere Studio provides documentation in the following formats:

- **Online documentation** The online documentation contains the complete SQL Anywhere Studio documentation, including both the books and the context-sensitive help for SQL Anywhere tools. The online documentation is updated with each maintenance release of the product, and is the most complete and up-to-date source of documentation.

  To access the online documentation on Windows operating systems, choose Start ➤ Programs ➤ SQL Anywhere 9 ➤ Online Books. You can navigate the online documentation using the HTML Help table of contents, index, and search facility in the left pane, as well as using the links and menus in the right pane.

  To access the online documentation on UNIX operating systems, see the HTML documentation under your SQL Anywhere installation.

- **Printable books** The SQL Anywhere books are provided as a set of PDF files, viewable with Adobe Acrobat Reader.

  The PDF files are available on the CD ROM in the *pdf_docs* directory. You can choose to install them when running the setup program.

- **Printed books** The complete set of books is available from Sybase sales or from eShop, the Sybase online store. You can access eShop by clicking How to Buy ➤ eShop at *http://www.ianywhere.com*.

Documentation conventions

This section lists the typographic and graphical conventions used in this documentation.

Syntax conventions

The following conventions are used in the SQL syntax descriptions:

- **Keywords** All SQL keywords appear in upper case, like the words ALTER TABLE in the following example:

  ALTER TABLE [*owner*.]*table-name*

- **Placeholders** Items that must be replaced with appropriate identifiers or expressions are shown like the words *owner* and *table-name* in the following example:

  ALTER TABLE [*owner*.]*table-name*

- **Repeating items** Lists of repeating items are shown with an element of the list followed by an ellipsis (three dots), like *column-constraint* in the following example:

  ADD *column-definition* [*column-constraint*, . . .]

  One or more list elements are allowed. In this example, if more than one is specified, they must be separated by commas.

- **Optional portions** Optional portions of a statement are enclosed by square brackets.

  RELEASE SAVEPOINT [*savepoint-name*]

  These square brackets indicate that the *savepoint-name* is optional. The square brackets should not be typed.

- **Options** When none or only one of a list of items can be chosen, vertical bars separate the items and the list is enclosed in square brackets.

  [**ASC** | **DESC**]

  For example, you can choose one of ASC, DESC, or neither. The square brackets should not be typed.

- **Alternatives** When precisely one of the options must be chosen, the alternatives are enclosed in curly braces and a bar is used to separate the options.

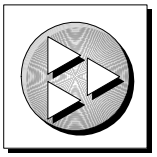
  [**QUOTES** { **ON** | **OFF** }]

  If the QUOTES option is used, one of ON or OFF must be provided. The brackets and braces should not be typed.

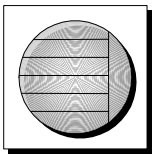
Graphic icons

The following icons are used in this documentation.

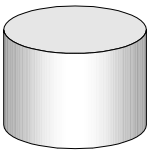
- A client application.



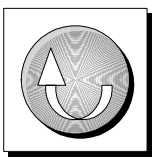
- A database server, such as Sybase Adaptive Server Anywhere.



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



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



- A programming interface.

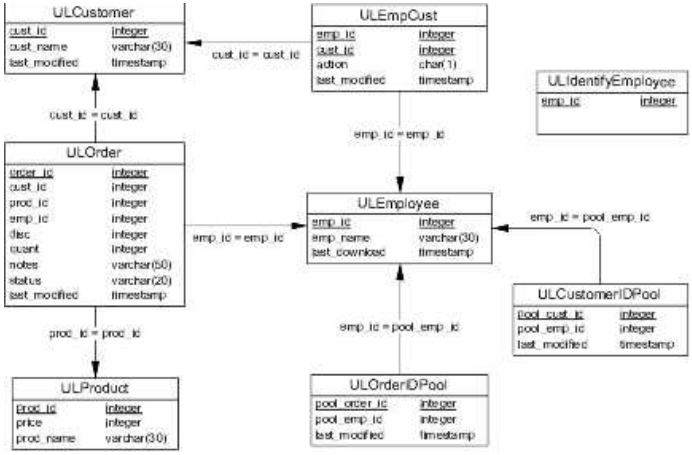
The CustDB sample database

Many of the examples in the MobiLink and UltraLite documentation use the UltraLite sample database.

The reference database for the UltraLite sample database is held in a file named *custdb.db*, and is located in the *Samples\UltraLite\CustDB* subdirectory of your SQL Anywhere directory. A complete application built on this database is also supplied.

The sample database is a sales-status database for a hardware supplier. It holds customer, product, and sales force information for the supplier.

The following figure shows the tables in the CustDB database and how they are related to each other.



Finding out more and providing feedback

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

You can provide feedback on this documentation and on the software through newsgroups set up to discuss SQL Anywhere technologies. These newsgroups can be found on the *forums.sybase.com* news server.

The newsgroups include the following:

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

Newsgroup disclaimer

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

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

CHAPTER 1

Introduction to Embedded SQL

About this chapter

This chapter introduces the embedded SQL interface to UltraLite databases. It assumes that you are familiar with the UltraLite database system and the development models it offers.

☞ For more information, see “Welcome to UltraLite” [*UltraLite Database User’s Guide*, page 3].

Contents

System requirements and supported platforms

Supported target platforms are the Palm Computing Platform and Microsoft Windows CE. Other Windows operating systems are supported for development purposes only.

Application development requires a C or C++ compiler running on a Windows operating system, such as Microsoft eMbedded Visual C++ for Windows CE development or Metrowerks CodeWarrior for Palm OS development. You must also have an Adaptive Server Anywhere reference data base.

☞ For more detailed information, see "UltraLite host platforms" [*Introducing SQL Anywhere Studio*, page 126], and "UltraLite target platforms" [*Introducing SQL Anywhere Studio*, page 136].

Developing embedded SQL applications

When developing embedded SQL applications, you mix SQL statements in with standard C or C++ source code. In order 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 database.

   Prepare an Adaptive Server Anywhere reference database that contains the tables and indexes you wish to include in 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 wish to execute, prefixed by the EXEC SQL keywords. For example:

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

3. Preprocess the *.sqc* files.

   SQL Anywhere Studio includes a SQL preprocessor (sqlpp), which reads the .sqc files, accesses an Adaptive Server Anywhere reference database, and generates .c or .cpp files. These files hold function calls to the UltraLite runtime library.

4. Compile your .c or .cpp files.

   You can compile the generated .c or .cpp files just as you compile other .c or .cpp files.

5. Link the .c or .cpp files.

   You must link the files against the UltraLite runtime library.

☞ For a full description of the embedded SQL development process, see "Building Embedded SQL Applications" on page 17.

Benefits and limitations of embedded SQL

UltraLite provides several programming interfaces, including both static development models (of which embedded SQL is one) and UltraLite components. Many of the benefits and disadvantages of embedded SQL are shared with the UltraLite static C++ API.

Embedded SQL has the following advantages:

- **Small footprint database** As embedded SQL uses an UltraLite database engine compiled specifically for each application, the footprint is generally smaller than when using an UltraLite component, especially for a small number of tables. For a large number of tables, this benefit is lost.
- **High performance** Combining the high performance of C and C++ applications with the optimization of the generated code, including data access plans, makes embedded SQL a good choice for high-performance application development.
- **Extensive SQL support** With embedded SQL you can use a wide range of SQL in your applications.

Embedded SQL has the following disadvantages:

- **Knowledge of C or C++ required** If you are not familiar with C or C++ programming, you may wish to use one of the other UltraLite interfaces. UltraLite components provide interfaces from several popular propgramming languages and tools.
- **Complex development model** The use of a reference database to hold the UltraLite database schema, together with the need to preprocess your source code files, makes the embedded SQL development process complex. The UltraLite components provide a much simpler development process.
- **SQL must be specified at design time** Only SQL statements defined at compile time can be included in your application. The UltraLite components allow dynamic use of SQL statements.

The choice of development model is guided by the needs of your particular project, and by the programming skills and experience available.

CHAPTER 2

Tutorial: Build an Application Using Embedded SQL

About this chapter

This chapter provides a tutorial to guide you through the process of developing an embedded SQL UltraLite application using eMbedded Visual C++.

☞ For an overview of the development process and background information on the UltraLite database, see "Developing embedded SQL applications" on page 3.

☞ For information on developing embedded SQL UltraLite Applications, see "Data Access Using Embedded SQL" on page 27.

☞ For a description of embedded SQL, see "Embedded SQL Library Functions" on page 101.

Contents

Introduction

In this tutorial, you create an embedded SQL source file and use it to build a simple UltraLite application. This UltraLite application can be executed on a remote device.

This tutorial assumes that you have UltraLite and Microsoft eMbedded Visual Tools installed on your computer. If you use a different C/C++ development tool, you will have to translate the eMbedded Visual C++ instructions into their equivalent for your development tool.

❖ To prepare for the tutorial

1. Create a directory to hold the files you will create.

   The remainder of the tutorial assumes that this directory is *c:\tutorial*.

Lesson 1: Configure eMbedded Visual C++

The following procedure configures eMbedded Visual C++ for UltraLite development. You may need to add additional library and include paths.

❖ To configure eMbedded Visual C++ for UltraLite development

1. Start Microsoft eMbedded Visual C++ 3.0.

   From the Start menu, choose Programs ➤ Microsoft Visual Tools ➤ eMbedded Visual C++ 3.0

2. Configure eMbedded Visual C++ to search the appropriate directories for embedded SQL header files and UltraLite library files.

   (a) Select Tools ➤ Options.

   The Options dialog is displayed.

   (b) Click the Directories tab

   (c) For each target platform and CPU combination,

   - Choose Include Files under the Show Directories For dropdown menu. Include the following directory, so that the embedded SQL header files are accessible.

     ```
     C:\Program Files\Sybase\SQL Anywhere 9\h
     ```

     If you have installed SQL Anywhere to a directory other than the default, substitute the *\h* subdirectory of your installation.
   - Choose Library Files under the Show Directories For dropdown menu. Include the UltraLite *\lib* directory, located in a platform-specific directory. For example, for the Pocket PC emulator, choose the following:

     ```
     C:\Program Files\Sybase\SQL Anywhere 9\UltraLite\ce\
             emulator30\lib
     ```

   (d) Click OK.

Lesson 2: Write an embedded SQL source file

The following procedure creates a sample program that establishes a connection with the UltraLite CustDB sample database and executes a query.

❖ To build the sample embedded SQL UltraLite application

1. Start Microsoft eMbedded Visual C++.

   Choose Start ➤ Programs ➤ Microsoft eMbedded Visual Tools ➤ eMbedded Visual C++.

2. Create a new workspace named **UltraLite**:
   - Select File ➤ New.
   - Click the Workspaces tab.
   - Choose Blank Workspace. Specify a workspace name **UltraLite** and specify *C:\tutorial* as the location to save this workspace. Click OK.

     The **UltraLite** workspace is added to the Workspace window.

3. Create a new project named **esql** and add it to the **UltraLite** workspace.
   - Select File ➤ New.
   - Click the Projects tab.
   - Choose WCE Pocket PC 2002 Application. Specify a project name **esql** and select Add To Current Workspace. Select the applicable CPUs. Click OK.
   - Choose Create An Empty Project and click Finish.

     The project is saved in the *c:\tutorial\esql* folder.

4. Create the *sample.sqc* source file.
   - Choose File ➤ New.
   - Click the Files tab.
   - Select C++ Source File.
   - Select Add to Project and select esql from the dropdown list.
   - Name the file *sample.sqc.* Click OK.
   - Copy the following source code into the file:

```
#include <stdio.h>
#include <wingdi.h>
#include <winuser.h>
#include <string.h>
#include "uliface.h"
EXEC SQL INCLUDE SQLCA;
int WINAPI WinMain( HINSTANCE hInstance, HINSTANCE
         hPrevInstance, LPSTR lpCmdLine, int nShowCmd)
{
   /* Declare fields */
   EXEC SQL BEGIN DECLARE SECTION;
      long pid=1;
      long cost;
      char pname[31];
   EXEC SQL END DECLARE SECTION;
      /* Before working with data*/
   db_init(&sqlca);
   /* Connect to database */
   EXEC SQL CONNECT "DBA" IDENTIFIED BY "SQL";
   /* Fill table with data first */
   EXEC SQL INSERT INTO ULProduct(
         prod_id, price, prod_name)
      VALUES (1, 400, '4x8 Drywall x100');
   EXEC SQL INSERT INTO ULProduct (
         prod_id, price, prod_name)
      VALUES (2, 3000, '8''2x4 Studs x1000');
   EXEC SQL COMMIT;
      /* Fetch row from database */
   EXEC SQL SELECT price, prod_name
         INTO :cost, :pname
         FROM ULProduct
         WHERE prod_id= :pid;
   /* Error handling.  If the row does not exist,
      or if    an error occurs, -1 is returned */
   if((SQLCODE==SQLE_NOTFOUND)||(SQLCODE<0)) {
      return(-1);
   }
```

```
/* Print query results */
   wchar_t query[100];
   wchar_t result[10];
   wchar_t wpname[31];
   mbstowcs(wpname, pname, 31);
   wcscpy(query, L"Product id: ");
   _ltow(pid, result, 10);
   wcscat(query, result);
   wcscat(query, L" Price: ");
   _ltow(cost, result, 10);
   wcscat(query, result);
   wcscat(query, L" Product name: ");
   wcscat(query, wpname);
   wcscpy(result, L"Result");
   MessageBox(NULL, query, result, MB_OK);
   /* Preparing to exit:
   rollback any outstanding changes and disconnect */
   EXEC SQL DISCONNECT;
   db_fini(&sqlca);
   return(0);
}
```

- Save the file.

5. Configure the *sample.sqc* source file settings to invoke the SQL preprocessor to preprocess the source file:
   - Right-click *sample.sqc* in the Workspace window and select Settings. The Project Settings dialog appears.
   - From the Settings For drop down menu, choose All Configurations.
   - In the Custom Build tab, enter the following statement in the Commands box. Ensure that the statement is entered all on one line.

     The following statement runs the SQL preprocessor *sqlpp* on the *sample.sqc* file, and writes the processed output in a file named *sample.cpp*. The SQL preprocessor translates SQL statements in the source file into C/C++.

     ```
     "%asany9%\win32\sqlpp.exe" -q -o WINDOWS -c
             "dsn=Ultralite 9.0 Sample" $(InputPath)
             sample.cpp
     ```

     ☞ For more information about the SQL preprocessor, see “The SQL preprocessor” [*ASA Programming Guide*, page 203].
   - Specify *sample.cpp* in the Outputs box.
   - Click OK to submit the changes.

6. Start the Adaptive Server Anywhere personal database server.

   By starting the database server, both the SQL preprocessor and the UltraLite analyzer will have access to your reference database. The

sample application uses the CustDB sample database *custdb.db* as a reference database and as consolidated database.

Start the database server at the command line from the *Samples\UltraLite\CusDB* directory containing *custdb.db* as follows:

```
dbeng9 custdb.db
```

Alternatively, you can start the database server by selecting Start ➤ Programs ➤ SQL Anywhere 9 ➤ UltraLite ➤ Personal Server Sample for UltraLite.

7. Preprocess the *sample.sqc* file.

   Because the sample application consists of only one source file, the preprocessor automatically runs the UltraLite analyzer as well and appends extra C/C++ code to the generated source file.

   - Select *sample.sqc* in the Workspace window. Choose Build ➤ Compile sample.sqc. A *sample.cpp* file will be created and saved in the *tutorial\esql* folder.

8. Add *sample.cpp* to the project:
   - Right-click the Source Files folder in the Workspace window and select Add Files to Folder.
   - Browse to *c:\tutorial\esql\sample.cpp* and click OK.

     The *sample.cpp* file appears inside the Source Files folder.

Explanation of the sample program

Although the sample program is simple, it contains elements that must be present in every embedded SQL source file used for database access.

The following list describes the key elements in the sample program. Use these steps as a guide when creating your own embedded SQL UltraLite application.

1. Include the appropriate header files.

   The sample program uses standard I/O, therefore the *stdio.h* header file has been included.

2. Define the SQL communications area, sqlca.

   Use the following command:

   ```
   EXEC SQL INCLUDE SQLCA;
   ```

   This definition must be your first embedded SQL statement, so place it at the end of your include list.

Prefix SQL statements
All SQL statements must be prefixed with the keywords EXEC SQL and must end with a semicolon.

3. Define host variables by creating a declaration section.

   Host variables are used to send values to the database server or receive values from the database server. Create a declaration section as follows:

   ```
   EXEC SQL BEGIN DECLARE SECTION;
      long pid=1;
      long cost;
      char pname[31];
   EXEC SQL END DECLARE SECTION;
   ```

   ☞ For information about host variables, see "Using host variables" on page 30.

4. Call the embedded SQL library function *db_init* to initialize the UltraLite runtime library.

   Call this function as follows:

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

5. Connect to the database using the CONNECT statement.

   To connect to the UltraLite sample database, you must supply the login user ID and password. Connect as user **DBA** with password **SQL** as follows:

   ```
   EXEC SQL CONNECT "DBA" IDENTIFIED BY "SQL";
   ```

6. Insert data into database tables.

   When an application is first started, its database tables are empty. When you synchronize the remote database with the consolidated database, the tables are filled with values so that you may execute select, update or delete commands.

   Rather than using synchronization, this sample code directly inserts data into the tables. Directly inserting data is a useful technique during the early stages of UltraLite development.

   If you use synchronization and your application fails to execute a query, it can be due to a problem in the synchronization process or due to a mistake in your program. To locate the source of failure may be difficult. If you directly fill tables with data in your source code rather than perform synchronization, then, if your application fails, you will know automatically that the failure is due to a mistake in your program.

   After you have tested that there are no mistakes in your program, remove the insert statements and replace them with a call to the **ULSynchronize**

function to synchronize the remote database with the consolidated database.

☞ For information on adding synchronization to an UltraLite application, see "Adding synchronization to your application" on page 15.

7. Execute your SQL query.

   The sample program executes a select query that returns one row of results. The results are stored in the previously defined host variables `cost` and `pname`.

8. Perform error handling.

   The sample program executes a select request that returns an error code, `sqlcode`. This code is negative if an error occurs; SQL_NOTFOUND is returned if there are no query results. The sample program handles these errors by returning –1.

9. Disconnect from the database.

   You should rollback or commit any outstanding changes before disconnecting.

   To disconnect, use the DISCONNECT statement as follows:

   ```
   EXEC SQL DISCONNECT;
   ```

10. End your SQL work with a call to the library function *db_fini*:

    ```
    db_fini(&sqlca);
    ```

Lesson 3: Build the sample embedded SQL UltraLite application

The following procedure uses the source file generated in the previous lesson, *sample.cpp*, to create the sample embedded SQL UltraLite application.

❖ To build the sample embedded SQL UltraLite application

1. Ensure that the Adaptive Server Anywhere personal database server is still running.
2. Configure the project settings:
   - Right-click **esql** and select Settings.
     The Project Settings dialog appears.
   - Select All Configurations under the Settings For drop down menu.
   - Click the Link tab and add the following runtime library to the Object/Library Modules box.

     ```
     ulimp.lib
     ```

   - Click the C/C++ tab. Select Preprocessor from the Category drop-down menu. Ensure that the following are included in the Preprocessor definitions:

     ```
     __NT__
     ```

   - Click OK to close the dialog.
3. Build the executable:
   - Select Build ➤ Build esql.exe.
     The **esql** executable is created. Depending on your settings, the executable may be created in a Debug directory within your tutorial directory.
4. Run the application:
   - Select Build ➤ Execute esql.exe.
     A screen appears and displays the first row of the product table.

Lesson 4: Add synchronization to your application

Once you have tested that your program is functioning properly, you can replace the code that manually insert data into the ULProduct table with instructions to synchronize the remote database with the consolidated database. Synchronization will fill the tables with data and you can subsequently execute a select query.

Synchronization via TCP/IP

You can synchronize the remote database with the consolidated database using a TCP/IP socket connection. Call `ULSynchronize` with the `ULSocketStream()` stream.

In order to synchronize with the CustDB consolidated database, the employee ID must be supplied. This ID identifies an instance of an application to the MobiLink server. You may choose a value of 50, 51, 52, or 53. The MobiLink server uses this value to determine the download content, to record the synchronization state, and to recover from interruptions during synchronization.

☞ For more information about the ULSynchronize function, see "ULSynchronize function" on page 136.

Running the sample application with synchronization

After you have made changes to *sample.sqc*, you must preprocess *sample.sqc* and rebuild *esql.exe*.

❖ To synchronize your application

1. Ensure that the Adaptive Server Anywhere database server is still running.
2. Delete the INSERT commands and add the following code. Replace *your-pc* with the name of your computer.

```
auto ul_synch_info synch_info;
ULInitSynchInfo( &synch_info );
synch_info.user_name = UL_TEXT("50");
synch_info.version = UL_TEXT("custdb 9.0");
synch_info.stream = ULSocketStream();
synch_info.send_column_names = ul_true;
synch_info.stream_parms = UL_TEXT("host=your-pc;port=2439");
ULSynchronize( &sqlca, &synch_info );
```

3. Preprocess *sample.sqc*.

Choose Build ➤ Compile *sample.sqc* to recompile the altered file. When prompted, choose to reload *sample.cpp*.

4. Build the executable.

   Select Build ➤ Build esql.exe to build the sample executable.

5. Start the MobiLink synchronization server.

   At a command prompt, execute the following command on a single line:

   ```
   dbmlsrv9 -c "DSN=UltraLite 9.0 Sample" -o ulsync.mls -vcr -x
            tcpip -za
   ```

6. Run the application:

   - Select Build ➤ Execute esql.exe to run the sample application.

     The remote database will be synchronized with the consolidated database, filling the tables in the remote database with data. The select query in the sample application will be processed, and a row of query results will appear on the screen.

CHAPTER 3

Building Embedded SQL Applications

About this chapter

This chapter describes how to build embedded SQL UltraLite applications and how to configure development tools for embedded SQL development.

There are two build processes, depending on whether you have a single embedded SQL file or multiple embedded SQL files.

Contents

Build procedure for UltraLite embedded SQL applications

This section describes a general build procedure for UltraLite embedded SQL applications. You can use a simpler modification if your application uses only a single *.sqc* file. For more information, see "Single-file build procedure" on page 21.

☞ This section assumes a familiarity with the overall embedded SQL development model. For more information, see "Using UltraLite Static Interfaces" [*UltraLite Database User's Guide*, page 195].

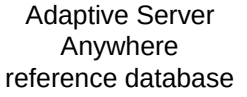
Sample code

You can find a makefile that uses this process in the *Samples\UltraLite\ESQLSecurity* directory. You require the separately-licensable transport-layer security option to build that sample.

☞ For information on obtaining the transport-layer security option, see the card in your SQL Anywhere package or see *http://www.sybase.com/detail?id=1015780*.

Procedure

The following diagram depicts the procedure for building an UltraLite embedded SQL application. In addition to your source files, you need a reference database that contains the tables and indexes you wish to use in your application.



❖ To build an UltraLite embedded SQL application

1. Start the Adaptive Server Anywhere personal database server, specifying your reference database.
2. Run the SQL preprocessor on *each* embedded SQL source file.

   The SQL preprocessor is the sqlpp command-line utility. It carries out two functions in an UltraLite development project:

   - It preprocesses the embedded SQL files, producing C files to be compiled into your application.
   - It adds the SQL statements to the reference database, for use by the UltraLite generator.

> **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 .c. When in doubt, specify the output file name explicitly, following the name of the source file.*

Use the sqlpp `-c` command-line option to connect to the reference database and the `-p` command-line option to specify a project name. Use the same project name for each embedded SQL file in your project.

☞ For detailed information about the SQL preprocessor, see "The SQL preprocessor" [*UltraLite Database User's Guide*, page 92].

☞ For information about projects, see "Creating an UltraLite project" [*UltraLite Database User's Guide*, page 204].

3. Run the UltraLite generator.

   The generator analyzes information collected while pre-processing your embedded SQL files. It prepares extra code and writes out a new C source file. This step also relies on your reference database.

   Enter the following command at a command-prompt:

   ```
   ulgen -c "connection-string" options
   ```

   where *options* depend on the specifics of your project.

   The UltraLite generator command line customizes its behavior. The following command-line switches are particularly important:

   - **-c** You must supply a connection string, to connect to the reference database.

     ☞ For information on Adaptive Server Anywhere connection strings, see "Connection parameters" [*ASA Database Administration Guide*, page 70].

   - **-f** Specify the output file name.
   - **-j** Specify the UltraLite project name.

   ☞ For more information on UltraLite generator options, see "The UltraLite generator" [*UltraLite Database User's Guide*, page 96].

4. Compile *each* C or C++ source file for the target platform of your choice. Include
   - each C files generated by the SQL preprocessor,
   - the C file made by the UltraLite generator,
   - any additional C or C++ source files that comprise your application.
5. Link *all* these object files, together with the UltraLite runtime library.

Example

- Suppose that your project contains *two* embedded SQL source files, called *store.sqc* and *display.sqc*. You could give your project the name *salesdb* and process these two commands using the following commands. (Each command should be entered on a single line.)

  ```
  sqlpp -c "uid=dba;pwd=sql" -p salesdb store.sqc
  sqlpp -c "uid=dba;pwd=sql" -p salesdb display.sqc
  ```

  These two commands generate the files *store.c* and *display.c*, respectively. In addition, they store information in the reference database for the UltraLite analyzer.

Single-file build procedure

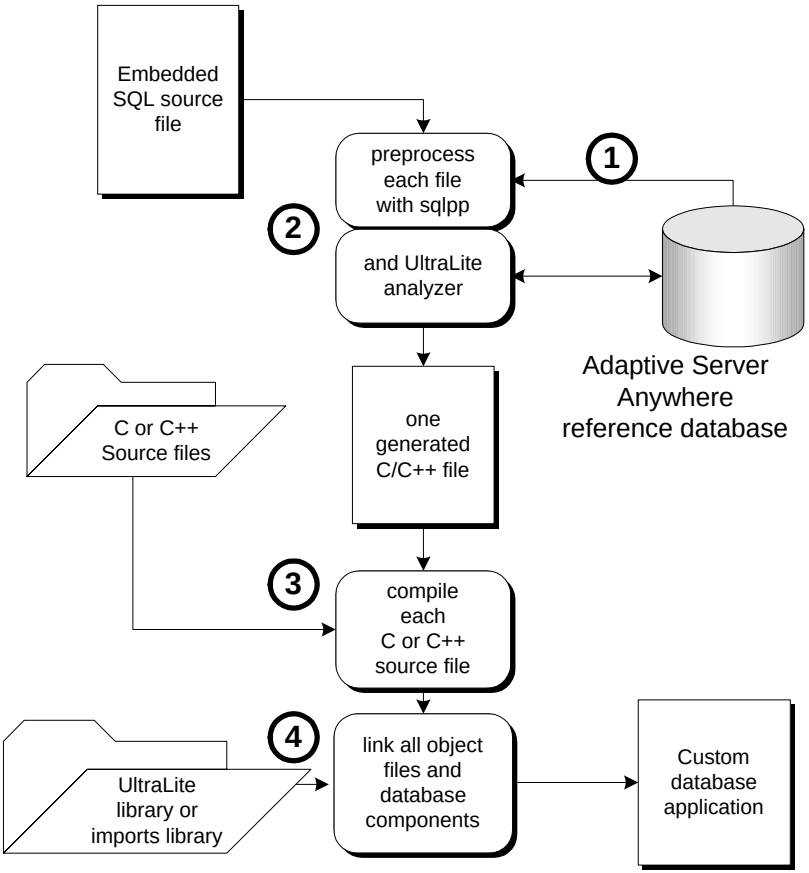
☞ This section assumes a familiarity with the overall embedded SQL development model. For more information, see "Using UltraLite Static Interfaces" [*UltraLite Database User's Guide*, page 195].

You can use a simpler single-file build procedure if the following requirements are also met:

- You are not using transport-layer security.
- You do not wish to use publications for synchronization.
- You do not need to specify an UltraLite project name.
- You have more than one embedded SQL source file.

If these requirements are not all met, you must use the general build process. For instructions, see "Build procedure for UltraLite embedded SQL applications" on page 18 .

The following diagram depicts the single-file build procedure for UltraLite database applications. In addition to your source files, you need a reference database that contains the tables and indexes you wish to use in your application.



❖ To build an UltraLite application (one embedded SQL file only)

1. Start the Adaptive Server Anywhere personal database server, specifying your reference database.
2. Preprocess the embedded SQL source file using the SQL preprocessor.

   The SQL preprocessor is the sqlpp command-line utility. The SQL preprocessor runs the UltraLite generator automatically and appends additional code to the generated C/C++ source file. This step relies on your reference database and on the database server.

   Use the sqlpp `-c` command-line option to connect to the reference database. In the single-file build procedure, do not specify a project on the SQL preprocessor command line.

   ☞ For a list of the parameters to *sqlpp*, see "The SQL preprocessor" [*ASA Programming Guide*, page 203].

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

Example

- Your application contains only *one* embedded SQL source file, called *store.sqc*. You can process this file using the following command. Do not specify a project name. This command causes the SQL preprocessor to write the file *store.c*.

  ```
  sqlpp -c "uid=dba;pwd=sql" store.sqc
  ```

  In addition, the preprocessor automatically runs the UltraLite generator, which generates more C/C++ code to implement your application database. This code is automatically appended to the file *store.c*.

Configuring development tools for embedded SQL development

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

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

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

☞ The UltraLite plug-in for Metrowerks CodeWarrior automatically provides Palm Computing platform developers with the techniques described here. For information on this plug-in, see "Developing UltraLite applications with Metrowerks CodeWarrior" on page 73.

☞ or a tutorial describing details for a very simple project, see "Tutorial: Build an Application Using Embedded SQL" on page 5.

SQL preprocessing

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

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

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

   The **development project** is defined in your development tool. It is separate from the UltraLite project name used by the UltraLite generator.

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):

     ```
     "%asany9%\win32\sqlpp.exe" -q -o WINNT
             -c connection-string -p project-name
             $(InputPath) $(InputName).c
     ```

     where *asany9* is an environment variable that points to your

SQL Anywhere installation directory, *connection-string* provides the connection to the reference database, and *project-name* is the name of your UltraLite project.

If you are generating an executable for a non-Windows platform, choose the appropriate setting instead of WINNT.

☞ For a full description of the SQL preprocessor command line, see "The SQL preprocessor" [*ASA Programming Guide*, page 203].

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

3. Compile the *.sqc* files, and add the generated *.c* 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 *.c* file, set the preprocessor definitions.
   - Under General or Preprocessor, add UL_USE_DLL to the Preprocessor definitions.
   - Under Preprocessor, add *$(asany9)\h* and any other include folders you require to your include path, as a comma-separated list.

UltraLite code generation

The following set of instructions describes how to add UltraLite code generation to your development tool.

❖ To add UltraLite code generation into a dependency-based development environment

1. Add a dummy file to your development project.

   Add a file named, for example, *uldatabase.ulg*, in the same directory as your generated files.

2. Set the build rules for this file to be the UltraLite generator command line.

   In Visual C++, use a command of the following form (which should be all on one line):

   ```
   "%asany9%\win32\ulgen.exe" -q -c "connection-string"
           $(InputName) $(InputName).c
   ```

   where *asany9* is an environment variable that points to your SQL Anywhere installation directory, *connection-string* is a connection to your reference database, and *InputName* is the UltraLite project name, and should match the root of the text file name. The output is *$(InputName).c*.

3. Set the dummy file to depend on the output files from the preprocessor.

In Visual C++, click Dependencies on the custom build page, and enter the names of the generated *.c* files produced by the SQL preprocessor.

This instructs Visual C++ to run the UltraLite generator after all the necessary embedded SQL files have been preprocessed.

4. Compile your dummy file to generate the *.c* file that implements the UltraLite database.
5. Add the generated UltraLite database file to your project and change its C/C++ settings.
6. Add the UltraLite imports library to your object/libraries modules list.

   In Visual C++, go to the project settings, choose the Link tab, and add the following to the Object/libraries module list for Windows development.

   ```
   $(asany9)\ultralite\win32\386\lib\ulimp.lib
   ```

   For other targets, choose the appropriate import library.
7. When you alter any SQL statements in the reference database, touch the dummy file, to update its timestamp and force the UltraLite generator to be run.

CHAPTER 4

Data Access Using Embedded SQL

About this chapter

This chapter describes how to write data access code for embedded SQL UltraLite applications.

Before you begin

☞ This chapter assumes an elementary familiarity with the UltraLite development process. For an overview, see "Using UltraLite Static Interfaces" [*UltraLite Database User's Guide*, page 195].

☞ For reference information, see "Embedded SQL Library Functions" on page 101.

☞ For detailed information about the SQL preprocessor, see "The SQL preprocessor" [*ASA Programming Guide*, page 203].

Contents

Introduction

The following is a very simple embedded SQL program. It updates the surname of employee 195 and commits the change.

```
#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 = 'Plankton'
      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 too simple to be useful, this example demonstrates 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 found in standard SQL. The INCLUDE SQLCA statement is one example.
- Embedded SQL provides library functions to perform some specific tasks. The functions **db_init** and **db_fini** are examples.

Before working with data

The above example demonstrates the necessary initialization statements. You must include these before working with the data in any database.

1. You must define the **SQL communications area**, sqlca, using the following command.

   ```
   EXEC SQL INCLUDE SQLCA;
   ```

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

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

2. Your first executable 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.

   ☞ For more information, see "db_init function" on page 104.

3. You must use the CONNECT statement to connect to your database.

Preparing to exit

This example also demonstrates the sequence of calls you must make when preparing to exit.

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

If you leave changes to the database uncommitted when you exit, any uncommitted operations 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 flow. The WHENEVER statement is used for error checking. The error action, GOTO in this example, is executed after any SQL statement causes an error.

Structure of embedded SQL programs

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

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

```
EXEC SQL INCLUDE SQLCA;
```

The first embedded SQL executable statement executed in any program must be a CONNECT statement. If you are not including UltraLite user authentication in your application, this CONNECT statement is ignored.

☞ For information about UltraLite user authentication in embedded SQL applications, see "Managing user IDs and passwords" on page 53, and "User authentication" [*UltraLite Database User's Guide*, page 38].

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.

Using host variables

Host variables are C variables that are identified to the SQL preprocessor. You use host variables to send values to the database server or receive values from the database server.

Declaring host variables

You can 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 that the SQL preprocessor can 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.

Example

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

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

Data types in embedded SQL

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

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

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

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

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

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

- 16-bit signed integer.

  ```
  short int i;
  unsigned short int i;
  ```

- 32-bit signed integer.

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

- 4-byte floating point number.

  ```
  float f;
  ```

- 8-byte floating point number.

  ```
  double d;
  ```

- Packed decimal number.

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

- NULL-terminated blank-padded character string.

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

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

hold **n – 1** characters.

> **Pointers to char, WCHAR, TCHAR**
> The SQL preprocessor assumes that a **pointer to char** points to a character array of size 2049 bytes and that this array can safely hold 2048 characters, plus the NULL terminator. In other words, a char* data type maps to a CHAR(2048) SQL type. If that is not the case, your application may corrupt memory. If you are using a 16-bit compiler, requiring 2049 bytes can make the program stack overflow. Instead, use a declared array, even as a parameter to a function, to let the SQL preprocessor know the size of the array. WCHAR and TCHAR behave similarly to char.

- NULL terminated UNICODE or wide character string.

  Each character occupies two bytes of space and so may contain UNICODE characters.

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

- NULL terminated system-dependent character string.

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

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

- Fixed-length blank padded character string.

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

- Variable-length character string with a two-byte length field.

  When supplying information to the database server, you must set the length field. When fetching information from the database server, the server sets the length field (not padded).

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

- Variable-length binary data with a two-byte length field.

  When supplying information to the database server, you must set the length field. When fetching information from the database server, the server sets the length field.

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

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

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

  The SQLDATETIME structure can be used to retrieve fields of 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 a programmer to manipulate this data. Note that DATE, TIME and TIMESTAMP fields can also be fetched and updated with any character type.

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

  ☞ For more information, see the DATE_FORMAT, TIME_FORMAT, TIMESTAMP_FORMAT, and DATE_ORDER database options in "Database Options" [*ASA Database Administration Guide*, page 555]. While these options cannot be set during execution of an UltraLite program, their values are identical to the settings in the reference database used to generate the program.

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

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

  The DECL_LONGVARCHAR struct may be used with more than 32K of

data. Large data may be fetched all at once, or in pieces using the GET DATA statement. Large data may be supplied to the server all at once, or in pieces by appending to a database variable using the SET statement. The data is not null terminated.

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

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

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

  The DECL_LONGBINARY struct may be used with more than 32K of data. Large data may be fetched all at once, or in pieces using the GET DATA statement. Large 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 *sqlca.h* file. The VARCHAR, BINARY, and TYPE_DECIMAL types contain a one-character array and are thus not useful for declaring host variables, but they are useful for allocating variables dynamically or typecasting other variables.

DATE and TIME database types

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

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

Host variable usage

Host variables can be used in the following circumstances:

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

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

The scope of host variables

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

The preprocessor assumes all host variables are global

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

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

Examples

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

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

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

```
// Rewritten example
#if 0
   // Declarations for the SQL preprocessor
   EXEC SQL BEGIN DECLARE SECTION;
      long emp_id;
      long manager_id;
   EXEC SQL END DECLARE SECTION;
#endif
long getManagerID( long emp_id )
{
   long manager_id = 0;
   EXEC SQL   SELECT manager_id
              INTO :manager_id
              FROM employee
              WHERE emp_number = :emp_id;
   return( manager_number );
}
void setManagerID( long emp_id, long manager_id )
{
   EXEC SQL   UPDATE employee
              SET manager_number = :manager_id
              WHERE emp_number = :emp_id;
}
```

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

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

Using expressions as host variables

Because host variables must be simple names, 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.

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

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

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

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

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

- pointer indirections

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

- array references

```
my_array[ i ]
```

- arbitrarily complex lvalues

Using host variables in C++

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

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

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

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

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

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

Using indicator variables

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

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

Example

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

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

Indicator variable values

The following table provides a summary of indicator variable usage.

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

Using indicator variables to handle NULL

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

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

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

Using indicator variables when inserting NULL

An INSERT statement can include an indicator variable as follows:

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

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

Using indicator variables when fetching NULL

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

☞ Errors and warnings are returned in the SQLCA structure, as described in "The SQL Communication Area" on page 48.

Fetching data

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

1. The SELECT statement returns at most one row.
2. The SELECT statement may return multiple rows.

Fetching one row

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

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

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

☞ Errors and warnings are returned in the SQLCA structure, as described in "The SQL Communication Area" on page 48.

Example

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

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

Fetching multiple rows

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

☞ For an introduction to cursors, see "Working with cursors" [*ASA Programming Guide*, page 21].

❖ To manage a cursor in embedded SQL

1. Declare a cursor for a particular SELECT statement, using the DECLARE statement.
2. Open the cursor using the OPEN statement.
3. Retrieve rows from the cursor one at a time using the FETCH statement.
   - Fetch rows until the SQLE_NOTFOUND warning is returned.

     ☞ Error and warning codes are returned in the variable SQLCODE, defined in the SQL communications area structure.
4. Close the cursor, using the CLOSE statement.

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

The following is a simple example of cursor usage:

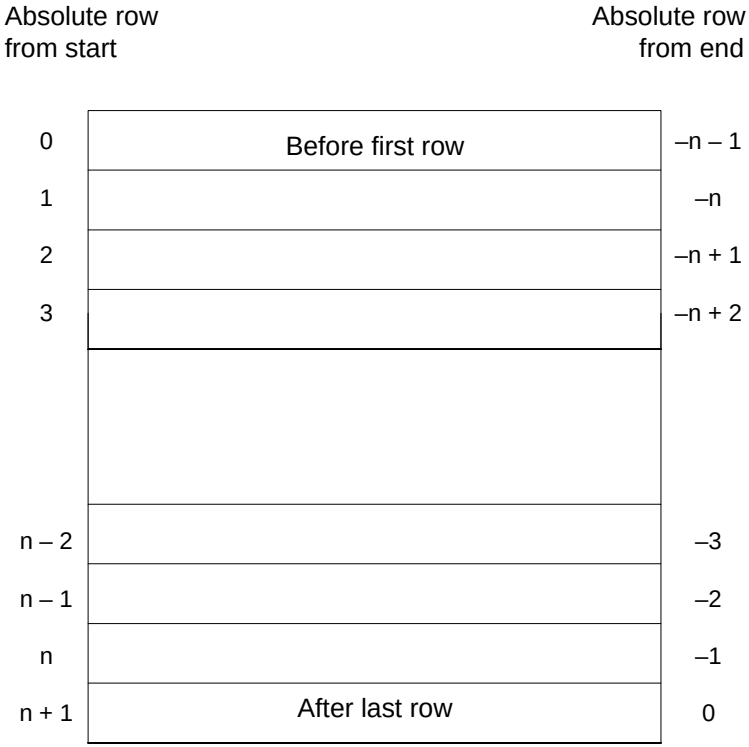
```
void print_employees( void )
{
   int status;
   EXEC SQL BEGIN DECLARE SECTION;
   char name[50];
   char sex;
   char birthdate[15];
   short int ind_birthdate;
   EXEC SQL END DECLARE SECTION;
   /* 1. Declare the cursor. */
   EXEC SQL DECLARE C1 CURSOR FOR
      SELECT emp_fname || ' ' || emp_lname,
                sex, birth_date
      FROM "DBA".employee
      ORDER BY emp_fname, emp_lname;
   /* 2. Open the cursor. */
   EXEC SQL OPEN C1;
   /* 3. Fetch each row from the cursor. */
   for( ;; ) {
      EXEC SQL FETCH C1 INTO :name, :sex,
            :birthdate:ind_birthdate;
      if( SQLCODE == SQLE_NOTFOUND ) {
         break; /* no more rows */
      } else if( SQLCODE < 0 ) {
         break; /* the FETCH caused an error */
      }
      if( ind_birthdate < 0 ) {
         strcpy( birthdate, "UNKNOWN" );
      }
      printf( "Name: %s Sex: %c Birthdate:
               %s\n",name, sex, birthdate );
   }
   /* 4. Close the cursor. */
   EXEC SQL CLOSE C1;
}
```

☞ For details of the FETCH statement, see “FETCH statement [ESQL] [SP]” [*ASA SQL Reference*, page 436].

Cursor positioning

A cursor is positioned in one of three places:

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



Order of the 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, your only guarantee is that fetching repeatedly will return each row in the result set once and only once before SQLE_NOTFOUND is returned.

Order of rows in a cursor
If the cursor must have a specific order, include an ORDER BY clause in the SELECT statement in the cursor definition. Without this clause, the ordering is unpredictable and can vary from one time to the next.

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 an 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 the end of the query results, or move it relative to the current cursor

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, an SQLE_NOTFOUND error is returned.

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

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

Cursor positioning after updates

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

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

☞ For more information about temporary tables, see “Use of work tables in query processing” [*ASA SQL User’s Guide*, page 185].

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.

The SQL Communication Area

The **SQL Communication 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.

A global SQLCA variable is defined in the generated code. The preprocessor generates an external reference for the global SQLCA variable. The external reference is named **sqlca** and is of type SQLCA. The actual global variable is declared in the imports library.

The SQLCA type is defined by the *sqlca.h* header file, which is located in the *h* subdirectory of your installation directory.

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 (see below). Some C macros are defined for referencing the **sqlcode** field and some other fields.

SQLCA fields

The fields in the SQLCA have the following meanings:

- **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 of the SQLCA structure (136 bytes).
- **sqlcode** A long integer that specifies the error code when the database detects an error on a request. Definitions for the error codes can be found in the header file *sqlerr.h.* The error code is 0 (zero) for a successful operation, positive for a warning and negative for an error.

  You can access this field directly using the **SQLCODE** macro.

  ☞ For a list of error codes, see "Database Error Messages" [*ASA Error Messages*, page 1].
- **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.

Using multiple SQLCAs

❖ **To manage multiple SQLCAs in your application**

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

   ☞ For more information, see "db_init function" on page 104.
2. The embedded SQL statement SET SQLCA is used to tell the SQL preprocessor to use a different SQLCA for database requests. Usually, a statement such as the following:

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

   is used at the top of your program or in a header file to set the SQLCA reference to point at task specific data. This statement does not generate any code and thus has no performance impact. It changes the state within the preprocessor so that any reference to the SQLCA will use the given string.

   ☞ For information about creating SQLCAs, see "SET SQLCA statement [ESQL]" [*ASA SQL Reference*, page 562].

Connection management with multiple SQLCAs

You do not need to use multiple SQLCAs to have more than one connection to a single database.

Each SQLCA can have one unnamed connection. Each SQLCA has an active or current connection. All operations on a given database connection must use the same SQLCA that was used when the connection was established.

☞ For more information, see "SET CONNECTION statement [Interactive SQL] [ESQL]" [*ASA SQL Reference*, page 553].

CHAPTER 5

Adding Non Data Access Features to UltraLite Applications

About this chapter

This chapter describes features in addition to data access you can add to UltraLite applications.

☞ For information about data access features, see "Data Access Using Embedded SQL" on page 27.

Contents

Adding user authentication to your application

UltraLite provides an optional built-in user authentication scheme. You can take advantage of this scheme to authenticate users before allowing them to connect to the UltraLite database. By default, UltraLite databases have no user userauthentication mechanism.

The UltraLite user authentication scheme does not provide the permissions features implemented in multi-user database systems and in MobiLink.

☞ For a general description of UltraLite user authentication, see "User authentication" [*UltraLite Database User's Guide*, page 38].

☞ When you create an UltraLite database with user authentication enabled, one authenticated user is created, with user ID **DBA** and password **SQL**. UltraLite permits up to four different users to be defined at a time, with both user ID and password being less than 16 characters long. Each user has full access to the database once successfully authenticated.

☞ The case sensitivity of the UltraLite user ID and password is determined by the reference database. If the reference database is case insensitive (the default) then the UltraLite database is also case insensitive, in cluding user authentication.

Enabling user authentication

Enabling user authentication requires the application to supply a valid UltraLite user ID and password when connecting to the UltraLite database. If you do not explicitly enable user authentication, UltraLite does not authenticate users.

❖ To enable user authentication (embedded SQL)

1. Call **ULEnableUserAuthentication** before calling **db_init**. For example:

```
app(){
   ...
   ULEnableUserAuthentication( &sqlca );
   db_init( &sqlca );
   ...
```

   The call to **db_init** precedes all other database activity in the application.

☞ Once you have enabled user authentication, you must add user management code to your application. For more information, see "Managing user IDs and passwords" on page 53.

Managing user IDs and passwords

There is a common sequence of events to managing user IDs and passwords.

1. New users have to be added from an existing connection. As all UltraLite databases are created with a default user ID and password of **DBA** and **SQL**, respectively, you must first attempt to connect as this initial user and implement user management only upon successful connection.
2. You cannot change a user ID: you add a user and delete an existing user. A maximum of four user IDs are permitted for each UltraLite database.
3. To change the password for an existing user ID, call the same function as adding a user ID. This function is **ULGrantConnectTo**.

Palm Computing Platform

Applications on the Palm Computing Platform do not terminate. If you wish to authenticate users whenever they return to an application from some other application, you must include the prompt for user and password information in your **PilotMain** routine.

User authentication example

The following code fragment performs user management and authentication for an embedded SQL UltraLite application.

A complete sample can be found in the *Samples\UltraLite\esqlauth* subdirectory of your SQL Anywhere directory. The code below is taken from *Samples\UltraLite\esqlauth\sample.sqc*.

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

The code carries out the following tasks:

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

☞ For more information, see "ULGrantConnectTo function" on page 119, and "ULRevokeConnectFrom function" on page 129.

Sharing MobiLink and UltraLite user IDs

Although UltraLite and MobiLink user authentication mechanisms are separate, you may wish to provide your end users with a single user ID and

password that provides both MobiLink and UltraLite user authentication. To share user IDs and passwords, store them in variables and use the same variable in the UltraLite user authentication calls and the synchronization call.

You can design your application so that, if passwords are reset at a MobiLink consolidated site, your application prompts for the new password.

❖ To prompt for a new MobiLink or UltraLite password

1. Save the user ID and password in variables.
2. Synchronize.
3. If synchronization fails because the user was not authenticated, prompt the user for a new password.
4. Update the UltraLite user's password using the appropriate function or method:
   - **ULGrantConnectTo**
5. Update the synch_info structure and synchronize again.

☞ For information on MobiLink user authentication, see "Authenticating MobiLink Users" [*MobiLink Synchronization User's Guide*, page 103].

Configuring and managing database storage

You can configure the following aspects of UltraLite persistent storage:

- The amount of memory used as a cache by the UltraLite database engine.
- Database encryption.
- Preallocation of file-system space.
- The file name for the database.
- The database page size.

This configuration is controlled by the UL_STORE_PARMS macro, which is placed in the header of your application source code so that it is visible to all **db_init()** or **ULPalmLaunch** calls. The encryption key and page size can be used on any supported C/C++ platform, while the other keys cannot be used on the Palm Computing Platform.

☞ For more information, see "UL_STORE_PARMS macro" [*UltraLite Database User's Guide*, page 216].

Encrypting UltraLite databases

By default, UltraLite databases are unencrypted on disk and in permanent memory. Text and binary columns are plainly readable within the database store when using a viewing tool such as a hex editor. Two options are provided for greater security:

- **Obfuscation** Obfuscating databases provides security against straightforward attempts to view data in the database directly using a viewing tool. It is not proof against skilled and determined attempts to gain access to the data. Obfuscation has little or no performance impact.

  ☞ For more information, see "Obfuscating an UltraLite database" on page 57.

- **Strong encryption** UltraLite database files can be strongly encrypted using the AES 128-bit algorithm, which is the same algorithm used to encrypt Adaptive Server Anywhere databases. Use of strong encryption does provide security against skilled and determined attempts to gain access to the data, but has a significant performance impact.

  Caution
  If the encryption key for a strongly encrypted database is lost or forgotten, there is no way to access the database. Under these circumstances, technical support cannot gain access to the database for you. It must be discarded and you must create a new database.

☞ For more information, see "Encrypting an UltraLite database" on page 57, and "Changing the encryption key for a database" on page 58.

Obfuscating an UltraLite database

❖ To obfuscate an UltraLite database

1. Define the UL_ENABLE_OBFUSCATION compiler directive when compiling the generated database.

   ☞ For more information, see "UL_ENABLE_OBFUSCATION macro" [*UltraLite Database User's Guide*, page 215].

Encrypting an UltraLite database

UltraLite databases are created on the first connection attempt. To encrypt an UltraLite database, you supply an encryption key before that connection attempt. On the first attempt, the supplied key is used to encrypt the database. On subsequent attempts, the supplied key is checked against the encryption key, and connection fails unless the key matches.

❖ To strongly encrypt an UltraLite database

1. Load the encryption module.

   Call **ULEnableStrongEncryption** before opening the database.

   You open a database by calling **db_init**.

   On the Palm Computing Platform, you open a database by calling **ULPalmLaunch** .

2. Specify the encryption key.

   Define the UL_STORE_PARMS macro with the parameter name **key**.

   ```
   #define UL_STORE_PARMS "key=a secret key"
   ```

   As with most passwords, it is best to choose a key value that cannot be easily guessed. The key can be of arbitrary length, but generally the longer the key, the better because a shorter key is easier to guess than a longer one. As well, 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.

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

   ☞ For more information on UL_STORE_PARMS, see "UL_STORE_PARMS macro" [*UltraLite Database User's Guide*, page 216].

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 passed in, **db_init** returns **ul_false** and SQLCODE -840 is set.

You can find a sample embedded SQL application demonstrating encryption in the directory *Samples\UltraLite\ESQLSecurity*. The encryption code is held in *Samples\UltraLite\ESQLSecurity\sample.sqc*.

Here is code from the sample:

```
static void initStoreParms(){
    char enteredKey[ 15 ];
    strcpy( storeParms, "key=" );
    // The key is used to encrypt the database on the first
         attempt.
    // On subsequent connections, the correct key is needed to
    // access the database.
    printf( "Enter encryption key: " );
    scanf( "%s", encryptionKey );
    strcat( storeParms, encryptionKey );
}

#undef  UL_STORE_PARMS
#define UL_STORE_PARMS ( initStoreParms(), storeParms )

int main( int argc, char * argv[] )
{
    /* Declare fields */
    EXEC SQL BEGIN DECLARE SECTION;
       long pid=1;
       long cost;
       char pname[31];
    EXEC SQL END DECLARE SECTION;

    /* Encryption must be enabled before working with data*/
    ULEnableStrongEncryption( &sqlca );
    db_init( &sqlca );
    if( SQLCODE == -840 ){ // bad encryption key
        printf( "Error: encryption key incorrect." );
        return( 1 );
    }

    EXEC SQL CONNECT "dba" IDENTIFIED BY "sql";
```

Changing the encryption key for a database

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

> ***Caution***
> *When the key is changed, every row in the database is decrypted using the old key and re-encrypted using the new key. This operation is unrecoverable. If the application is interrupted part-way through, the database is invalid and cannot be accessed. A new one must be created.*

❖ To change the encryption key on an UltraLite database

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

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

   ☞ For more information, see "ULChangeEncryptionKey function" on page 106.

Using the encryption key on the Palm Computing Platform

If you encrypt an UltraLite database on the Palm Computing Platform, you are prompted to re-enter the key each time you launch the application. This section describes how to add code that circumvents the re-entering of the key.

You can save the encryption key in dynamic memory as a Palm **feature**, and retrieve the key when you launch the application rather than prompting the user. Features are indexed by creator and a feature number. Users can pass in their creator ID or NULL, along with the feature number or NULL, to save and retrieve the encryption key.

The encryption key is not backed up and is cleared on any reset of the device. The retrieval of the key then fails, and the user is prompted to re-enter the key.

The following sample code illustrates how to save and retrieve the encryption key:

```
#define UL_STORE_PARMS StoreParms
static ul_char StoreParms[STORE_PARMS_MAX];
...
startupRoutine() {
   ul_char buffer[MAX_PWD];

   if( !ULRetrieveEncryptionKey(
          buffer, MAX_PWD, NULL, NULL  ) ){
      // prompt user for key
      userPrompt( buffer, MAX_PWD );
      if( !ULSaveEncryptionKey( buffer, NULL, NULL ) ) {
         // inform user save failed
      }
   }
   // build store parms
   StrCopy( StoreParms, "key=" );
   StrCat( StoreParms, buffer );
   ULPalmLaunch(&sqlca, UL_NULL );
}
```

The following sample code illustrates how to use a menu item to secure the device by clearing the encryption key:

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

☞ For more information, see "ULClearEncryptionKey function" on page 107, "ULRetrieveEncryptionKey function" on page 128, and "ULSaveEncryptionKey function" on page 130.

Defragmenting UltraLite databases

The UltraLite store is designed to efficiently reuse free space, so explicit defragmentation is not required under normal circumstances. This section describes a technique to explicitly defragment UltraLite databases, for use by applications with extremely strict space requirements.

UltraLite provides a defragmentation step function, which defragments a small part of the database. To defragment the entire database at once, call the defragmentation step function in a loop until it returns **ul_true**. This can be an expensive operation, and SQLCODE must also be checked to detect errors (an error here usually indicates a file I/O error).

Explicit defragmentation occurs incrementally under application control during idle time. Each step is a small operation.

☞ For more information, see "ULStoreDefragFini function" on page 133, "ULStoreDefragInit function" on page 134, and "ULStoreDefragStep function" on page 135.

❖ To defragment an UltraLite database

1. Obtain a p_ul_store_defrag_info information block. For example,

```
p_ul_store_defrag_info     DefragInfo;
//...
db_init( &sqlca );
DefragInfo = ULStoreDefragInit( &sqlca );
```

2. During idle time, call UlStoreDefragStep to defragment a piece of the database. For example,

```
ULStoreDefragStep( &sqlca, DefragInfo );
```

3. When complete, dispose of the defragmentation block. For example,

```
ULStoreDefragFini( &sqlca, DefragInfo );
```

Example

In this embedded SQL sample, defragmentation occurs incrementally under application control during idle time. Each defragmentation step is a small operation.

```
p_ul_store_defrag_info     DefragInfo;

idle()
{
   for( i = 0; i < DEFRAG_IDLE_STEPS; i++ ){
      ULStoreDefragStep( &sqlca, DefragInfo );
      if( SQLCODE != SQLE_NOERROR ) break;
   }
}

main()
{
   db_init( &sqlca );
   DefragInfo = ULStoreDefragInit( &sqlca );
   //
   // main application code,
   // calls idle() when appropriate...
   //
   ULStoreDefragFini( &sqlca, DefragInfo );
   db_fini( &sqlca );
}
```

To defragment the entire store at once, you can call **ULStoreDefragStep**in a loop until it returns **ul_true**. This can be an expensive operation, and you must check SQLCODE to detect errors such as file I/O errors.

Adding synchronization to your application

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

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

Overview

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

❖ **To add synchronization to your application**

1. Initialize the structure (C/C++) or object (Java) that holds the synchronization parameters.

   ☞ For information, see “Initializing the synchronization parameters” on page 62.

2. Assign the parameter values for your application.

   ☞ For information, see “Synchronization stream parameters” on page **??**.

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

   ☞ For information, see “Invoking synchronization” on page 64.

You must ensure that there are no uncommitted changes when you synchronize. For more information, see “Commit all changes before synchronizing” on page 64.

Synchronization parameters

Synchronization specifics are controlled through a set of synchronization parameters. For information on these parameters, see “Synchronization stream parameters” on page **??**.

Initializing the synchronization parameters

The synchronization parameters are stored in a C/C++ structure or Java object.

In C/C++ the members of the structure may not be well-defined on initialization. You must set your parameters to their initial values with a call to a special function. The synchronization parameters are defined in a structure declared in the UltraLite header file *ulglobal.h*.

In Java, the details of any synchronization, including the URL of the MobiLink synchronization server, the script version to use, the MobiLink user ID, and so on, are all held in a **UlSynchOptions** object.

☞ For a complete list of synchronization parameters, see "Synchronization parameters" [*UltraLite Database User's Guide*, page 162].

❖ To initialize the synchronization parameters (embedded SQL)

1. Call the **ULInitSynchInfo** function. For example:

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

Setting synchronization parameters

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

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

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

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

Invoking synchronization

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

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

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

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

❖ To invoke synchronization (HotSync)

1. Call **ULInitSynchInfo** to initialize the synchronization parameters, and call **ULPalmExit** and **ULPalmLaunch** functions to manage synchronization.

   ☞ For more information, see “ULPalmExit function” on page 124, and “ULPalmLaunch function” on page 125.

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

Commit all changes before synchronizing

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

☞ For more information on download-only synchronizations, see "download_only synchronization parameter" [*UltraLite Database User's Guide*, page 165].

Adding initial data to your application

Many UltraLite application need data in order 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.

Development tip

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

For more synchronization development tips, see "Development tips" [*MobiLink Synchronization User's Guide*, page 71].

Monitoring and canceling synchronization

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

- An API for monitoring synchronization progress and for canceling synchronization.
- A progress indicator component that implements the interface, which you can add to your application.

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 called whenever the synchronization state changes. The following section describes the synchronization state.

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

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

Handling synchronization status information

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

The **ul_synch_status** structure has the following members:

```
ul_synch_state    state;
ul_u_short        tableCount;
ul_u_short        tableIndex;
   struct {
      ul_u_long   bytes;
      ul_u_short  inserts;
      ul_u_short  updates;
      ul_u_short  deletes;
    }       sent;
    struct {
      ul_u_long   bytes;
      ul_u_short  inserts;
      ul_u_short  updates;
      ul_u_short  deletes;
    }       received;
p_ul_synch_info   info;
ul_bool           stop;
```

- **state** One of the following states:
  - **UL_SYNCH_STATE_STARTING** No synchronization actions have yet been taken.
  - **UL_SYNCH_STATE_CONNECTING** The synchronization stream has been built, but not yet opened.
  - **UL_SYNCH_STATE_SENDING_HEADER** The synchronization stream has been opened, and the header is about to be sent.
  - **UL_SYNCH_STATE_SENDING_TABLE** A table is being sent.
  - **UL_SYNCH_STATE_SENDING_DATA** Schema information or data is being sent.
  - **UL_SYNCH_STATE_FINISHING_UPLOAD** The upload stage is completed and a commit is being carried out.
  - **UL_SYNCH_STATE_RECEIVING_UPLOAD_ACK** An acknowledgement that the upload is complete is being received.

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

  ☞ For a description of the synchronization process, see "The synchronization process" [*MobiLink Synchronization User's Guide*, page 21].

- **tableCount** Returns the number of tables being synchronized. For each table there is a sending and receiving phase, so this number may be more than the number of tables being synchronized.
- **tableIndex** The current table which is being uploaded or downloaded, starting at 0. This number may skip values when not all tables are being synchronized.
- **info** A pointer to the **ul_synch_info** structure.
- **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.
- **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.
- **getUserData** Returns the user data object.
- **getStatement** Returns the statement that called the synchronization. The statement is an internal UltraLite statement, and this method is unlikely to be of practical use, but is included for completion.
- **getErrorCode** When the synchronization state is set to ERROR, this method returns a diagnostic error code.
- **isOKToContinue** This is set to **false** when **cancelSynchronization** is called. Otherwise, it is **true**.

Example

The following code illustrates a very simple observer function:

```
extern void __stdcall ObserverFunc(
    p_ul_synch_status status )
{
    printf( "UL_SYNCH_STATE is %d: ",
            status->state );
    switch( status->state ) {
        case UL_SYNCH_STATE_STARTING:
        printf( "Starting\n");
        break;
    case UL_SYNCH_STATE_CONNECTING:
        printf( "Connecting\n"  );
        break;
    case UL_SYNCH_STATE_SENDING_HEADER:
        printf( "Sending Header\n" );
        break;
    case UL_SYNCH_STATE_SENDING_TABLE:
        printf( "Sending Table %d of %d\n",
                status->tableIndex + 1,
                status->tableCount );
        break;
...
```

This observer produces the following output when synchronizing two tables:

```
UL_SYNCH_STATE is 0: Starting
UL_SYNCH_STATE is 1: Connecting
UL_SYNCH_STATE is 2: Sending Header
UL_SYNCH_STATE is 3: Sending Table 1 of 2
UL_SYNCH_STATE is 3: Sending Table 2 of 2
UL_SYNCH_STATE is 4: Receiving Upload Ack
UL_SYNCH_STATE is 5: Receiving Table 1 of 2
UL_SYNCH_STATE is 5: Receiving Table 2 of 2
UL_SYNCH_STATE is 6: Sending Download Ack
UL_SYNCH_STATE is 7: Disconnecting
UL_SYNCH_STATE is 8: Done
```

CustDB example

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

The CustDB sample code is in the *Samples\UltraLite\CustDB* subdirectory of your SQL Anywhere directory. The observer function is contained in the platform-specific subdirectories of the *CustDB* directory.

Developing multi-threaded applications

You can develop multi-threaded UltraLite applications for the Windows, and Windows CE platforms. You cannot develop multi-threaded UltraLite applications on the Palm Computing Platform, as the platform does not support such applications.

Each thread of a multi-threaded application must make its own call to **db_init()**. A SQLCA cannot be shared among different threads. Consequently, each thread must have separate connections and separate transactions from other threads.

☞ For more information, see "db_init function" on page 104.

CHAPTER 6

Developing UltraLite Applications for the Palm Computing Platform

About this chapter

This chapter describes details of development, deployment and synchronization that are specific to developing applications for the Palm Computing Platform. These instructions assume familiarity with the general UltraLite development process.

Contents

Introduction

This chapter describes features of UltraLite development specific to the Palm Computing Platform.

Development environments

You can use one of the following development environments to build UltraLite Palm applications:

- Metrowerks CodeWarrior, version 8 or 9.

  ☞ See "Developing UltraLite applications with Metrowerks CodeWarrior" on page 73.

  CodeWarrior includes a version of the Palm SDK. Depending on the particular devices you are targeting, you may want to upgrade your Palm SDK to a more recent version than that included in the development tool. Palm SDK versions 3.1, 3.5, and 4.x of the Palm SDK are supported.

- AppForge MobileVB, using the UltraLite MobileVB component. This chapter does not describe development using the MobileVB component.

☞ For general information on development environments for the Palm, including more information on each of the supported host platforms, see the Palm Computing Platform Development Zone Web site.

For information on supported development environments, see "UltraLite host platforms" [*Introducing SQL Anywhere Studio*, page 126].

Target platforms

☞ For a list of supported target operating systems, see "UltraLite target platforms" [*Introducing SQL Anywhere Studio*, page 136].

Palm-specific notes

☞ The information in this chapter concerning Palm development supplements the general information on UltraLite development provided "Using Static Development Models" [*UltraLite Database User's Guide*, page 195].

Developing UltraLite applications with Metrowerks CodeWarrior

Metrowerks CodeWarrior versions 8 and 9 are supported development platforms for Palm Computing Platform UltraLite development using the static C++ API and embedded SQL.

A CodeWarrior plug-in is supplied to make building UltraLite applications easier. This plug-in is supplied in the *UltraLite\Palm\68k\cwplugin* directory.

This section describes how to develop UltraLite applications using CodeWarrior. It assumes a familiarity with CodeWarrior programming for the Palm Computing Platform.

Installing the UltraLite plug-in for CodeWarrior

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

❖ To install the UltraLite plug-in for CodeWarrior

1. Ensure that you are running CodeWarrior version 8 or CodeWarrior version 9. You can obtain patches for CodeWarrior from the Metrowerks Web site.
2. From a command prompt, change to the *UltraLite\palm\68k\cwplugin* subdirectory of your SQL Anywhere directory.
3. Run *install.bat* to copy the appropriate files into your CodeWarrior installation directory: The *install.bat* file takes two arguments:
   - Your CodeWarrior directory
   - Your CodeWarrior version.

   For example, the following command (which should be entered on one line) installs the plug-in for CodeWarrior 9 in the default CodeWarrior installation directory.

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

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

Uninstalling the CodeWarrior plug-in

There is also a file *uninstall.bat*, that you can use in the same way as *install.bat* to uninstall the UltraLite Plug-in from CodeWarrior.

Creating UltraLite projects in CodeWarrior

This section describes how to use the UltraLite Plug-in for CodeWarrior.

❖ To create an UltraLite project in CodeWarrior

1. Start CodeWarrior.
2. Create a new project.

   From the CodeWarrior menu, choose File ➤ New. A tabbed dialog appears.

   On the Projects dialog, choose one of the available choices, and choose a name and location for the project. Click OK.
3. Choose an UltraLite stationery.

   The UltraLite plug-in adds two choices to the stationery list, one for C++ API applications and one for embedded SQL applications.

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

   This stationery is standard C stationery for embedded SQL, and standard C++ stationery for the C++ API, and contains almost-empty source files.
4. Configure the target settings for your project.

   On your project window (*.mcp*), choose the Targets tab, and click the Settings icon on the toolbar. The Project Settings window opens.

   In the tree on the left pane, choose Target ➤ UltraLite preprocessor. You can enter the settings for your project, such as which reference database to use.

When you build an embedded SQL project, the UltraLite project calls *sqlpp* and *ulgen* utilities to convert any *.sqc* files into *.c* or *.cpp* files and to generate the database code.

The plugin also adds paths to required UltraLite files, such as headers and runtime library, to the search paths.

Converting an existing CodeWarrior project to an UltraLite application

If you install the UltraLite plug-in into CodeWarrior, you will be asked to convert each existing project when you open it. In this conversion, CodeWarrior sets the default SQL preprocessor settings and saves them in the project file. This causes no disruption to projects that do not use the SQL

preprocessor. If you want to further convert a project to invoke the SQL preprocessor automatically, you need to do the following:

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

   These files are of file type **TEXT** and the Compiler is **UltraLite Preprocessor**. *All flags for these files should be unchecked.*

2. For embedded SQL applications, remove all *.cpp* files generated by the SQL preprocessor from the Files view. These files are automatically generated and re-added when the *.sqc* files are built.

3. For C++ API applications, mark the *.ulg* dummy file dirty and remove the UltraLite Files folder.

Using the UltraLite plug-in for CodeWarrior

The UltraLite plug-in for CodeWarrior integrates the UltraLite preprocessing steps (running the UltraLite generator and, for embedded SQL applications, running the SQL preprocessor) into the CodeWarrior compilation model. It ensures that the SQL preprocessor and UltraLite generator run when required.

If you change the UltraLite project name, or if you change the generated database name, you should delete the UltraLite Files folder. This forces regeneration of the generated files. To avoid filename collisions, do not use a generated database name that is the same as the *.sqc* file name.

If you change a SQL statement in a C++ API UltraLite project, or if you alter a publication used in a C++ API project, you must manually touch the dummy.*ulg* file to prompt the UltraLite generator to run.

☞ For an overview of the tasks the plug-in carries out, see “Configuring development tools for static UltraLite development” [*UltraLite Database User’s Guide*, page 210].

Using prefix files

A **prefix file** is a header file that all source files in a Metrowerks CodeWarrior project include. You should use *ulpalmXX.h*, where *XX* indicates the version of the Palm SDK you are using, from the *h* subdirectory of your SQL Anywhere Studio installation directory as your prefix file. The CodeWarrior plug-in sets this for you automatically.

If you have your own prefix file, it must include *ulpalmXX.h*. The *ulpalmXX.h* file defines macros required by Palm applications, such as the UL_PALMOS_SDK macro (which is set to the version of the Palm OS in use) and the UNDER_PALM_OS macro.

Building the CustDB sample application from CodeWarrior

CustDB is a simple sales-status application.

☞ For a diagram of the sample database schema, see "The UltraLite sample database" on page **??**.

Files for the application are located in the *Samples\UltraLite\CustDB* subdirectory of your SQL Anywhere directory. Generic files are located in the *CustDB* directory. Files specific to CodeWarrior for the Palm Computing Platform are in the following locations:

- **cwcommon** Files common to all versions of CodeWarrior.
- **cw8** Files for CodeWarrior 8.
- **cw9** Files for CodeWarrior 9.

The instructions in this section describe how to build the CustDB application using CodeWarrior 9. The process is very similar for CodeWarrior 8.

❖ To build the CustDB sample application using CodeWarrior

1. Start the CodeWarrior IDE.
2. Open the CustDB project file:
   - Choose File ➤ Open.
   - Open the project file *Samples\UltraLite\custdb\cw9\custdb.mcp* under your SQL Anywhere directory.
3. To build the target application (*custdb.prc*), choose Project ➤ Make.

You can use the UltraLite plug-in to customize settings for your own application. For more information, see "Developing UltraLite applications with Metrowerks CodeWarrior" on page 73.

Maintaining state in UltraLite applications

This discussion describes how developers can restore positions within tables so that applications appear to suspend instead of terminate when a user switches to another application. This is accomplished by providing a value for the persistent name parameter in the Open method of the ULTable object.

Palm OS applications are single threaded. To maintain the illusion that an application is running in the background after you close it, the application must save its internal state when the user switches to another application. When the application is launched again, it must restore its internal state. Saving and restoring state in a database application can be challenging, as the application must re-open previously open result sets and re-position within those result sets.

This section describes how to handle launching and closing of an UltraLite Palm application. Two Palm-specific UltraLite functions save and restore internal state information. These functions also handle synchronization if you are using the HotSync synchronization streams, but not if you are using TCP/IP or HTTP streams.

Launching an UltraLite Palm application

Whenever your UltraLite application is launched, your code must call **ULPalmLaunch** to restore state.

If your application has never been run before, or was abnormally terminated the last time it was run, the function returns a value of LAUNCH_SUCCESS_FIRST. In this case, you must initialize the UltraLite data store. Otherwise, you must *not* initialize the data store.

☞ For more information, see "ULPalmLaunch function" on page 125"PalmLaunch method" on page **??**.

Closing an UltraLite Palm application

Whenever your UltraLite application is closed, and the user switches to another application, your code must call**ULPalmExit** to save its state. Some kinds of data cannot be kept open during the time that you move away from an UltraLite application.

Do not call **db_fini** to close the application. Instead, call **ULPalmExit**. All connections (on a single SQLCA) and cursors remain open.

☞ For more information, see "ULPalmExit function" on page 124, and "PalmExit method" [*UltraLite Static C++ User's Guide*, page 91].

Building multi-segment applications

☞ Application code for the Palm Computing Platform must be divided into **segments**. For CodeWarrior, these segments are at most 64 kb in size. This section describes how to manage the assignment of code into segments.

☞ UltraLite applications include the following types of code:

- **User-defined code** Application code, including the *.cpp* file generated by the SQL Preprocessor.
- **Generated code for SQL statements** Code generated by the UltraLite Analyzer to execute SQL statements.
- **Generated code for the database schema** Code generated by the UltraLite Analyzer to represent the database tables.
- **Runtime library** The UltraLite runtime library is compiled as multi-segment code. Segment names of the form ULRT*n* and ULRT*nn* are reserved for the UltraLite runtime libraries.

☞ Building multi-segment applications is a general feature of application development for the Palm Computing Platform, whether or not you are using UltraLite. Some familiarity with building multi-segment applications using your development tool is assumed. User-defined code is no different to other standard Palm applications. For a reminder about assigning user-defined code to segments, see "Assigning user-defined code to segments" on page 79.

You can partition generated code into segments in the following ways:

- Enable multi-segment code generation, but let the UltraLite Analyzer assign segments in a default manner.

  ☞ For more information, see "Enabling multi-segment code generation" on page 78.

- Enable multi-segment code-generation and explicitly assign segments yourself.

  ☞ For more information, see "Explicitly assigning segments" on page 79.

Enabling multi-segment code generation

This section describes how to instruct the UltraLite Analyzer to generate multi-segment code using its default scheme. If you wish to customize the assignment of code to segments by explicitly assigning functions to

segments, you can do so. For more information, see "Explicitly assigning segments" on page 79.

You enable generated code segments by defining macros.

❖ To enable multi-segment code generation

1. Define a prefix file for your CodeWarrior project with the following contents:

```
#define UL_ENABLE_SEGMENTS
#include "ulpalmXX.h"
```

   where XX=30, 31, 35, or 40.

   ☞ For more information, see "UL_ENABLE_SEGMENTS macro" [*UltraLite Database User's Guide*, page 216].

Notes

When multi-segment code generation is enabled, the default behavior of the UltraLite Analyzer is as follows:

- The generated schema code fits into a single segment and is assigned to a segment named ULSEGDB.
- For the C++ API, the generated statement code is assigned to a segment named ULSEGDEF.
- For embedded SQL, the generated statement code is assigned to a segment with a generated name based on the *.sqc* file. All the code for a single *.sqc* file goes into a single segment.

Explicitly assigning segments

This section describes how to explicitly assign the generated code for SQL statements to segments. You must first enable multi-segment code generation as described in "Enabling multi-segment code generation" on page 78.

Explicit segment assignment requires a database upgraded to version 8 or later standards.

❖ To explicitly assign generated statement code to segments

1. Split your *.sqc* files into separate files. The generated code for the statements in each *.sqc* file is placed into a separate segment.

Assigning user-defined code to segments

Assigning user-defined code to segments is a standard part of programming applications for the Palm Computing Platform. This section is intended as a reminder for Palm programmers.

❖ To assign user-defined code to segments (CodeWarrior)

1. Add the following line at various places in your *.sqc* file or *.cpp* file:

   ```
   #pragma segment segment-name
   ```

   where *segment-name* is a unique name for the segment This forces code after each `#pragma` line to be in a separate segment.

The first segment

You must ensure that **PilotMain** and all functions called in **PilotMain** are in the first segment.

If necessary, you can add a line of the following form before your startup code:

```
#pragma segment segment-name
```

where *segment-name* is the name of your first segment.

For more information on prefix files and segments, see your Palm developer documentation.

Adding HotSync synchronization to Palm applications

If you use HotSync, then you synchronize by calling **ULPalmLaunch** when your application is launched, and **ULPalmExit** when your application is closed. Do not use **ULSynchronize** for HotSync synchronization.

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

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

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

   For example:

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

   if( !ULPalmExit( &sqlca, &info ) ) {
     return( false );
   }
   ```

3. Call **ULPalmLaunch**.

☞ For more information, see "Launching and closing UltraLite applications" on page 77, and "Synchronization parameters" [*UltraLite Database User's Guide*, page 162].

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

Specifying the stream parameters

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

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

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

  ☞ For a list of available values, see "Stream parameters reference" [*UltraLite Database User's Guide,* page 179].

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

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

  ☞ For more information, see "HotSync configuration overview" [*MobiLink Synchronization User's Guide,* page 211].

See also

☞ For information about configuring HotSync, including a description of how to set up your MobiLink HotSync conduit, see "Configuring the MobiLink HotSync conduit" [*MobiLink Synchronization User's Guide,* page 214].

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

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

☞ For a general description of how to add synchronization to UltraLite applications, see "Adding synchronization to your application" on page 62.

Transport layer security on the Palm Computing Platform

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

☞ For information on transport-layer security, see "Transport-Layer Security" [*MobiLink Synchronization User's Guide*, page 337].

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

When using TCP/IP, HTTP, or HTTPS synchronization, **ULPalmLaunch** and **ULPalmExit()** save and restore the state of the application on exiting and activating the application, but do not participate in synchronization. These functions take the **ul_synch_info** structure as an argument, but in this case do not use it. You should set the stream member to NULL (the default) when calling **ULPalmExit()** or **ULPalmLaunch** .

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

☞ For information on the **ul_synch_info** structure, see "Stream parameters reference" [*UltraLite Database User's Guide*, page 179].

Deploying Palm applications

This section describes the following aspects of deploying Palm applications:

- Deploying the application.

  ☞ See "Deploying applications on the Palm Computing Platform" on page **??**.

- Deploying the MobiLink synchronization conduit for HotSync.

  ☞ See "Deploying the MobiLink HotSync conduit" [*MobiLink Synchronization User's Guide*, page 216].

- Deploying an initial copy of the UltraLite database.

  ☞ See "Deploying UltraLite databases on the Palm Computing Platform" on page **??**.

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

❖ To install an application on a Palm device

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

Deploying the MobiLInk synchronization conduit

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

☞ For more information about installing the MobiLink synchronization conduit, see "Deploying the MobiLink HotSync conduit" [*MobiLink Synchronization User's Guide*, page 216].

Deploying UltraLite databases

If you deploy your application without a database, the database is created the first time it is accessed from the application. The user must then download an initial copy of data on the first synchronization. You can use the **ULUtil** utility to back up the UltraLite database to the PC. To deploy many UltraLite databases with an initial database including data, you can perform an initial synchronization and then back up the UltraLite database. The database can be deployed on other devices so they do not need to perform an initial synchronization.

☞ For more information, see "The UltraLite utility" [*UltraLite Database User's Guide*, page 103].

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

☞ For information on installing the synchronization conduit, see "Configuring the MobiLink HotSync conduit" [*MobiLink Synchronization User's Guide*, page 214].

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

If you wish the backup bit to remain set to true, you can set the store parameter **palm_allow_backup** in UL_STORE_PARMS.

☞ For more information, see "UL_STORE_PARMS macro" [*UltraLite Database User's Guide*, page 216].

CHAPTER 7

Developing UltraLite Applications for Windows CE

About this chapter

This chapter describes details of development, deployment and synchronization that are specific to Windows CE. These instructions assume familiarity with the general development process. They assist in building the CustDB sample application, included with your UltraLite software, on each of these platforms.

Contents

Introduction

This section contains instructions pertaining to building UltraLite applications for use under Microsoft Windows CE.

☞ For a list of supported host platforms and development tools for Windows CE development, and for a list of supported target Windows CE platforms, see "Supported platforms for C/C++ applications" on page **??**.

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

Preparing for Windows CE development

The recommended development environment for Windows CE at the time of writing is Microsoft eMbedded Visual C++ 3.0. This development environment is available from Microsoft as part of eMbedded Visual Tools.

☞ You can download eMbedded Visual C++ from the Microsoft Developer Network at *http://www.microsoft.com/mobile/downloads/emvt30.asp*.

A first application

A sample eMbedded Visual C++ 3.0 project is provided in the *Samples\UltraLite\CEStarter* directory under your SQL Anywhere directory. The workspace file is *Samples\UltraLite\CEStarter\ul_wceapplication.vcw*.

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

- Compiler settings:
  - Add *$(ASANY9)\h* to the include path.
  - Define appropriate compiler directives. For example, the UNDER_CE macro should be defined for eMbedded Visual C++ projects.
- Linker settings:
  - Add "*$(ASANY9)\ultralite\ce\processor\lib\ulrt.lib*" where *processor* is the target processor for your application.
  - Add *winsock.lib*.
- The *.sqc* file:
  - Add *ul_database.sqc* and *ul_database.cpp* to the project
  - Add the following custom build step for the *.sqc* file:

```
"$(ASANY9)\win32\sqlpp" -q -c "dsn=UltraLite 9.0 Sample"
        $(InputPath) ul_database.cpp
```

  - Set the output file to *ul_database.cpp*.
  - Disable the use of precompiled headers for *ul_database.cpp*.

Choosing how to link the runtime library

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

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

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

❖ **To build and deploy an application using the UltraLite runtime DLL**

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

Building the CustDB sample application

CustDB is a simple sales-status application. It is located in the UltraLite *samples* directory of your Adaptive Server Anywhere installation. Generic files are located in the *CustDB* directory. Files specific to Windows CE are located in the *ce* subdirectory of *CustDB*.

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

☞ For a diagram of the sample database schema, see "The UltraLite sample database" on page **??**.

❖ To build the CustDB sample application

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

   When eMbedded Visual C++ has finished building the application, it automatically attempts to upload it to the remote device.
7. Start the synchronization server:
   - To start the MobiLink synchronization server, select Programs ➤ Sybase SQL Anywhere 9 ➤ MobiLink ➤ Synchronization Server Sample.
8. Run the CustDB application:

   Press CTRL+F5 or select Build ➤ Execute CustDB.exe

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

The build process uses the SQL preprocessor, *sqlpp*, to preprocess the file *CustDB.sqc* into the file *CustDB.c*. 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 followed by one *ulgen* command to create the customized remote database.

☞ For more information, see "Pre-processing your embedded SQL files" on page **??**.

Storing persistent data

The UltraLite database is stored in the Windows CE file system. The default file is *\UltraLiteDB\ul_<project>.udb*, with *project* being truncated to eight characters. You can override this choice using the **file_name** parameter which specifies the full pathname 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 in order for the file name to be valid, the application must ensure that any directories are created before calling **db_init**.

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

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

Example

The following sample embedded SQL code sets the **file_name** parameter:

```
#undef UL_STORE_PARMS
#define UL_STORE_PARMS UL_TEXT(
   "file_name=\\uldb\\my own name.udb;cache_size=128k" )
...
db_init( &sqlca );
```

Deploying Windows CE applications

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

❖ **To build and deploy an application using the UltraLite runtime DLL**

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

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

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

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

Deploying applications that use ActiveSync

Applications that use ActiveSync synchronization must be registered with ActiveSync and copied to the device. The MobiLink provider for ActiveSync must also be installed.

☞ For more information, see “Deploying applications that use ActiveSync” [*MobiLink Synchronization User’s Guide*, page 225], “Installing the MobiLink provider for ActiveSync” [*MobiLink Synchronization User’s Guide*, page 223] and “Registering applications for use with ActiveSync” [*MobiLink Synchronization User’s Guide*, page 224].

Assigning class names for applications

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

Microsoft Foundation Classes (MFC) dialog boxes are given a generic class name of **Dialog**, which is shared by all dialogs in the system. This section

describes how to assign a distinct class name for your application if you are using MFC and eMbedded Visual C++.

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

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

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

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

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

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

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

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

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

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

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

   ```
   m_pMainWnd = &dlg;
   ```

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

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

5. Save your changes.

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

6. Modify the resource file for your project.

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

Locate the resource ID of your main dialog.

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

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

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

- Save the *.rc* file.

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

   ☞ For information, see "Adding ActiveSync synchronization (MFC)" on page 97.

Synchronization on Windows CE

UltraLite applications on Windows CE can synchronize through the following streams:

- ActiveSync See "Adding ActiveSync synchronization to your application" on page 96
- TCP/IP See "TCP/IP, HTTP, or HTTPS synchronization from Windows CE" on page 99.
- HTTP See "TCP/IP, HTTP, or HTTPS synchronization from Windows CE" on page 99.

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

☞ For information on adding synchronization to your application, see "Adding synchronization" on page **??**. For detailed information on synchronization parameters, see "Synchronization stream parameters" on page **??**.

Adding ActiveSync synchronization to your application

ActiveSync is synchronization software for Microsoft Windows CE handheld devices. UltraLite supports ActiveSync versions 3.1 and 3.5.

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

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

☞ For information on setting up ActiveSync synchronization, see "Deploying applications that use ActiveSync" on page 93.

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.

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

Adding ActiveSync synchronization (Windows API)

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

Here is an example of how to handle the message:

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

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

☞ For more information, see "ULIsSynchronizeMessage function" on page 122.

Adding ActiveSync synchronization (MFC)

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

☞ Your application must create and register a custom window class name for notification. See "Assigning class names for applications" on page 93.

❖ To add ActiveSync synchronization in the main dialog class

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

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

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

2. Implement the message handler.

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

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

   The return value of this function should be 0.

   ☞ For information on handling the synchronization message, see "ULIsSynchronizeMessage function" on page 122.

❖ To add ActiveSync synchronization in the Application class

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

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

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

☞ For information on handling the synchronization message, see "ULIsSynchronizeMessage function" on page 122.

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

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

☞ For more information, see "Adding synchronization to your application" on page 62.

CHAPTER 8

Embedded SQL Library Functions

About this chapter

This chapter lists functions that can be used in UltraLite embedded SQL applications. Use the EXEC SQL INCLUDE SQLCA command to include prototypes for the functions in this chapter.

Contents

db_fini function

Prototype

unsigned short **db_fini(** SQLCA * *sqlca* **);**

Description

Frees resources used by the UltraLite runtime library.

You must not make any other library calls or execute any embedded SQL commands after **db_fini** is called. If an error occurs during processing, the error code is set in SQLCA and the function returns 0.If there are no errors, a non-zero value is returned.

You need to call **db_fini** once for each SQLCA being used.

Palm Computing Platform
Do not call **db_fini** on the Palm Computing Platform. The database must be kept open when you leave the application. Use **ULPalmExit** to save the state of the application between sessions instead of calling **db_fini**.

See also

"db_init function" on page 104

db_init function

Prototype

unsigned short **db_init(**SQLCA * *sqlca* **) ;**

Description

Initializes the UltraLite runtime library and creates a new UltraLite database, if one does not exist.

This function must be called before any other library call is made, and before any embedded SQL command is executed. Exceptions to this rule are as follows:

- On the Palm Computing Platform, the **ULPalmLaunch** function can be called before **db_init**. The resources that this library requires for your program are allocated and initialized on this call.

  On the Palm Computing Platform, call **db_init** whenever **ULPalmLaunch** returns LAUNCH_SUCCESS_FIRST. For more information, see "ULPalmLaunch function" on page 125.

- Functions that configure database storage can be called. These functions have names starting with **ULEnable**.

If there are any errors during processing (for example, during initialization of the persistent store), they are returned in the SQLCA and 0 is returned. If there are no errors, a non-zero value is returned and you can begin using embedded SQL commands and functions.

In most cases, this function should be called only once (passing the address of the global **sqlca** variable 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. Subsequent connections and transactions that use this SQLCA must be carried out on a single thread.

See also

"db_fini function" on page 103

"ULPalmLaunch function" on page 125

"Developing multi-threaded applications" on page 70

ULActiveSyncStream function

Prototype

ul_stream_defn **ULActiveSyncStream(** void **);**

Description

Defines an ActiveSync stream suitable for synchronization.

The ActiveSync stream is available only on Windows CE devices.

Synchronization using ULActiveSyncStream must be initiated from the ActiveSync software. The application receives a message, which must be handled in its **WindowProc** function. You can use **ULIsSynchronizeMessage** to identify the message as an instruction to synchronize.

See also

"ULIsSynchronizeMessage function" on page 122

"ULSynchronize function" on page 136

"Synchronize method" [*UltraLite Static C++ User's Guide*, page 87]

"ActiveSync synchronization stream parameters" [*UltraLite Database User's Guide*, page 179]

ULChangeEncryptionKey function

Prototype: ul_bool **ULChangeEncryptionKey(** SQLCA *_sqlca_, ul_char *_new_key_ **);**

Description: Changes the encryption key for an UltraLite database.

> ***Caution***
> *When the key is changed, every row in the database is decrypted using the old key and re-encrypted using the new key. This operation is unrecoverable. If the application is interrupted part-way through, the database is invalid and cannot be accessed. A new one must be created.*

See also: "Changing the encryption key for a database" on page 58

ULClearEncryptionKey function

Prototype

ul_bool **ULClearEncryptionKey(**
ul_u_long * *creator*,
ul_u_long * *feature-num* **);**

Description

On the Palm Computing Platform the encryption key is saved in dynamic memory as a Palm **feature**. Features are indexed by creator and a feature number.

This function clears the encryption key.

Parameters

creator A pointer to the creator ID of the feature holding the encryption key. A value of NULL is the default.

feature-num A pointer to the feature number holding the encryption key. A value of NULL uses the UltraLite default, which is feature number 100.

See also

"ULRetrieveEncryptionKey function" on page 128

"ULSaveEncryptionKey function" on page 130

"Using the encryption key on the Palm Computing Platform" on page 59

ULCountUploadRows function

Prototype

ul_u_long **ULCountUploadRows (**
SQLCA * *sqlca*,
ul_publication_mask *publication-mask*,
ul_u_long *threshold* **);**

Description

Returns the number of rows that need to be synchronized, either in a set of publications or in the whole database.

One use of the function is to prompt users to synchronize.

Parameters

sqlca A pointer to the SQLCA.

publication-mask A set of publications to check. A value of 0 corresponds to the entire database. The set is supplied as a mask. For example, the following mask corresponds to publications PUB1 and PUB2.:

```
UL_PUB_PUB1 | UL_PUB_PUB2
```

☞ For more information on publication masks, see “Designing sets of data to synchronize separately” [*UltraLite Database User’s Guide*, page 156].

threshold A value that determines the maximum number of rows to count, and so limits the amount of time taken by the call. A value of 0 corresponds to no limit. A value of 1 determines if any rows need to be synchronized.

Example

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

```
count = ULCountUploadRows( sqlca, 0, 0 );
```

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

```
count = ULCountUploadRows( sqlca,
UL_PUB_PUB1 | UL_PUB_PUB2, 1000 );
```

The following call checks to see if any rows need to be synchronized:

```
count = ULCountUploadRows( sqlca, UL_SYNC_ALL, 1 );
```

ULDropDatabase function

Prototype

ul_bool **ULDropDatabase (** SQLCA * *sqlca*, ul_char * *store-parms* **);**

Description

Delete the UltraLite database file.

> ***Caution***
> *This function deletes the database file and all data in it. Use with care.*

Do not call this function while a database connection is open. Call this function only before **db_init** or after **db_fini**.

On the Palm OS, call this function only after **ULPalmExit** or before **ULPalmLaunch** (but after any **ULEnable** functions have been called)

Parameters

sqlca A pointer to the SQLCA.

store-parms A string of connection parameters, including the file name to delete as a keyword-value pair of the form **file_name=***file.udb*. It is often convenient to use the UL_STORE_PARMS macro as this argument. A value of UL_NULL deletes the default database filename.

☞ For more information, see “UL_STORE_PARMS macro” [*UltraLite Database User’s Guide*, page 216].

Returns

- **ul_true** Indicates that database files was successfully deleted.
- **ul_false** The detailed error message is defined by the sqlcode field in the SQLCA. The usual reason for failure is that an incorrect filename was supplied or that access to the file was denied, perhaps because it is opened by an application.

Example

The following call deletes the UltraLite database file *myfile.udb*.

```
#define UL_STORE_PARMS UL_TEXT("file_name=myfile.udb")
if( ULDropDatabase(&sqlca;, UL_STORE_PARMS ) ){
    // success
};
```

ULEnableFileDB function

Prototype

void **ULEnableFileDB(** SQLCA * *sqlca* **);**

Description

Use a file-based data store on a device operating the Palm Computing Platform version 4.0 or later. To use the file-based data store on a Palm expansion card, an UltraLite application must call **ULEnableFileDB** to load the persistent storage file-I/O modules before calling **ULPalmLaunch**.

This function can be used by C++ API applications as well as embedded SQL applications.

Parameters

sqlca A pointer to the SQLCA. This argument is supplied even in C++ API applications.

Examples

The following code sample illustrates the use of the **ULEnableFileDB** function, which is called before **ULPalmLaunch**.

```
ULEnableFileDB( &sqlca );
switch( ULPalmLaunch( &sqlca, &sync_info ) ( {
case LAUNCH_SUCCESS_FIRST:
   // do init
   break;
case LAUNCH_SUCCESS:
   // do something
   break;
case LAUNCH_FAIL:
   // handle error
   break;
}
```

See also

"ULEnablePalmRecordDB function" on page 112

ULEnableGenericSchema function

Prototype

void **ULEnableGenericSchema(** SQLCA * *sqlca* **);**

Description

When a new UltraLite application is deployed to a device, UltraLite by default re-creates an empty database, losing any data that was in the database before the new application was deployed. If you call **ULEnableGenericSchema**, the existing database is instead upgraded to the schema of the new application.

This function can be used by C++ API applications as well as embedded SQL applications. It must be called before **dbinit** or **ULData.Open()**. An exception is the Palm Computing Platform, where there is no need to close all cursors before upgrading. Immediately following an upgrade on the Palm Computing Platform the LAUNCH_SUCCESS_FIRST launch code is returned.

Backup before upgrading
It is strongly recommended that you backup your data before attempting an upgrade, either by copying the database file or by synchronizing.

For more information about the schema upgrade process, see "How the schema upgrade works" [*UltraLite Database User's Guide*, page 31].

Parameters

sqlca A pointer to the SQLCA. This argument is supplied even in C++ API applications.

ULEnablePalmRecordDB function

Prototype

void **ULEnablePalmRecordDB(** SQLCA * *sqlca* **);**

Description

Use a standard record-based data store on a device operating the Palm Computing Platform. You must call **ULEnablePalmRecordDB** or **ULEnableFileDB** before calling **ULPalmLaunch**.

This function can be used by C++ API applications as well as embedded SQL applications.

Parameters

sqlca A pointer to the SQLCA. This argument is supplied even in C++ API applications.

Examples

The following code sample illustrates the use of the **ULEnablePalmRecordDB** function, which is called before **ULPalmLaunch**.

```
ULEnablePalmRecordDB( &sqlca );
switch( ULPalmLaunch( &sqlca, &sync_info ) ( {
case LAUNCH_SUCCESS_FIRST:
   // do init
   break;
case LAUNCH_SUCCESS:
   // do something
   break;
case LAUNCH_FAIL:
   // handle error
   break;
}
```

See also

"ULEnableFileDB function" on page 110

ULEnableStrongEncryption function

Prototype	void **ULEnableStrongEncryption(** SQLCA * *sqlca* **)**
Description	Strongly encrypt an UltraLite database. This function can be used by C++ API applications as well as embedded SQL applications. It must be called before **dbinit()** or **ULData.Open()**.
Parameters	**sqlca** A pointer to the SQLCA. This argument is supplied even in C++ API applications.
See also	"Encrypting UltraLite databases" on page 56 "Changing the encryption key for a database" on page 58

ULEnableUserAuthentication function

Prototype

void **ULEnableUserAuthentication(** SQLCA * *sqlca* **);**

Description

Enable user authentication in the UltraLite application.

If you do not call this function, no user ID or password is required to access an UltraLite database. With this function, your application must supply a valid user ID and password. UltraLite databases are created with a single authenticated user ID **DBA** which has initial password **SQL**.

This function can be used by C++ API applications as well as embedded SQL applications. It must be called before **dbinit()** or **ULData.Open()**.

See also

"User authentication" [*UltraLite Database User's Guide*, page 38]

"Adding user authentication to your application" on page 52

ULGetLastDownloadTime function

Prototype

ul_bool **ULGetLastDownloadTime(**
SQLCA * *sqlca*,
ul_publication_mask *publication-mask*,
DECL_DATETIME * *value* **);**

Description

Obtains the last time a specified publication was downloaded.

Parameters

sqlca A pointer to the SQLCA.

publication-mask A set of publications for which the last download time is retrieved. A value of 0 corresponds to the entire database. The set is supplied as a mask. For example, the following mask corresponds to publications PUB1 and PUB2.:

```
UL_PUB_PUB1 | UL_PUB_PUB2
```

☞ For more information on publication masks, see "Designing sets of data to synchronize separately" [*UltraLite Database User's Guide*, page 156].

value A pointer to the DECL_DATETIME structure to be populated.

A value of **January 1, 1990** indicates that the publication has yet to be synchronized.

Returns

- **true** Indicates that *value* is successfully populated by the last download time of the publication specified by *publication-mask*.
- **false** Indicates that *publication-mask* specifies more than one publication or that the publication is undefined. If the return value is false, the contents of *value* are not meaningful.

Examples

The following call populates the **dt** structure with the date and time that publication UL_PUB_PUB1 was downloaded:

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

The following call populates the **dt** structure with the date and time that the entire database was last downloaded. It uses the special UL_SYNC_ALL publication mask.

```
ret = ULGetLastDownloadTime( &sqlca, UL_SYNC_ALL, &dt );
```

See also

"UL_SYNC_ALL macro" [*UltraLite Database User's Guide*, page 217]

"UL_SYNC_ALL_PUBS macro" [*UltraLite Database User's Guide*, page 217]

ULGetSynchResult function

Prototype

ul_bool **ULGetSynchResult(** ul_synch_result * *synch-result* **);**

Description

Stores the results of the most recent synchronization, so that appropriate action can be taken in the application:

The application must allocate a **ul_synch_result** object before passing it to **ULGetSynchResult**. The function fills the **ul_synch_result** with the result of the last synchronization. These results are stored persistently in the database.

The function is of particular use when synchronizing applications on the Palm Computing Platform using HotSync, as the synchronization takes place outside the application itself. The SQLCODE value set in the call to **ULPalmLaunch** reflects the **ULPalmLaunch** operation itself. The synchronization status and results are written to the HotSync log only. To obtain extended synchronization result information, call **ULGetSynchResult** after a successful **ULPalmLaunch**.

Parameters

synch-result A structure to hold the synchronization result. It is defined in *ulglobal.h* as follows:.

```
typedef struct {
an_sql_code sql_code;
ul_stream_error stream_error;
ul_bool  upload_ok;
ul_bool  ignored_rows;
ul_auth_status auth_status;
ul_s_long  auth_value;
SQLDATETIME timestamp;
ul_synch_status status;
} ul_synch_result, * p_ul_synch_result;
```

where the individual members have the following meanings:

- **sql_code** The SQL code from the last synchronization. For a list of SQL codes, see “Error messages indexed by Adaptive Server Anywhere SQLCODE” [*ASA Error Messages*, page 2].
- **stream_error** The communication stream error code from the last synchronization. For a listing, see “Database Error Messages” [*ASA Error Messages*, page 1].
- **upload_ok** Set to **true** if the upload was successful; **false** otherwise.
- **ignored_rows** Set to **true** if uploaded rows were ignored; **false** otherwise.

- **auth_status** The synchronization authentication status. For more information, see "auth_status parameter" on page 139.
- **auth_value** The value used by the MobiLink synchronization server to determine the **auth_status** result. For more information, see "auth_value synchronization parameter" on page 140.
- **timestamp** The time and date of the last synchronization.
- **status** The status information used by the observer function. For more information, see "observer synchronization parameter" on page 142.

Returns

The function returns a Boolean value.

true Success.

false Failure.

Examples

The following code checks for success of the previous synchronization.

```
ul_synch_result synch_result;
memset( &synch_result, 0, sizeof( ul_synch_result ) );
db_init( &sqlca );
EXEC SQL CONNECT "dba" IDENTIFIED BY "sql";
if( !ULGetSynchResult( &sqlca, &synch_result ) ) {
    prMsg( "ULGetSynchResult failed" );
}
```

See also

"ULPalmLaunch function" on page 125

ULGlobalAutoincUsage function

Prototype	short **ULGlobalAutoincUsage(** SQLCA * *sqlca* **);**
Description	Obtains the percent of the default values used in all the columns having global autoincrement defaults. 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.
Returns	The function returns a value of type short in the range 0–100.
See also	"ULSetDatabaseID function" on page 131

ULGrantConnectTo function

Prototype

void **ULGrantConnectTo(**
SQLCA * *sqlca*,
ul_char * *userid*,
ul_char * *password* **);**

Description

Grant access to an UltraLite database for a user ID with a specified password. If an existing user ID is specified, this function updates the password for the user.

Parameters

sqlca A pointer to the SQLCA.

userid Character array holding the user ID. The maximum length is 16 characters.

password Character array holding the password for *userid.* The maximum length is 16 characters.

See also

"User authentication" [*UltraLite Database User's Guide*, page 38]

"Adding user authentication to your application" on page 52

"ULRevokeConnectFrom function" on page 129

ULHTTPSStream function

Prototype

ul_stream_defn **ULHTTPSStream(** void **);**

Description

Defines an UltraLite HTTPS stream suitable for synchronization via HTTP.

The HTTPS stream uses TCP/IP as its underlying transport. UltraLite applications act as Web browsers and MobiLink acts as a web server.

See also

"ULSynchronize function" on page 136

"Synchronize method" [*UltraLite Static C++ User's Guide*, page 87]

"stream synchronization parameter" on page 146

"HTTPS stream parameters" [*UltraLite Database User's Guide*, page 186]

ULHTTPStream function

Prototype

ul_stream_defn **ULHTTPStream(** void **);**

Description

Defines an UltraLite HTTP stream suitable for synchronization via HTTP.

The HTTP stream uses TCP/IP as its underlying transport. UltraLite applications act as Web browsers and MobiLink acts as a web server. UltraLite applications send POST requests to send data to the server and GET requests to read data from the server.

See also

"ULSynchronize function" on page 136

"Synchronize method" [*UltraLite Static C++ User's Guide*, page 87]

"stream synchronization parameter" on page 146

"HTTP stream parameters" [*UltraLite Database User's Guide*, page 184]

ULIsSynchronizeMessage function

Prototype

ul_bool **ULIsSynchronizeMessage(** ul_u_long *uMsg* **);**

Description

On Windows CE, this function checks a message to see if it is a synchronization message from the MobiLink provider for ActiveSync, so that code to handle such a message can be called.

This function should be included in the **WindowProc** function of your application.

Example

The following code snippet illustrates how to use ULIsSynchronizeMessage to handle a synchronization message.

```
LRESULT CALLBACK WindowProc( HWND hwnd,
         UINT uMsg,
         WPARAM wParam,
         LPARAM lParam )
{
  if( ULIsSynchronizeMessage( uMsg ) ) {
    // execute synchronization code
    if( wParam == 1 ) DestroyWindow( hWnd );
    return 0;
  }

  switch( uMsg ) {

  // code to handle other windows messages

  default:
    return DefWindowProc( hwnd, uMsg, wParam, lParam );
  }
  return 0;
}
```

See also

"Adding ActiveSync synchronization to your application" on page 96

ULPalmDBStream function (deprecated)

Prototype

ul_stream_defn **ULPalmDBStream(** void **);**

Description

Defines a stream under the Palm Computing Platform suitable for HotSync and Scout Sync.

This function is deprecated. The **stream** parameter is not needed for HotSync synchronization, and may be UL_NULL.

See also

"ULPalmExit function" on page 124

"ULPalmLaunch function" on page 125

"HotSync synchronization stream parameters" [*UltraLite Database User's Guide*, page 181]

"Synchronize method" [*UltraLite Static C++ User's Guide*, page 87]

ULPalmExit function

Prototype

ul_bool **ULPalmExit(** SQLCA * *sqlca*, ul_synch_info * *synch_info* **);**

Description

Saves application state for UltraLite applications on the Palm Computing Platform, and writes out an upload stream for HotSync synchronization. This function is required by all UltraLite Palm applications.

Call this function just before your application is closed, to save the state of the application.

This function saves the application state when the application is deactivated. For applications using HotSync or Scout Sync synchronization, it carries out the additional task of writing an upload stream. When the user uses HotSync or Scout Sync to synchronize data between their Palm device and a PC, the upload stream is read by the MobiLink HotSync conduit or the MobiLink Scout conduit respectively.

The MobiLink HotSync conduit synchronizes with the MobiLink synchronization server through a TCP/IP or HTTP stream using stream parameters. Specify the stream and stream parameters in **synch_info.stream_parms**. Alternatively, you may specify the stream and stream parameters via the *ClientParms* registry entry. If the *ClientParms* registry entry does not exist, a default setting of `{stream=tcpip;host=localhost}` is used.

Parameters

sqlca A pointer to the SQLCA.

synch_info A synchronization structure.

If you are using TCP/IP or HTTP synchronization, supply UL_NULL instead of the ul_synch_info structure. When using these streams, the synchronization information is supplied instead in the call to **ULSynchronize**.

If you use HotSync or Scout Sync synchronization, supply the synchronization structure. The value of the **stream** parameter is ignored, and may be UL_NULL.

☞ For information on the members of the *synch_info* structure, see "Synchronization Parameters Reference" on page 137.

Returns

The function returns a Boolean value.

true Success.

false Failure.

ULPalmLaunch function

Prototype

UL_PALM_LAUNCH_RET **ULPalmLaunch(**
SQLCA * *sqlca*,
ul_synch_info * *synch_info* **);**

typedef enum **{**
LAUNCH_SUCCESS_FIRST,
LAUNCH_SUCCESS,
LAUNCH_FAIL
} UL_PALM_LAUNCH_RET**;**

Parameters

sqlca A pointer to the SQLCA.

synch_info A synchronization structure. For information on the members of this structure, see "Synchronization parameters" on page 138.

If you are using TCP/IP or HTTP synchronization, supply UL_NULL as *synch_info*.

Description

This function restores application state for UltraLite applications on the Palm Computing Platform. This function is required by all UltraLite Palm applications.

Your application must call **ULEnablePalmDB** or **ULEnableFileDB** before calling **ULPalmLaunch**.

All UltraLite Palm applications need to use this function to handle the launch code in your application's **PilotMain**.

This function restores the application state when the application is activated. For applications using HotSync or Scout Sync synchronization, it carries out the additional task of processing the download stream prepared by the MobiLink HotSync conduit or MobiLink Scout conduit.

If you are using TCP/IP or HTTP synchronization, supply a null value for the stream parameter in the **ul_synch_info** synchronization structure. This information is supplied instead in the call to **ULSynchronize**.

Returns

A member of the **UL_PALM_LAUNCH_RET** enumeration. The return values have the following meanings:

- **LAUNCH_SUCCESS_FIRST** This value is returned the first time the application is successfully launched and at any subsequent time the internal state of the UltraLite database needs to be re-established. In general, the state of the database needs to be re-established only after severe failures.

  In embedded SQL applications you should call **db_init** immediately after this return code is detected; in C++ API applications, you should open a

database object.

- **LAUNCH_SUCCESS** This value is returned when an application is successfully launched, after the Palm user has been using other applications.
- **LAUNCH_FAIL** This value is returned when the launch fails.

Examples

A typical embedded SQL example is

```
ULEnablePalmRecordDB( &sqlca );
switch( ULPalmLaunch( &sqlca, &synch_info ) ){
case LAUNCH_SUCCESS_FIRST:
   if( !db_init( &sqlca ) ){
      // db_init failed: add error handling here
      break;
   }
   // fall through
case LAUNCH_SUCCESS:
   // do work here
   break;
case LAUNCH_FAIL:
   // error
   break;
}
```

See also

“Launching an UltraLite Palm application” on page 77

“ULEnableFileDB function” on page 110

“ULEnablePalmRecordDB function” on page 112

ULResetLastDownloadTime function

Prototype

void **ULResetLastDownloadTime(**
SQLCA * *sqlca*,
ul_publication_mask *publication-mask* **);**

Description

This function can be used to repopulate values and return an application to a known clean state. It resets the last download time so that the application resynchronizes previously downloaded data.

Parameters

sqlca A pointer to the SQLCA.

publication-mask A set of publications to check. A value of 0 corresponds to the entire database. The set is supplied as a mask. For example, the following mask corresponds to publications PUB1 and PUB2.:

```
UL_PUB_PUB1 | UL_PUB_PUB2
```

☞ For more information on publication masks, see "publication synchronization parameter" on page 144.

Example

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

```
ULResetLastDownloadTime( &sqlca, UL_SYNC_ALL );
```

See also

"ULGetLastDownloadTime function" on page 115

"Timestamp-based synchronization" [*MobiLink Synchronization User's Guide*, page 72]

ULRetrieveEncryptionKey function

Prototype

ul_bool **ULRetrieveEncryptionKey(**
ul_char * *key*,
ul_u_short *len*,
ul_u_long * *creator*,
ul_u_long * *feature-num* **);**

Description

On the Palm Computing Platform the encryption key is saved in dynamic memory as a Palm **feature**. Features are indexed by creator and a feature number.

This function retrieves the encryption key from memory.

Parameters

key A pointer to a buffer in which to hold the retrieved encryption key.

len The length of the buffer that holds the encryption key with a terminating null character.

creator A pointer to the creator ID of the feature holding the encryption key. A value of NULL is the default.

feature-num A pointer to the feature number holding the encryption key. A value of NULL uses the UltraLite default, which is feature number 100.

Returns

- **true** if the operation is successful.
- **false** if the operation is unsuccessful. This occurs if the feature was not found or if the supplied buffer length is insufficient to hold the key plus a terminating null character.

See also

"ULClearEncryptionKey function" on page 107

"ULSaveEncryptionKey function" on page 130

"Using the encryption key on the Palm Computing Platform" on page 59

ULRevokeConnectFrom function

Prototype
void **ULRevokeConnectFrom(** SQLCA * *sqlca***,** ul_char * *userid* **);**

Description
Revoke access from an UltraLite database for a user ID.

Parameters
sqlca A pointer to the SQLCA.

userid Character array holding the user ID to be excluded from database access. The maximum length is 16 characters.

See also
"User authentication" [*UltraLite Database User's Guide*, page 38]

"Adding user authentication to your application" on page 52

"ULGrantConnectTo function" on page 119

ULSaveEncryptionKey function

Prototype

ul_bool **ULSaveEncryptionKey(**
ul_char * *key*,
ul_u_long * *creator*,
ul_u_long * *feature-num* **);**

Description

On the Palm Computing Platform the encryption key is saved in dynamic memory as a Palm **feature**. Features are indexed by creator and a feature number. They are not backed up and are cleared on any reset of the device.

This function saves the encryption key in Palm dynamic memory.

Parameters

key A pointer to the encryption key.

creator A pointer to the creator ID of the feature holding the encryption key. A value of NULL is the default.

feature-num A pointer to the feature number holding the encryption key. A value of NULL uses the UltraLite default, which is feature number 100.

Returns

- **true** if the operation is successful.
- **false** if the operation is unsuccessful.

See also

"ULClearEncryptionKey function" on page 107

"ULRetrieveEncryptionKey function" on page 128

"Using the encryption key on the Palm Computing Platform" on page 59

ULSetDatabaseID function

Prototype	void **ULSetDatabaseID(** SQLCA * *sqlca***,** ul_u_long *id* **);**
Description	Sets the database identification number.
Parameters	**sqlca** A pointer to the SQLCA. **id** A positive integer that uniquely identifies a particular database in a replication or synchronization setup.
See also	"ULGlobalAutoincUsage function" on page 118

ULSocketStream function

Prototype	ul_stream_defn **ULSocketStream(** void **);**
Description	Defines an UltraLite socket stream suitable for synchronization via TCP/IP.
See also	“ULSynchronize function” on page 136
	“Synchronize method” [*UltraLite Static C++ User’s Guide*, page 87]

ULStoreDefragFini function

Prototype

ul_ret_void **ULStoreDefragFini(**
SQLCA * *sqlca*,
p_ul_store_defrag_info *dfi* **);**

Description

This function disposes of the defragmentation information block returned by **ULStoreDefragInit**.

Parameters

sqlca A pointer to the SQLCA.

dfi A defragmentation information block.

See also

"Defragmenting UltraLite databases" on page 60

"ULStoreDefragInit function" on page 134

ULStoreDefragInit function

Prototype	p_ul_store_defrag_info **ULStoreDefragInit(** SQLCA * *sqlca* **);**
Description	This function initializes and returns a defragmentation information block to maintain the defragmentation state of the database.
Parameters	**sqlca** A pointer to the SQLCA.
Returns	If successful, returns a defragmentation information block **p_ul_store_defrag_info**. If unsuccessful, for example if there is not enough memory, returns **UL_NULL**.
See also	"Defragmenting UltraLite databases" on page 60 "ULStoreDefragFini function" on page 133

ULStoreDefragStep function

Prototype	ul_bool **ULStoreDefragStep(** SQLCA * *sqlca* p_ul_store_defrag_info *dfi* **);**
Description	This function defragments a piece of the database.
Parameters	**sqlca** A pointer to the SQLCA. **dfi** A defragmentation information block.
Returns	If the entire store has been defragmented, returns **ul_true**. If the entire store is not defragmented, returns **ul_false**. If an error occurs, SQLCODE is set.
See also	“Defragmenting UltraLite databases” on page 60 “ULStoreDefragFini function” on page 133 “ULStoreDefragInit function” on page 134

ULSynchronize function

Prototype

void **ULSynchronize(**
SQLCA * *sqlca*,
ul_synch_info * *synch_info* **);**

Description

Initiates synchronization in an UltraLite application.

For TCP/IP or HTTP synchronization, the **ULSynchronize** function initiates synchronization. Errors during synchronization that are not handled by the **handle_error** script are reported as SQL errors. Your application should test the SQLCODE return value of this function.

Parameters

sqlca A pointer to the SQLCA.

synch_info A synchronization structure. For information on the members of this structure, see “Synchronization parameters” on page 138.

See also

“MobiLink Synchronization Server Options” [*MobiLink Synchronization Reference*, page 3]

“START SYNCHRONIZATION DELETE statement [MobiLink]” [*MobiLink Synchronization Reference*, page 258]

CHAPTER 9

Synchronization Parameters Reference

About this chapter

This chapter provides reference information about synchronization parameters.

Contents

Synchronization parameters

The synchronization parameters are members of a structure that is provided as an argument in the call to synchronize. The **ul_synch_info** structure that holds the synchronization parameters is defined in *ulglobal.h* as follows:

```
struct ul_synch_info {
    ul_char *             user_name;
    ul_char *             password;
    ul_char *             new_password;
    ul_char *             version;
    p_ul_stream_defn      stream;
    ul_char *             stream_parms;
    p_ul_stream_defn      security;
    ul_char *             security_parms;
    ul_synch_observer_fn  observer;
    ul_void *             user_data;
    ul_publication_mask   publication;
    ul_bool               upload_only;
    ul_bool               download_only;
    ul_bool               send_download_ack;
    ul_bool               send_column_names;
    ul_bool               ping;
    ul_bool               checkpoint_store;
    ul_bool               disable_concurrency;
    ul_byte               num_auth_params;
    ul_char * *           auth_parms;

    // fields set on output
    ul_stream_error       stream_error;
    ul_bool               upload_ok;
    ul_bool               ignored_rows;
    ul_auth_status        auth_status;
    ul_s_long             auth_value;

    p_ul_synch_info       init_verify;
};
```

The **init_verify** field is reserved for internal use.

Use UL_TEXT around constant strings

The **UL_TEXT** macro allows constant strings to be compiled as single-byte strings or wide-character strings. Use this macro to enclose all constant strings supplied as members of the **ul_synch_info** structure so that the compiler handles these parameters correctly.

☞ For a description of the role of each synchronization parameter, see "Synchronization parameters" [*UltraLite Database User's Guide*, page 162].

auth_parms parameter

Function — Provides parameters to a custom user authentication script.

Usage — Set the parameters as follows:

```
ul_char * Params[ 3 ] = { UL_TEXT( "parm1" ),
                          UL_TEXT( "parm2" ),
                          UL_TEXT( "parm3" ) };
// ...
info.num_auth_parms = 3;
info.auth_parms = Params;
```

See also — “num_auth_parms parameter” on page 142

“authenticate_parameters connection event” [*MobiLink Synchronization Reference*, page 98]

“authenticate_user connection event” [*MobiLink Synchronization Reference*, page 100]

auth_status parameter

Function — Reports the status of MobiLink user authentication.

Usage — Access the parameter as follows:

```
ul_synch_info info;
// ...
returncode = info.auth_status;
```

Allowed values — After synchronization, the parameter must hold one of the following values. If a custom **authenticate_user** synchronization script at the consolidated database returns a different value, the value is interpreted according to the rules given in “authenticate_user connection event” [*MobiLink Synchronization Reference*, page 100].

Constant	Value	Description
UL_AUTH_STATUS_-UNKNOWN	0	Authorization status is unknown, possibly because the connection has not yet synchronized.
UL_AUTH_STATUS_VALID	1000	User ID and password were valid at the time of synchronization.
UL_AUTH_STATUS_VALID_-BUT_EXPIRES_SOON	2000	User ID and password were valid at thetime of synchronization but will expire soon.
UL_AUTH_STATUS_EXPIRED	3000	Authorization failed: user ID or password have expired.
UL_AUTH_STATUS_INVALID	4000	Authorization failed: bad user ID or password.
UL_AUTH_STATUS_IN_USE	5000	Authorization failed: user ID is already in use.

See also

“Authenticating MobiLink Users” [*MobiLink Synchronization User’s Guide*, page 103].

auth_value synchronization parameter

Function

Reports return values from custom user authentication synchronization scripts.

Default

The values set by the default MobiLink user authentication mechanism are described in “auth_status synchronization parameter” on page 139.

Usage

The parameter is read-only.

Access the parameter as follows:

```
ul_synch_info info;
// ...
returncode = info.auth_value;
```

See also

“authenticate_user connection event” [*MobiLink Synchronization Reference*, page 100]

“authenticate_user_hashed connection event” [*MobiLink Synchronization Reference*, page 104]

“auth_status synchronization parameter” on page 139

checkpoint_store synchronization parameter

Function	Adds additional checkpoints of the database during synchronization to limit database growth during the synchronization process.
Default	By default, limited checkpointing is done.

Usage	Set the parameter as follows:
	`ul_synch_info info;`
	`// ...`
	`info.checkpoint_store = ul_true ;`

disable_concurrency synchronization parameter

Function	Disallow database access from other threads during synchronization.
Default	By default, data access is available. Data access is read-write during the download phase, and read-only otherwise.

Usage	Set the parameter as follows:
	`ul_synch_info info;`
	`// ...`
	`info.disable_concurrency = ul_false ;`

See also	"Threading in UltraLite applications" [*UltraLite Database User's Guide*, page 47]

download_only synchronization parameter

Function	Do not upload any changes from the UltraLite database during this synchronization.
Default	The parameter is an optional Boolean value, and by default is false.

Usage	Set the parameter as follows:
	`ul_synch_info info;`
	`// ...`
	`info.download_only = ul_true;`

See also	"Including read-only tables in an UltraLite database" on page **??**. "upload_only synchronization parameter" on page 150

ignored_rows synchronization parameter

Function	Reports if any rows were ignored by the MobiLink synchronization server during synchronization because of absent scripts.

The parameter is read-only.

Access methods

new_password synchronization parameter

Function — Sets a new MobiLink password associated with the user name.

Default — There is no default.

Usage — Set the parameter as follows:

```
ul_synch_info info;
// ...
info.password = UL_TEXT( "myoldpassword" );
info.new_password = UL_TEXT( "mynewpassword" );
```

See also — "Authenticating MobiLink Users" [*MobiLink Synchronization User's Guide*, page 103].

num_auth_parms parameter

Function — The number of authentication parameter strings passed to a custom authentication script.

Default — No parameters passed to a custom authentication script.

Usage — The parameter is used together with auth_parms to supply information to custom authentication scripts.

☞ For more information, see "auth_parms parameter" on page 139.

See also — "auth_parms parameter" on page 139

"authenticate_parameters connection event" [*MobiLink Synchronization Reference*, page 98]

"authenticate_user connection event" [*MobiLink Synchronization Reference*, page 100]

observer synchronization parameter

Function — A pointer to a callback function that monitors synchronization.

See also — "Monitoring and canceling synchronization" on page 65

"user_data synchronization parameter" on page 150

password synchronization parameter

Function — A string specifying the MobiLink password associated with the **user_name**.

This user name and password are separate from any database user ID and password, and serves to identify and authenticate the application to the MobiLink synchronization server.

Default: There is no default.

Usage: Set the parameter as follows:

```
ul_synch_info info;
// ...
info.password = UL_TEXT( "mypassword" );
```

See also: "Authenticating MobiLink Users" [*MobiLink Synchronization User's Guide*, page 103].

ping synchronization parameter

Function: Confirm communications between the UltraLite client and the MobiLink synchronization server. When this parameter is set to true, no synchronization takes place.

When the MobiLink synchronization server receives a ping request, it connects to the consolidated database, authenticates the user, and then sends the authenticating user status and value back to the client.

If the ping succeeds, the MobiLink server issues an information message. If the ping does not succeed, it issues an error message.

If the MobiLink user name cannot be found in the ml_user system table and the MobiLink server is running with the command line option -zu+, the MobiLink server adds the user to ml_user.

The MobiLink synchronization server may execute the following scripts, if they exist, for a ping request:

- begin_connection
- authenticate_user
- authenticate_user_hashed
- end_connection

Default: The parameter is optional, and is a boolean.

Usage: Set the parameter as follows:

```
ul_synch_info info;
// ...
info.ping = ul_true;
```

See also: "-pi option" [*MobiLink Synchronization Reference*, page 76]

publication synchronization parameter

Function — Specifies the publications to be synchronized.

Default — If you do not specify a publication, all data is synchronized.

Usage — The UltraLite generator identifies the publications specified on the *ulgen* `-v` command line option as upper case constants with the name UL_PUB_pubname, where pubname is the name given to the -v option.

For example, the following command line generates a publication identified by the constant UL_PUB_SALES:

```
ulgen -v sales ...
```

When synchronizing, set the publication parameter to a **publication mask**: an OR'd list of publication constants. For example:

```
ul_synch_info info;
// ...
info.publication = UL_PUB_MYPUB1 | UL_PUB_MYPUB2 ;
```

The special publication mask **UL_SYNC_ALL** describes all the tables in the database, whether in a publication or not. The mask **UL_SYNC_ALL_PUBS** describes all tables in publications in the database.

See also — "The UltraLite generator" on page **??**

"Designing sets of data to synchronize separately" on page **??**

security synchronization parameter

Function — Set the UltraLite client to use Certicom encryption technology when exchanging messages with the MobiLink synchronization server.

> **Separately-licensable option required**
> Use of Certicom technology requires that you obtain the separately-licensable SQL Anywhere Studio security option and is subject to export regulations. For more information on this option, see "Welcome to SQL Anywhere Studio" [*Introducing SQL Anywhere Studio*, page 4].

Default — The Security parameter is null by default, corresponding to no transport-layer security.

Usage — The security stream is specified in addition to the synchronization stream. Allowed values are as follows:

- **ULSecureCerticomTLSStream()** Elliptic-curve transport-layer security provided by Certicom.

- **ULSecureRSATLSStream()** RSA transport-layer security provided by Certicom.

```
ul_synch_info info;
...
info.stream = ULSocketStream();
info.security = ULRSATLSStream();
```

See also "Transport-Layer Security" [*MobiLink Synchronization User's Guide*, page 337].

security_parms synchronization parameter

Function Sets the parameters required when using transport-layer security. This parameter must be used together with the **security** parameter.

☞ For more information, see "security synchronization parameter" on page 144.

Usage The ULSecureCerticomTLSStream() and ULSecureRSATLSStream() security parameters take a string composed of the following optional parameters, supplied in an semicolon-separated string.

- **certificate_company** The UltraLite application only accepts server certificates when the organization field on the certificate matches this value. By default, this field is not checked.
- **certificate_unit** The UltraLite application only accepts server certificates when the organization unit field on the certificate matches this value. By default, this field is not checked.
- **certificate_name** The UltraLite application only accepts server certificates when the common name field on the certificate matches this value. By default, this field is not checked.

For example:

```
ul_synch_info info;
...
info.stream = ULSocketStream();
info.security = ULSecureCerticomTLSStream();
info.security_parms =
     UL_TEXT( "certificate_company=Sybase" )
     UL_TEXT( ";" )
     UL_TEXT( "certificate_unit=Sales" );
```

The **security_parms** parameter is a string, and by default is null.

If you use secure synchronization, you must also use the `-r` command-line option on the UltraLite generator. For more information, see "The UltraLite generator" on page **??**.

send_column_names synchronization parameter

Function

When **send_column_names** is set to **ul_true** UltraLite sends each column name to the MobiLink synchronization server. By default UltraLite does not send column names.

This parameter is typically used together with the `-za` or `-ze` switch on the MobiLink synchronization server for automatically generating synchronization scripts.

See also

"-za option" [*MobiLink Synchronization Reference*, page 28]

send_download_ack synchronization parameter

Function

Set this boolean parameter to **false** to instruct the MobiLink synchronization server that the client will not provide a download acknowledgement.

If the client does send download acknowledgement, the MobiLink synchronization server worker thread must wait for the client to apply the download. If the client does not sent a download acknowledgement, the MobiLink synchronization server is freed up sooner for its next synchronization.

stream synchronization parameter

Function

Set the MobiLink synchronization stream to use for synchronization.

☞ For more information, see "stream_parms synchronization parameter" on page 149.

Default

The parameter has no default value, and must be explicitly set.

Usage

```
ul_synch_info info;
...
info.stream = ULSocketStream();
```

When the type of stream requires a parameter, pass that parameter using the **stream_parms** parameter; otherwise, set the **stream_parms** parameter to null.

The following stream functions are available, but not all are available on all target platforms:

Stream	Description
ULActiveSyncStream()	ActiveSync synchronization (Windows CE only). ☞ For a list of stream parameters, see "ActiveSync parameters" [*UltraLite Database User's Guide*, page 179].
ULHTTPStream()	Synchronize via HTTP. The HTTP stream uses TCP/IP as its underlying transport. UltraLite applications act as Web browsers and the MobiLink synchronization server acts as a Web server. UltraLite applications send POST requests to send data to the server and GET requests to read data from the server. ☞ For a list of stream parameters, see "HTTP stream parameters" [*UltraLite Database User's Guide*, page 184].
ULHTTPSStream()	Synchronize via the HTTPS synchronization stream. The HTTPS stream uses SSL or TLS as its underlying protocol. It operates over Internet protocols (HTTP and TCP/IP). The HTTPS stream requires the use of technology supplied by Certicom. Use of Certicom technology requires that you obtain the separately-licensable SQL Anywhere Studio security option and is subject to export regulations. For more information on this option, see "Welcome to SQL Anywhere Studio" [*Introducing SQL Anywhere Studio*, page 4]. ☞ For a list of stream parameters, see "HTTPS stream parameters" [*UltraLite Database User's Guide*, page 186].
ULSocketStream()	Synchronize via TCP/IP. ☞ For a list of stream parameters, see "TCP/IP stream parameters" [*UltraLite Database User's Guide*, page 182].

stream_error synchronization parameter

Function Sets a structure to hold communications error reporting information.

Default

The parameter has no default value, and must be explicitly set.

Description

The **stream_error** field is a structure of type **ul_stream_error**.

```
typedef struct ss_error {
    ss_stream_id         stream_id;
    ss_stream_context    stream_context;
    ss_error_code        stream_error_code;
    asa_uint32           system_error_code;
    rp_char              *error_string;
    asa_uint32           error_string_length;
} ss_error, *p_ss_error;
```

The structure is defined in *sserror.h*, in the *h* subdirectory of your SQL Anywhere directory.

The **ul_stream_error** fields are as follows:

- **stream_id** The network layer reporting the error. This enumeration has the following constants:

  ```
  STREAM_ID_TCPIP
  STREAM_ID_HTTP
  STREAM_ID_CERTICOM_TLS
  STREAM_ID_PALM_CONDUIT
  STREAM_ID_ACTIVESYNC
  ```

- **stream_context** The basic network operation being performed, such as open, read, or write. For details, see *sserror.h*.

- **stream_error_code** The error reported by the stream itself. The **stream_error_code** is of type **ss_error_code**. The stream error codes are all prefixed with STREAM_ERROR_. A write error, for example, is STREAM_ERROR_WRITE.

  ☞ For a listing of error numbers, see "MobiLink Communication Error Messages" [*MobiLink Synchronization Reference*, page 347]. For the error code suffixes, see *sserror.h*.

  In this version, to find the constant associated with each number you must count down the number of lines prefixed by DO_STREAM_Error in *sserror.h*. For example, to find the constant for error number 10, you use the tenth DO_STREAM_ERROR entry in *sserror.h*, which is as follows:

  ```
  DO_STREAM_ERROR( WRITE )
  ```

  The constant associated with this error is therefore STREAM_ERROR_WRITE.

- **stream_error** The network operation being performed (the context) and the error itself as an enumeration constant.

- **system_error_code** A system-specific error code.

- **error_string** An application-provided error message

Usage

Check for SQLE_COMMUNICATIONS_ERROR as follows:

```
ul_char  error_buff[ 100 ];
ul_synch_info info;
...
ULInitSynchInfo( &info );
info.stream_error.error_string = error_buff;
info.stream_error.error_string_length =
                    sizeof( error_buff );
...
ULSynchronize( &sqlca, &info )
if( SQLCODE == SQLE_COMMUNICATIONS_ERROR ){
    printf( error_buff );
...// more error handling here
```

stream_parms synchronization parameter

Function

Sets parameters to configure the synchronization stream.

A semi-colon separated list of parameter assignments. Each assignment is of the form *keyword=value*, where the allowed sets of keywords depends on the synchronization stream.

For a list of available parameters for each stream, see the following sections:

- "ActiveSync parameters" [*UltraLite Database User's Guide*, page 179]
- "HotSync parameters" [*UltraLite Database User's Guide*, page 181]
- "HTTP stream parameters" [*UltraLite Database User's Guide*, page 184]
- "HTTPS stream parameters" [*UltraLite Database User's Guide*, page 186]
- "TCP/IP stream parameters" [*UltraLite Database User's Guide*, page 182]

Default

The parameter is optional, is a string, and by default is null.

Usage

Set the parameter as follows:

```
ul_synch_info info;
// ...
info.stream_parms= UL_TEXT( "host=myserver;port=2439" );
```

See also

"Synchronization stream parameters" on page **??**.

upload_ok synchronization parameter

Function

Reports the status of MobiLink uploads. The MobiLink synchronization server provides this information to the client.

The parameter is read-only.

Usage

After synchronization, the **upload_ok** parameter holds **true** if the upload was successful, and **false** otherwise.

Access the parameter as follows:

```
ul_synch_info info;
// ...
returncode = info.upload_ok;
```

upload_only synchronization parameter

Function

Indicates that there should be no downloads in the current synchronization, which can save communication time, especially over slow communication links. When set to true, the client waits for the upload acknowledgement from the MobiLink synchronization server, after which it terminates the synchronization session successfully.

Default

The parameter is an optional Boolean value, and by default is false.

Usage

Set the parameter to true as follows:

```
ul_synch_info info;
// ...
info.upload_only = ul_true;
```

See also

“Synchronizing high-priority changes” on page **??**

“download_only synchronization parameter” on page 141

user_data synchronization parameter

Function

Make application-specific information available to the synchronization observer.

Usage

When implementing the synchronization observer callback function **observer**, you can make application-specific information available by providing information using **user_data**.

See also

“observer synchronization parameter” on page 142

“Monitoring and canceling synchronization” on page **??**

user_name synchronization parameter

Function

A string specifying the user name that uniquely identifies the MobiLink client to the MobiLink synchronization server. MobiLink uses this value to determine the download content, to record the synchronization state, and to recover from interruptions during synchronization.

Default — The parameter is required, and is a string.

Usage — Set the parameter as follows:

```
ul_synch_info info;
// ...
info.user_name= UL_TEXT( "mluser" );
```

See also — “Authenticating MobiLink Users” [*MobiLink Synchronization User’s Guide*, page 103].

“The MobiLink user” [*MobiLink Synchronization User’s Guide*, page 20].

version synchronization parameter

Function — Each synchronization script in the consolidated database is marked with a version string. For example, there may be two different **download_cursor** scripts, identified by different version strings. The version string allows an UltraLite application to choose from a set of synchronization scripts.

Default — The parameter is a string, and by default is the MobiLink default version string.

Usage — Set the parameter as follows:

```
ul_synch_info info;
// ...
info.version = UL_TEXT( "default" );
```

See also — “Script versions” [*MobiLink Synchronization User’s Guide*, page 49].

Index

Symbols

A

B

C

D

E

F

G

H

I

L

M

N

O

P

Q

R

S

T

U

V

W