iAnywhere™ SOLUTIONS A SYBASE COMPANY

UltraLite™ Database User's Guide

Part number: 37121-01-0900-01

Last modified: June 2003

Contents

About This Manual

Subject	This manual introduces the UltraLite database system for small devices.
Audience	This manual is intended for all developers who wish to take advantage of the performance, resource efficiency, robustness, and security of an UltraLite relational database for data storage and synchronization.

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.

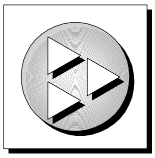
  [**QUOTES** { **ON** | **OFF** }]

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

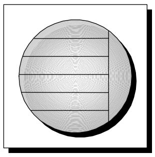
Graphic icons

The following icons are used in this documentation.

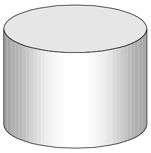
- A client application.



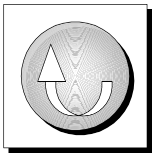
- A database server, such as Sybase Adaptive Server Anywhere.



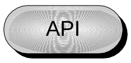
- A database. In some high-level diagrams, the icon may be used to represent both the database and the database server that manages it.



- 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.



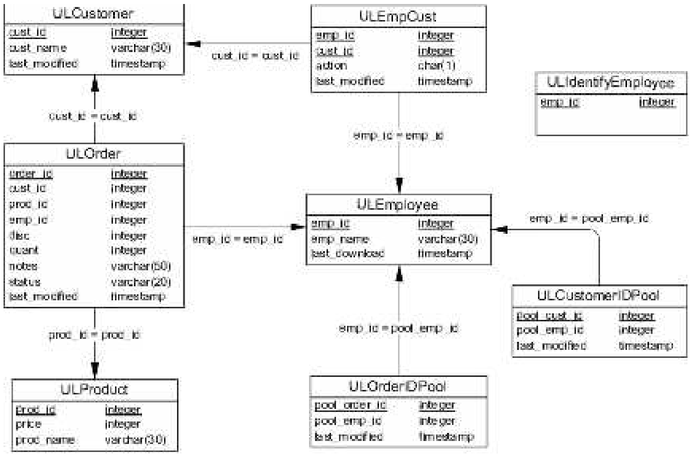
The CustDB sample database

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.

PART I

ULTRALITE DATABASES

This part introduces the UltraLite relational database system for handheld devices and describes general features of the UltraLite database..

CHAPTER 1

Welcome to UltraLite

About this chapter

This chapter introduces you to UltraLite features, platforms, architecture, and functionality.

Contents

Introduction

UltraLite is a relational database and synchronization technology for small, mobile, and embedded devices. You can build UltraLite applications using a variety of interfaces. Each interface uses the same underlying database engine.

- ♦ **UltraLite components** UltraLite components provide users of rapid application development with database and synchronization features. They provide a familiar interface for each supported development tool. UltraLite components provide a simple table-based data access interface and also dynamic SQL for more complex queries.

  The following components are available:

  - **UltraLite for AppForge MobileVB** Development using the AppForge MobileVB extension to Microsoft Visual Basic.

    ☞ See “Introduction” [*UltraLite for MobileVB User’s Guide*, page 1].

  - **UltraLite ActiveX** Development using eMbedded Visual Basic or JScript with Pocket IE.

    ☞ See “Introduction” [*UltraLite ActiveX User’s Guide*, page 1].

  - **Native UltraLite for Java** Development using a supported JDK. The UltraLite component itself contains native (C++) methods for improved performance.

    ☞ See “Introduction to Native UltraLite for Java” [*Native UltraLite for Java User’s Guide*, page 1]

  - **UltraLite.NET** Development using Visual Studio .NET.

    ☞ See “Introduction to UltraLite.NET” [*UltraLite.NET User’s Guide*, page 1]

  - **UltraLite for C++** Development using a C++ interface.

    ☞ See “Introduction to UltraLite for C++” [*UltraLite C++ User’s Guide*, page 1]

- ♦ **Static interfaces** Static interfaces provide a rich SQL interface for C/C++ and Java developers comfortable with a preprocessor-based interface. All SQL statements used in the application must be defined at compile time.

  The following static interfaces are available:

  - **UltraLite for Embedded SQL** Development using C/C++ with embedded SQL statements.

    ☞ See “Introduction” [*UltraLite Embedded SQL User’s Guide*, page 1].

  - **UltraLite static C++** Development in C++ using an application-specific generated C++ API.

    ☞ See “Introduction” [*UltraLite Static C++ User’s Guide*, page 1].

- **UltraLite static Java** Development in pure Java using a JDBC interface.

  ☞ See "Introduction" [*UltraLite Static Java User's Guide*, page 6].

UltraLite benefits

UltraLite provides the following benefits to application developers:

♦ **Robust data management** Data held on small devices is as important as data in enterprise databases. UltraLite brings transaction processing, referential integrity, and other benefits of relational databases to small devices.

☞ For more information about UltraLite database features, see "UltraLite databases" on page 5.

♦ **Powerful synchronization** An information system is only as robust as its weakest link. UltraLite gives you the ability to synchronize data with a central database-management system when used with SQL Anywhere Studio.

UltraLite uses MobiLink synchronization technology, included in SQL Anywhere Studio, to synchronize with industry-standard database-management systems. MobiLink synchronization works with ODBC-compliant data sources such as Sybase Adaptive Server Anywhere, Sybase Adaptive Server Enterprise, IBM DB2, Microsoft SQL Server, and Oracle. It provides flexible, programmable, and scalable synchronization that can manage thousands of UltraLite databases.

☞ For more information, see "Synchronization for UltraLite Applications" on page 143.

♦ **Straightforward development** UltraLite components provide an object-based programming interface for straightforward access to data. Integration into popular development tools such as eMbedded Visual Basic, AppForge MobileVB, and Borland JBuilder makes developers productive. A graphical tool enables you to design and modify UltraLite databases rapidly.

♦ **Multi-platform availability** You can develop and deploy UltraLite database applications for Windows CE, Palm OS, and Java-based devices.

☞ For more information, see "System requirements and supported platforms" on page **??**.

UltraLite databases

UltraLite databases are transaction-processing relational databases, and provide you with the following features:

- **Tables** A single UltraLite database can hold many tables. The number and type of columns in a relational database table is fixed at design time, but each table can have any number of rows (up to 64 K). Each row has a single entry for each column. The special NULL entry is used when there is no value for the entry.

  When designing your database, each table should represent a separate type of item, such as Customers, Employees, and so on.

- **Data types** UltraLite databases can manage a full range of data types, as well as default values and NULL values.

- **Indexes** The rows in a relational database table are not ordered. You can create indexes to access the rows in order and to provide fast access to data. Indexes are commonly associated with a single column, but UltraLite also provides multi-column indexes.

- **Keys** Each table has a special index called the **primary key**. Entries in the primary key column or columns must be unique.

  Foreign keys relate the data in one table to that in another. Each entry in the foreign key column must correspond to an entry in the primary key of another table.

  Between them, primary keys and foreign keys ensure that the database has **referential integrity**. Referential integrity is enforced in UltraLite databases, so that you cannot, for example, enter an order for a customer unless that customer exists in the database.

  By enforcing referential integrity, UltraLite ensures that the data in your UltraLite database is correct, in the same manner that data elsewhere in the enterprise is correct.

- **Publications** If you wish to synchronize the data in your UltraLite database with other databases you must have a valid SQL Anywhere Studio license. SQL Anywhere Studio includes MobiLink synchronization technology to synchronize UltraLite databases with desktop, workgroup or enterprise databases.

  Publications define sets of data to be synchronized. It is often desirable to synchronize all the data in an UltraLite database, but publications provide extra flexibility and control. They allow you to perform priority synchronizations, which means you can specify that only certain tables or groups of tables should be synchronized.

- **Transactions and recovery** UltraLite has commit and rollback features, together with automatic recovery in the event of device failure, to guarantee that transactions are executed completely or not at all.

- **Security** UltraLite provides user authentication and database encryption, as well as encryption on the device and during synchronization, to build secure applications.
- **Performance and small footprint** UltraLite target devices tend to have relatively slow processors. UltraLite employs algorithms and data structures that provide high performance and low memory use. For example, UltraLite provides a caching algorithm designed specifically for small devices.
- **Multi-threaded applications** You can develop multi-threaded UltraLite applications on those platforms that support it (Windows and Windows CE).

☞ For more information about UltraLite databases, see "UltraLite Databases" on page 27.

Using the UltraLite documentation

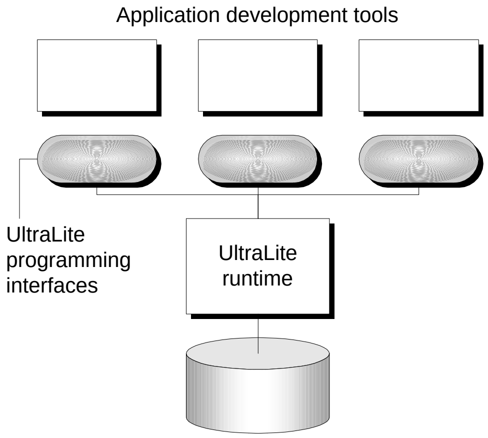
Once you have selected which UltraLite interface to use, you can find the information you need in the following books, all of which are included in the SQL Anywhere online books:

- **UltraLite Database User's Guide (this book)** This book presents information that is useful for all UltraLite interfaces, including information about the UltraLite database management, SQL, and synchronization.
- **Interface book** Each UltraLite interface has a separate book, which contains all you need for developing applications using that interface.
- **MobiLink books** If your application includes synchronization, The *MobiLink Synchronization User's Guide* and the *MobiLink Synchronization Reference* provide a complete guide to the synchronization system.

Developing UltraLite applications

UltraLite application development is carried out using either an UltraLite component or a static interface. An UltraLite component is provided for each development tool. Each component exposes a set of objects for data manipulation in the form of an API tailored to suit the expectations of users of a particular tool or language.

All UltraLite components and the UltraLite runtime library used by the static interfaces are built on the same underlying database engine.


Application development tools
UltraLite programming interfaces
UltraLite runtime

Applications that you create with an UltraLite Component will consist of the following:

- Your application code
- The UltraLite component or runtime library
- An UltraLite database

Developing applications with UltraLite components

To develop an application using an UltraLite component you follow the following basic sequence of steps.

1. Design your database.

   A database **schema** is the database definition, including all tables, indexes, and so on. You create a database schema using the UltraLite Schema Painter or writing an XML file. Users of SQL Anywhere Studio can generate an UltraLite database schema from an Adaptive Server Anywhere database.

UltraLite holds the database schema in a schema file. The UltraLite components use the information in this file to create a database when an application is first run.

☞ For more information on the UltraLite Schema Painter, see "The UltraLite Schema Painter" on page 73.

☞ For more information on the UltraLite utility *ulinit*, see "UltraLite initialization utility" on page 86.

2. Set up your development environment.

   In each component, you are required to develop your application on a specific development platform, and to deploy to a specific target device. To achieve this end, you need to set up your development environment in tandem with the target environment. Tutorials in the companion books show you how you can accomplish this setup.

   ☞ For more information on creating a project architecture for UltraLite ActiveX, see "Adding the UltraLite component to the design environment" [*UltraLite ActiveX User's Guide*, page 60].

   ☞ For more information on creating a project architecture for UltraLite for MobileVB, see the tutorial "Lesson 1: Create a project architecture" [*UltraLite for MobileVB User's Guide*, page 9].

   ☞ For more information on creating a project architecture for Native UltraLite for Java, see "Understanding UltraLite Development" [*Native UltraLite for Java User's Guide*, page 35].

3. Write your application code.

   Create forms for your application and write code that includes:

   ♦ Code to create, open and connect to a database

   ♦ Code to access and modify data

   ♦ Code for synchronization, if required for your application.

4. Deploy your application to the device.

   You can run the application in the development environment to confirm functionality, and you can configure application settings or synchronize your data to an enterprise database.

Each of these processes is outlined in detail in each interface book.

Developing applications with UltraLite static interfaces

For the static interfaces, the SQL statements to be used in an application must be specified at compile time. This is in contrast to the UltraLite components, which support a dynamic interface.

The overall development process for each static interface is similar, but the details are different. For more information, see the following:

- ♦ “Developing static C++ applications” [*UltraLite Static C++ User’s Guide,* page 3]
- ♦ “Developing embedded SQL applications” [*UltraLite Embedded SQL User’s Guide,* page 3]
- ♦ “Introduction to Native UltraLite for Java” [*Native UltraLite for Java User’s Guide,* page 1]

Data access in UltraLite

UltraLite provides the following data access methods:

- ♦ **Static SQL** The static interfaces (embedded SQL, static C++, and static Java) use structured query language (SQL) to access data. The SQL they use is *static*, meaning that the statements are all specified at the time the application is built.
- ♦ **Dynamic SQL** The UltraLite components can use *dynamic SQL*, which permits SQL statements to be constructed at runtime.
- ♦ **Table-based API** Both the UltraLite components and static C++ interface provide a table-based API that accesses rows one at a time.

Static SQL

The distinguishing features of static SQL are as follows:

- ♦ The full richness of the SQL language supported by Adaptive Server Anywhere can be used. An application will, however, use a limited number of these features, and so the supporting database engine can be quite small. By contrast, there must be support for every feature that could be used in a dynamic SQL implementation.

  ☞ For more information about Adaptive Server Anywhere SQL support, see “SQL Statements” [*ASA SQL Reference*, page 213].
- ♦ Since all SQL requirements are known at the time the application is built, the SQL statements themselves can be compiled into an executable form. They will tend to execute faster than an equivalent dynamic SQL implementation (which must be interpreted).
- ♦ More extensive optimization of how to execute a query is possible since this happens when the application is built.

Dynamic SQL

The distinguishing features of dynamic SQL are as follows:

- ♦ The ability to define SQL statements at runtime provides additional flexibility.
- ♦ There is no need for the analysis step during application development.
- ♦ The data structures used to execute SQL statements can be built as required. This contrasts with static SQL in which these data structures are built when the application is built. Dynamic SQL applications are therefore larger than the corresponding static SQL implementations unless a large number (probably greater than 100) of statements are present in the application.

- While the development model for static interfaces ensures that the data addressed by queries in the application is included in the UltraLite database, dynamic SQL shifts that responsibility to the application developer.

Table-based API

The distinguishing features of the table-based API are as follows:

- You can develop applications without learning SQL.
- The interface is simpler. However, you can access data in only one table at a time.

CHAPTER 2

Tutorial: Walking Through a Sample UltraLite Application

About this chapter

This chapter illustrates some key features of UltraLite by walking through a sample application. The sample application is a simple sales-status application built around a database named CustDB (Customer Database).

The chapter includes information on how to run the sample application, and a brief description of how the application works.

Before you begin

To get the most from this chapter, you should be able to run the sample application as you read.

This chapter assumes that you have read the chapter "Welcome to UltraLite" on page 3. Much of the material in this chapter is explained in a more general manner elsewhere in the book. Cross references to these places are provided.

Contents

Introduction

CustDB is a sample application included with UltraLite. It is a simple sales-status application that you can run against any of the supported databases, and on any of the supported target operating systems.

By working with the CustDB sample application, this chapter demonstrates the following core features of UltraLite.

- UltraLite database applications run on small devices using very limited resources.
- UltraLite applications include a relational database engine.
- UltraLite applications share data with a central, consolidated database in a two-way synchronization scheme. The UltraLite databases are also called **remote** databases.
- Each remote database contains a subset of the data in the consolidated database.
- The MobiLink synchronization server carries out data synchronization between the consolidated database and each UltraLite installation.
- SQL scripts stored in the consolidated database implement the synchronization logic.
- You can use Sybase Central to browse and edit the synchronization scripts.

A separate version of the CustDB application is provided for each UltraLite programming interface. Each version has similar features, although sometimes variations are made to conform to the expectations of a particular platform. The CustDB sample code for each interface provides a source for your own application development, as it covers many common UltraLite development tasks.

This chapter walks through a desktop version of the application to illustrate the features included in the applcation.

The CustDB sample application

Versions of the CustDB application are supplied for each supported interface, including complete source control. This tutorial uses the compiled version of the application for Windows NT/2000/XP.

When running the sample application, you are acting as an order taker or sales manager. The application allows you to view outstanding orders, approve or deny orders, and add new orders.

You can carry out the following tasks with the sample application.

- View lists of customers and products.
- Add new customers.
- Add or delete orders.
- Scroll through the list of outstanding orders.
- Accept or deny orders.
- Synchronize changes with the consolidated database.

When you run the CustDB UltraLite application, you are working on a single remote database, and synchronizing your changes with a consolidated database.

In a typical UltraLite installation, there will be many remote databases, each running on a handheld device, and each containing a small subset of the data from the consolidated database.

File locations for the sample application

Your UltraLite installation includes the files needed to run the sample application, and the source code used to develop it. Studying the sample application source code is a good way to learn more about UltraLite.

When you install SQL Anywhere Studio, the UltraLite sample files are installed into a directory named *Samples\Ultralite* under your installation directory.

Runtime file location

To run the CustDB sample application, you need the following components:

- **The consolidated database** An Adaptive Server Anywhere version of the customer database is installed as the file *custdb.db* in the *Samples\UltraLite\Custdb* subdirectory of your SQL Anywhere directory.

  This database serves as a consolidated database. It contains the following information:

  - MobiLink system tables that hold the synchronization metadata.
  - The CustDB data, stored in rows in base tables.
  - The synchronization scripts.

  During the installation process, an ODBC data source named UltraLite 9.0 Sample is created for this database.

- **The MobiLink synchronization server** The MobiLink synchronization server is in the *win32* subdirectory of your SQL Anywhere directory.
- **The UltraLite application executable** A version of the sample is supplied for each interface and operating system. For the static programming interfaces (embedded SQL, static C++ API, and static Java API) the executable and source code is held in a subdirectory of your *Samples\Ultralite* directory. Each UltraLite component has a separate directory immediately under the *Samples* directory.

Source file locations

Source code is provided for both the consolidated database and the UltraLite application in the *Samples\UltraLite\CustDB* and *Samples\MobiLink\CustDB* subdirectories of your SQL Anywhere directory.

- **Consolidated database source** In this chapter we use the Adaptive Server Anywhere CustDB database as the consolidated database.

  You can also build Sybase Adaptive Server Enterprise, Microsoft SQL Server, or Oracle consolidated databases and run the application against those database-management systems.

  You can use one of the SQL scripts in the *Samples\MobiLink\CustDB* directory to build a consolidated database for a DBMS other than Adaptive Server Anywhere.

  - **custase.sql** Sybase Adaptive Server Enterprise.
  - **custdb.sql** Sybase Adaptive Server Anywhere.
  - **custdb2.sql** IBM DB2.
  - **custmss.sql** Microsoft SQL Server.
  - **custora.sql** Oracle.

  The Adaptive Server Anywhere consolidated database is already built and installed. You only need the scripts to make a consolidated database using another relational database product. You do not need the scripts for this tutorial.

  ☞ For more information, see “The CustDB Sample Application” [*MobiLink Synchronization User’s Guide,* page 429].

- **Application source** For the static programming interfaces (embedded SQL, static C++ API, and static Java API) the executable and source code is held in a subdirectory of your *Samples\Ultralite* directory. Each UltraLite component has a separate directory immediately under the *Samples* directory.

Synchronization techniques in the sample application

The sample application demonstrates several useful synchronization techniques. This chapter provides a glimpse at synchronization, but in order to understand how to use these techniques in applications, you need to understand in more detail how the synchronization process works.

Synchronization is carried out using the MobiLink synchronization server, running on your desktop machine, against the CustDB sample database. For more information, see “The CustDB Sample Application” [*MobiLink Synchronization User’s Guide,* page 429].

For more information

☞ For an overview of the synchronization process, see “The synchronization process” [*MobiLink Synchronization User’s Guide,* page 21].

☞ For a description of how to write the synchronization scripts that control synchronization, see “Writing Synchronization Scripts” [*MobiLink Synchronization User’s Guide,* page 37].

☞ For information on the techniques used in the CustDB sample application, see “The CustDB Sample Application” [*MobiLink Synchronization User’s Guide,* page 429].

Lesson 1: Start the MobiLink synchronization server

When you start the sample UltraLite application for the first time, it contains no data. The application carries out an initial synchronization to download an initial copy of the data from the consolidated database. You must have the database server running in order to carry out this initial download, and you must also have the MobiLink synchronization server running against the UltraLite sample database.

The SQL Anywhere Studio installation adds some items to the Start menu to make this step easier.

❖ To start the MobiLink synchronization server against the consolidated database

1. Start the consolidated database server, running the CustDB sample database.

   The Adaptive Server Anywhere consolidated database server runs on your desktop machine. From the Start menu, choose Programs ➤ Sybase SQL Anywhere 9 ➤ UltraLite ➤ Personal Server Sample for UltraLite.

2. Start the MobiLink synchronization server against the CustDB database.

   The MobiLink synchronization server connects to the consolidated database server through ODBC. It could run from a separate machine from the database server, but in this example we will run it on the same machine.

   From the Start menu, select Programs ➤ SQL Anywhere 9 ➤ MobiLink ➤ Synchronization Server Sample.

   The command executed by this icon connects the MobiLink synchronization server to the consolidated database server.

Lesson 2: Start the sample application and synchronize

When started for the first time, the sample UltraLite application contains no data. In this step, you start the sample application, and carry out an initial synchronization with the consolidated database to obtain an initial set of data. The particular data you download depends on the user ID you enter when you start the application.

Start the application

❖ To start and synchronize the sample application

1. Launch the sample application.

   From the Start menu, choose Programs ➤ SQL Anywhere 9 ➤ UltraLite ➤ Windows Sample Application.

2. Enter an employee ID.

   When running through this section as a tutorial, enter a value of 50 and press ENTER. The application also allows values of 51, 52, or 53, but behaves slightly differently in these cases.

   The application synchronizes after you enter the employee ID, and a set of customers, products, and orders are downloaded to the application.

3. Confirm that the data has been synchronized into the application.

   Confirm that a company name and a sample order appear on the application window.

You have now synchronized your data.

☞ For the next step, see "Lesson 3: Add an order" on page 20.

Lesson 3: Add an order

In this section, you display the initial data in the sample application and add a new order. These step are carried out in a similar way in each version of the application.

The application holds information about a set of orders. For each order, this data includes the customer, the product, the quantity, the price, and any applicable discount. Also included are a status field and a notes field, which you can modify from the application.

Only unapproved orders are downloaded to the application. The sample application does not receive all the orders listed in the ULOrder table in the consolidated database. You control which information is sent to your application using synchronization scripts.

Add an order

❖ **To add an order**

1. Scroll through the outstanding orders.

   Click Next to display the next customer.

2. Open the window to enter a new order.

   From the Order menu, choose New.

   The Add New Order screen is displayed.

3. Choose a customer.

   The UltraLite application holds the complete list of customers from the consolidated database. To see this list, open the Customer drop-down list.

   Choose **Basements R Us** from the list. The current list of orders does not have any from this customer.

4. Choose a product.

   The UltraLite application holds the complete list of products from the consolidated database. To see this list, open the Product drop-down list box.

   Choose **Screwmaster Drill** from the list. The price of this item is automatically entered in the Price field.

5. Enter the quantity and discount.

   Enter a value of 20 for the quantity, and a value of 5 for the discount.

6. Press Enter to add the new order.

7. Click X to close the New Order screen.

You have now modified the data in your local UltraLite database. This data is not shared with the consolidated database until you synchronize.

☞ For the next step, see "Lesson 4: Act on some existing orders" on page 22.

Lesson 4: Act on some existing orders

In this step, you approve one order and deny another. Approving or denying orders updates two columns in the local database. No data in the consolidated database is changed until you synchronize.

The instructions for this step are very similar for all platforms.

❖ To approve, deny, and delete orders

1. Approve the order from Apple Street Builders.
   - Go to the first order in the list, which is from Apple Street Builders.
   - Tap or Click Approve to approve the order.
   - Add a note to your approval, saying **Good Work!**.
   - The order appears with a status of Approved.

2. Deny the order from Art's Renovations.
   - Go to the next order in the list, which is from Art's Renovations.
   - Tap or click Deny to deny this order.
   - Add a note stating **Discount too high**.

3. Delete the order from Awnings R Us.
   - Go to the next order in the list, which is from Awnings R Us.
   - Delete this order by choosing the menu item Options ➤ Delete. It disappears from your local copy of the data.

Having changed these orders, you now need to communicate your changes to the consolidated database.

☞ For the next step, see "Lesson 5: Synchronize your changes" on page 23.

Lesson 5: Synchronize your changes

In this step, you synchronize changes you made on your handheld device to the consolidated database.

For synchronization to take place, your MobiLink synchronization server must be running. If you have shut down your MobiLink synchronization server since the beginning of the tutorial, restart it.

☞ For instructions, see "Lesson 1: Start the MobiLink synchronization server" on page 18.

Synchronize your changes

❖ **To synchronize your changes**

1. If you are running on a Windows CE handheld device, place the device in its cradle, so that it can connect to the machine running the MobiLink synchronization server.

2. Choose File ➤ Synchronize to synchronize your data.

3. Confirm that the synchronization took place.
   - ♦ Confirm that the approved order for Apple Street Builders is no longer in your application.
   - ♦ The synchronization process for this sample application removes approved orders from your application.

☞ For the next step, see "Lesson 6: Confirm the synchronization at the consolidated database" on page 24.

Lesson 6: Confirm the synchronization at the consolidated database

In this step, you use Interactive SQL to connect to the consolidated database and confirm that the changes made have been synchronized. This step is independent of the platform on which your UltraLite application is running

❖ **To confirm that the changes are synchronized to the consolidated database**

1. Connect to the consolidated database from Interactive SQL.

   In the Interactive SQL Connect dialog, choose the **UltraLite 9.0 Sample** ODBC data source.

2. Confirm the status change of the approved and denied orders.

   To confirm that the approval and denial have been synchronized, issue the following statement.

   ```
   SELECT order_id, status
   FROM ULOrder
   WHERE status IS NOT NULL
   ```

   The results show that order 5100 is approved, and 5101 is denied.

3. Confirm that the deleted order has been removed.

   The deleted order has an order_id of 5102. The following query returns no rows, demonstrating that the order has been removed from the system.

   ```
   SELECT *
   FROM ULOrder
   WHERE order_id = 5102
   ```

The tutorial is now complete.

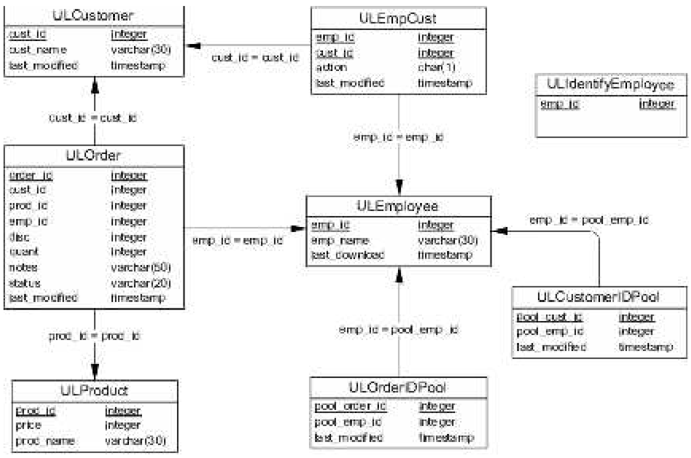
Lesson 7: Browse the consolidated database

You can use Sybase Central to manage MobiLink synchronization. The synchronization logic is held in the consolidated database.

This section describes how to use Sybase Central to browse the scripts in the CustDB consolidated database.

The CustDB database

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



The tables hold the following information.

- **ULCustomer** A list of customers.
- **ULProduct** A list of products.
- **ULEmployee** A list of sales employees. This table is not present in the UltraLite database.
- **ULEmpCust** A many-to-many relationship between employees and customers. This table is not present in the UltraLite database.
- **ULOrder** A list of orders, including details of the customer who placed the order, the employee who took the order, and the product being ordered.
- **ULCustomerIDPool** A table to maintain unused unique primary key values on the customer table throughout a deployed UltraLite system.
- **ULOrderIDPool** A table to maintain unused unique primary key values on the order table throughout a deployed UltraLite system.

♦ **ULIdentifyEmployee** This table holds a list of employee ID numbers.

Connect to the CustDB database from Sybase Central

1. Start the CustDB database:
   ♦ Select Programs ➤ Sybase SQL Anywhere 9 ➤ UltraLite ➤ Personal Server Sample for UltraLite.

   An Adaptive Server Anywhere database server starts, running the CustDB UltraLite Sample Database.

2. Start Sybase Central:
   ♦ From the Start menu, select Programs ➤ Sybase SQL Anywhere 9 ➤ Sybase Central.

3. Connect Sybase Central to the sample database:
   ♦ In Sybase Central, select Tools ➤ Connect. If there is a choice of connection types, select MobiLink. The MobiLink Connect dialog appears.

   Select ODBC, enter **UltraLite 9.0 Sample** in the Data Source box. Click OK to connect.

   You are now connected to the CustDB sample database.

Browse the synchronization scripts

From Sybase Central, you can browse through the tables, users, synchronized tables, and synchronization scripts that are stored in the consolidated database. Sybase Central is the primary tool for adding these scripts to the database.

Open the Connection Scripts folder. The right hand pane lists a set of synchronization scripts and a set of events that these scripts are associated with. As the MobiLink synchronization server carries out the synchronization process, it triggers a sequence of events. Any synchronization script associated with an event is run at that time. By writing synchronization scripts and assigning them to the synchronization events, you can control the actions that are carried out during synchronization.

Open the Synchronized Tables folder, and open the ULCustomer table folder. The right hand pane lists a pair of scripts that are specific to this table, and their corresponding events. These scripts control the way that data in the ULCustomer table is synchronized with the remote databases.

This section does not discuss the content of the synchronization scripts. These are discussed in detail in the chapter “Writing Synchronization Scripts” [*MobiLink Synchronization User’s Guide*, page 37] and in “The CustDB Sample Application” [*MobiLink Synchronization User’s Guide*, page 429].

CHAPTER 3

UltraLite Databases

About this chapter

This chapter provides basic information about data storage in UltraLite databases. It also describes features other than data access features, such as user authentication and character sets.

Contents

Databases and schema files

On most platforms, UltraLite databases are held in files. On the Palm OS the database can also be held in the Palm persistent store.

UltraLite database files contain tables and indexes that represent the data. They also hold supplementary information to provide transaction processing features. For more information, see "How UltraLite tracks row states" on page 33.

Each database has a well-defined collation sequence (character set and sort order). The collation sequence is defined when the database is created. For more information, see "Character sets in UltraLite" on page 40.

Most relational database systems include a special set of tables called the system tables, or catalog. These tables hold the database schema, or metadata: they hold the definitions of all the other tables and objects in the database.

UltraLite does not store its database schema in a set of system tables. Instead, the schema is held in the database file in a compact manner. The schema definition is built into the database file in one of the following ways:

♦ If you are using an UltraLite component, the schema definition is held in a separate file called a **schema file**. The schema file is specified in a connection parameter, and is applied to the database on the first connection attempt.

  You can upgrade the schema of your database by supplying a new schema file and applying it to the existing database.

  ☞ For more information, see "The UltraLite Schema Painter" on page 73.

♦ If you are using a static UltraLite interface, the schema definition is built into the application when the database code is generated from the reference database.

  ☞ For more information, see "The static UltraLite development process" on page 197, and "The UltraLite generator" on page 96.

UltraLite database files

The physical storage of the UltraLite database depends on the target platform.

♦ **Palm Computing Platform** The database is stored in the Palm persistent (static) memory using the Data Manager API. For devices operating Palm OS version 4.0, you can store UltraLite databases in the file-based storage of expansion cards.

- **Windows CE** The database is stored in the file system. On Windows CE the default file is *\UltraLiteDB\ul.udb*. On other versions of Windows the default file is *ul_<project>.udb* in the working directory of the application, where *<project>* is the UltraLite project name used during the development process.

- **Static Java** The database is either transient, or is stored as a file in the file system. By default, it is transient.

When creating your database, there are several configuration parameters you can supply. These control features such as encryption.

For more information

- **UltraLite for MobileVB** See “CreateDatabaseWithParms method” [*UltraLite ActiveX User’s Guide*, page 121].

- **UltraLite ActiveX** See “CreateDatabaseWithParms method” [*UltraLite ActiveX User’s Guide*, page 121].

- **Native UltraLite for Java** See **ianywhere.native_ultralite.DatabaseManager** in the Native UltraLite for Java API Reference.

- **UltraLite.NET** See **DatabaseManager class** in the UltraLite.NET API Reference.

- **UltraLite for C++** See **UltraLite_DatabaseManager class** in the UltraLite C++ API Reference.

- **UltraLite for embedded SQL** See “Macros and compiler directives for UltraLite C/C++ applications” on page 215.

- **UltraLite static C++** See “Macros and compiler directives for UltraLite C/C++ applications” on page 215.

- **UltraLite static Java** See “UltraLite API reference” [*UltraLite Static Java User’s Guide*, page 58].

UltraLite database characteristics

When you create an UltraLite database, you set some database-wide characteristics. These include the following:

- **Case sensitivity** The case sensitivity of a database affects all string comparisons. It must be specified when the database is created because indexes are stored in sorted order, and the order depends on whether the database is case sensitive or not.

- ♦ **Character set** The character set of a database also affects sort orders, and must be specified when the database is created.

  ☞ For more information, see "Character sets in UltraLite" on page 40.

Altering the schema of UltraLite databases

If you develop a new version of an UltraLite application, you may wish to alter the schema of the database. You can deploy an upgraded UltraLite schema and maintain the data in existing end-user databases subject to some restrictions. This feature is not available to UltraLite applications built using the static Java API.

The mechanism for deploying an upgraded schema depends on whether you use an UltraLite component (which requires a new schema file) or whether you use a static interface (in which case the schema definition is held in the generated application code).

❖ To deploy a new schema (UltraLite components)

1. Create a new schema file.

   You can create your new schema using the Schema Painter or using ulinit. If you use the schema painter, you can rename columns. If you use ulinit to create the new schema, you lose any data held in renamed columns—the schema is interpreted as dropping the old column and creating a new column.

2. Apply the schema file to the existing database.

   The components expose a database schema as a DatabaseSchema or ULDatabaseSchema object. You obtain a schema object using the DatabaseSchema property on the Connection or ULConnection object.

   Use the ApplyFile method on the ULDatabaseSchema object to apply the new schema.

❖ To deploy a new schema (embedded SQL and static C++ API)

1. Modify the schema in your reference database.

2. Create the new version of your application.

3. Ensure that your application calls ULEnableGenericSchema().

   When a new UltraLite application built with a static interface 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.

☞ See “ULEnableGenericSchema function” [*UltraLite Embedded SQL User’s Guide*, page 111].

4. The schema is upgraded automatically when the new application is applied.

   For information on how the schema upgrade happens, see

How the schema upgrade works

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.

The schema upgrading process relies on matching names in the old and new schema. If a row in the database is incompatible with the new schema, that row is deleted from the database. In general, adding constraints to tables that have data in them or carrying out unpredictable column conversions may result in lost rows.

The schema upgrade proceeds as follows:

1. Any tables that were in the old schema but not in the new schema are dropped.

2. Any tables that are in the new schema but were not in the old are created.

3. For any table that exists in both old and new, but with a different definition, columns are added and dropped as needed. If a new column is not nullable and has no default value, it is filled with zeros (numeric data types), the empty string (character data types) and an empty binary value.

4. Columns whose properties have changed are then modified.

   Caution
   If an error occurs during conversion for any row, that row is dropped and the SQL warning SQLE_ROW_DROPPED_DURING_SCHEMA_-UPGRADE is set.

5. Indexes and constraints are rebuilt. This step may also result in rows being dropped if, for example, an index is redefined as UNIQUE but has duplicate values.

Upgrading UltraLite software

For any UltraLite 9.0 database, you can apply a new schema file to your UltraLite 9.0 database.

On Windows operating systems other than Windows CE, if you are using UltraLite for MobileVB or UltraLite ActiveX, you cannot open an UltraLite database file created using 8.0.2 software. For more information, see “UltraLite behavior changes” [*What’s New in SQL Anywhere Studio,* page 44].

☞ For compatibility of UltraLite database files with previous releases, see “UltraLite runtime character sets” on page 41.

Information storage in UltraLite databases

UltraLite stores the rows of data in each table. It also stores state information about each row, and stores indexes to efficiently access the rows.

UltraLite compresses variable length strings, integers, numerical values, and date/time data in the database. It does not compress columns containing character or binary data, except on Windows CE where Unicode strings are compressed by storing in a UTF-8 representation.

How UltraLite tracks row states

Each row in an UltraLite database has a one-byte marker to keep track of the state of the row. The row states are used to control transaction processing, recovery, and synchronization.

When a delete is issued, the state of each affected row is changed to reflect the fact that it was deleted. Rolling back a delete is as simple as restoring the original state of the row.

When a delete is committed, the affected rows are not always removed from memory. If the row has never been synchronized, then it is removed. If the row has been synchronized, then it is not removed until the next synchronization confirms the delete with the consolidated database. After the next synchronization, the row is removed from memory.

Similarly, when a row is updated in an UltraLite database, a new version of the row is created. The states of the old and new rows are set so the old row is no longer visible and the new row is visible. When an update is synchronized, both the old and new versions of the row are needed to allow conflict detection and resolution.

The old version of the row is deleted after synchronization. If a row is updated many times between synchronizations, only the oldest version of the row and the most recent version of the row are kept.

Indexes in UltraLite databases

UltraLite indexes are B+ trees with very small index entries.

Except for pure Java databases, each index entry is exactly two bytes, and each index page contains 256 entries. Since index pages are rarely 100% full and each index has some fixed overhead, the memory used by an UltraLite index is more than two bytes per row in the table. The overhead for each index is just over 1 kb per index. Typically, UltraLite index pages on larger tables will be least 85% full.

No similar consistent rule can be given for the memory requirements of UltraLite Java databases.

Backup, recovery, and transaction processing

The best way of making a backup of an UltraLite application is to synchronize with a consolidated database. To restore an UltraLite database, start with an empty database and populate it from the consolidated database through synchronization.

UltraLite provides protection against system failures, but not against media failures. If the UltraLite data store itself is corrupted, the only way to protect is through synchronization.

UltraLite provides transaction processing. If an application using an UltraLite database stops running unexpectedly, the UltraLite database automatically recovers to a consistent state when the application is restarted. All transactions committed prior to the unexpected failure are present in the UltraLite database. All transactions not committed at the time of the failure are rolled back.

UltraLite does not use a transaction log to perform recovery. Instead, UltraLite uses the state byte for every row to determine the fate of a row when recovering. When a row is inserted, updated, or deleted in an UltraLite database, the state of the row is modified to reflect the operation and the connection that performed the operations. When a transaction is committed, the states of all rows affected by the transaction are modified to reflect the commit. If an unexpected failure occurs during a commit, the entire transaction is rolled back on recovery.

☞ For more information on state bytes, see "How UltraLite tracks row states" on page 33.

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** This option provides protection against casual attempts to access data in the database. It does not provide as much security as strong encryption. Obfuscation has minimal performance impact.
- **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 provides 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.

Encrypting an UltraLite database

To encrypt an UltraLite database, you supply an encryption key when you create the database (that is, on the first connection 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.

For more information

- **UltraLite for MobileVB** See “Encryption and obfuscation” [*UltraLite for MobileVB User’s Guide*, page 57].
- **UltraLite ActiveX** See “Encryption and obfuscation” [*UltraLite ActiveX User’s Guide*, page 73].
- **Native UltraLite for Java** See **ianywhere.native_ultralite.ConnectionParms** in the API Reference.
- **UltraLite.NET** See ConnectionParms class in the UltraLite.NET API Reference.
- **UltraLite for C++** See **UltraLite_Connection_iface** in the UltraLite for C++ API Reference.

- **UltraLite for embedded SQL** See “Encrypting UltraLite databases” [*UltraLite Embedded SQL User’s Guide*, page 56].
- **UltraLite static C++** See “Encrypting UltraLite databases” [*UltraLite Static C++ User’s Guide*, page 31].
- **UltraLite static Java** See “Encrypting UltraLite databases” [*UltraLite Static Java User’s Guide*, page 41].

User authentication

UltraLite provides optional database user IDs and passwords for user authentication. Unlike Adaptive Server Anywhere and other multi-user database systems, UltraLite user IDs are used for authentication only, not for permission checking or object ownership within a database. By default, UltraLite databases have no user authentication.

UltraLite provides a built-in scheme to authenticate users before allowing them to connect to the UltraLite database.

When an UltraLite database is created, it has an initial user ID of DBA, with a password of SQL. These are also default values on connection parameters. You can avoid user authentication by not supplying **uid** or **pwd** connection parameters when connecting.

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

If the database is case insensitive (the default) then the user ID and password are case insensitive. If the database is case sensitive, then the password is case sensitive.

UltraLite user IDs are separate from MobiLink user names and from user IDs in any reference database or consolidated databases you use during development and after deployment. In many cases you may wish to provide code so that the values used for each are the same, but they do remain distinct concepts. For example, in the CustDB sample application, you are prompted for an employee number when starting the application. This employee number identifies the database for the purposes of MobiLink synchronization, and is not an UltraLite user ID for connection or data access purposes.

❖ To add user authentication to your application

1. Connect to the database using the default **uid** and **pwd** parameters.

   New users have to be added from an existing connection.

2. Prompt for a user ID and password.

3. Grant access to this user.

Use the ULConnection.GrantConnectTo method to enable user authentication and provide access to a specific user ID and password combination.

4. Optionally, revoke access from the original user ID.

For more information

- **UltraLite for MobileVB** See “User authentication” [*UltraLite ActiveX User’s Guide*, page 93].
- **UltraLite ActiveX** See “User authentication” [*UltraLite ActiveX User’s Guide*, page 93].
- **Native UltraLite for Java** See “User authentication” [*Native UltraLite for Java User’s Guide*, page 53].
- **UltraLite.NET** See “User authentication” [*UltraLite.NET User’s Guide*, page 43].
- **UltraLite for C++** See “User authentication” [*UltraLite C++ User’s Guide*, page 37].
- **UltraLite for embedded SQL** See “Adding user authentication to your application” [*UltraLite Embedded SQL User’s Guide*, page 52].
- **UltraLite static C++** See “Adding user authentication to your application” [*UltraLite Static C++ User’s Guide*, page 28].
- **UltraLite static Java** See “Adding user authentication to your application” [*UltraLite Static Java User’s Guide*, page 38].

Character sets in UltraLite

There are several places in UltraLite applications where character set issues can arise:

- **The UltraLite schema** The database itself has a single collating sequence (character set and sort order), which is specified when the database is created. The collating sequence determines the order of character data in indexes, the results of string comparisons, and so on.

  ☞ For more information, see "UltraLite database character sets" on page 40.

- **The UltraLite runtime library or component** The UltraLite component or runtime library that accesses the database uses a character set for messages and other interactions with the environment.

  The runtime character sets may be Unicode or ANSI. The character set used determines how the data is stored in the database file. Once a database is created, you must use a single runtime character set to manage it.

  ☞ For more information, see "UltraLite runtime character sets" on page 41.

- **Synchronization** When data in the UltraLite database is synchronized with a MobiLink synchronization server, the character set used in the UltraLite database and that in the consolidated database must be consistent.

  ☞ For more information, see "Synchronization and character sets" on page 43.

UltraLite database character sets

If you create an UltraLite database schema using the schema painter, you specify the character set and collating sequence as you create the database schema.

UltraLite applications use the native multi-byte character encoding of the target platform for reasons of efficiency. When the reference database uses a different character encoding, the UltraLite application uses the default collation of the target device.

UltraLite applications use the collating sequence of the reference database if either of the following conditions is met.

- The reference database uses a single-byte character set.

- The native character encoding of the target device is multi-byte, the reference database uses the same multi-byte character encoding, and the UltraLite analyzer can find a compact representation for the collation sequence used by the reference database.

For example, if you use a 932JPN reference database to build an UltraLite application for the Windows CE platform, the application will use Unicode and the default Unicode collation information. If, instead, you use a 932JPN reference database to build an application for the Japanese Palm Computing Platform, then the UltraLite application can inherit the collation information because the native character encoding is the same as that of the reference database.

Sort orders

If the character set is single byte, or the native character set of the target device is the same as the character set of the reference database, columns that are CHAR(*n*) or VARCHAR(*n*) compare and sort according to the collation sequence of the reference database.

☞ For information about creating databases, see "The UltraLite Schema Painter" on page 73.

Data storage

The way that character data is stored depends not only on the collation sequence used when creating the schema, but also on the character set (ANSI or Unicode) of the UltraLite runtime library that manages the database.

☞ For more information, see "UltraLite runtime character sets" on page 41.

UltraLite runtime character sets

The character set of the UltraLite runtime library is different depending on the target operating system. The character set determines how the .

Palm Computing Platform

Single-byte Palm Computing Platform devices uses a character set based on code page 1252 (the Windows US code page). The 1252Latin1 code page is appropriate for developing applications for the Palm Computing Platform. The 1252Latin1 code page is the default Adaptive Server Anywhere collation sequence. Japanese Palm Computing Platform devices use 932JPN.

Windows CE

The Windows CE operating system uses Unicode. UltraLite running on Windows CE also uses Unicode to store CHAR(*n*) and VARCHAR(*n*) columns. Adaptive Server Anywhere collating sequences define behavior for 8-bit ASCII character sets.

UltraLite for Windows CE uses the Adaptive Server Anywhere collating sequence when comparing Unicode characters that have a corresponding 8-bit ASCII character in the collating sequence being used, allowing accented characters to compare equal and sort with unaccented characters.

Unicode characters that have no corresponding 8-bit ASCII character use a comparison of two Unicode values.

Windows desktop operating systems

The UltraLite components are all Unicode based.

The runtime library used by UltraLite embedded SQL and static C++ applications on Windows NT/2000/XP and Windows 98 is provided in both ANSI and UNICODE versions. UltraLite versions before version 9 included only an ANSI version of the runtime library.

Compatibility of database files

The runtime or component that is used to create the database determines how characters are stored within the database. You cannot create an UltraLite database using an ANSI component or runtime library and then use that database file with a Unicode component or runtime library.

The following table lists the character set in use by UltraLite components and runtime libraries. These character sets dictate whether or not you can use a database file created by one version of UltraLite in another. In particular, note that you cannot open a database created by version 8.0.2 UltraLite for MobileVB or UltraLite ActiveX on a Windows operating system (other than Windows CE) in version 9.0 or later software.

Environment	**8.0.2**	**9.0 and later**
Windows components	ANSI	Unicode
Windows CE components	Unicode	Unicode
Windows (Native UltraLite for Java)	Unicode	Unicode
Windows CE (Native UltraLite for Java)	Unicode	Unicode
Windows 8.0.2 DLL	ANSI	N/A
Windows (ulrt.lib)	N/A	ANSI
Windows (ulrtw.lib)	N/A	Unicode
Windows (ulrtc.lib (engine))	N/A	Unicode
Windows CE	Unicode	Unicode

Static Java

The error-handling objects **SQLException** and **SQLWarning** provide the capability for Java applications to obtain error or warning messages. By default, these messages are supplied in English.

Localized error and warning messages may be obtained in a non-English language by setting the Java Locale to the appropriate language.
For example, to obtain French messages, the following code fragment might be used:

```
java.util.Locale locale = new java.util.Locale( "fr", "" );
java.util.Locale.setDefault( locale );
```

The default Locale should be set at the start of the program. Once a message is placed in an error-handling object, the language to be used for the message is established for that execution of the program.

Synchronization and character sets

When you synchronize, the MobiLink synchronization server always translates characters uploaded from your application database to Unicode and passes them to your consolidated database server using the Unicode ODBC API. The consolidated database server, or its ODBC driver, then performs any translation that may be required to convert them to the character encoding of your consolidated database. This second translation will always occur unless your consolidated database uses Unicode.

When information is downloaded, the consolidated database server converts the characters to Unicode. The MobiLink Synchronization server then automatically translates the characters, if necessary, to suit the requirements of your UltraLite application.

When both UltraLite application and consolidated database use the same character encoding, no translation is necessary. If translation is necessary, problems can arise when multiple character codes in your UltraLite application map to a single Unicode value, or vice versa. In this event, the MobiLink synchronization server translates in a consistent manner, but behavior is influenced by the translation mechanism within the consolidated database server.

UltraLite database limitations

The following table lists the absolute limitations imposed by data structures in the software on the size and number of objects in an UltraLite database. In most cases, the memory, CPU, and storage device of the computer impose stricter limits.

Item	Limitation
Number of connections per database	14
Number of columns per table	65535 but limited by row size [1]
Number of indexes	65535
Number of rows per database	Limited by persistent store
Number of rows per table	65534
Number of tables per database	Approximately 1000 [2]
Number of tables referenced per transaction	No limit
Row size	Approximately 4 kb (compressed). LONG VARCHAR and LONG BINARY values are stored separately, and are in addition to the 4 kb limit.
File-based persistent store	2 Gb file or OS limit on file size
Palm Computing Platform database size	128 Mb (Primary storage) 2 Gb (expansion card file system)

☞ For other limitations, see "Overview of SQL support in UltraLite" on page 108.

Adaptive Server Anywhere features not available in UltraLite databases

The following Adaptive Server Anywhere features are not available in UltraLite databases:

[1]Row size is limited to about 4 kb, so the practical limit on the number of columns per table is much smaller than this: much less than 4000 in most situations.

[2]If you set the page size to 2 kb, the maximum number of tables is reduced to approximately 500.

- **Cascading updates and deletes** Some applications rely on declarative referential integrity to implement business rules. These features are not available in UltraLite databases.
- **Check constraints** You cannot include table or column check constraints in an UltraLite database.
- **Computed columns** You cannot include computed columns in an UltraLite database.
- **Timestamp columns** You cannot use Transact-SQL timestamp columns in UltraLite databases. Transact-SQL timestamp columns are created with the following default:

  ```
  DEFAULT TIMESTAMP
  ```

  You can use columns created as follows:

  ```
  DEFAULT CURRENT TIMESTAMP
  ```

  There is a behavior difference between the two: a DEFAULT CURRENT TIMESTAMP column is not automatically updated when the row is updated, while a DEFAULT TIMESTAMP column is automatically updated. You must explicitly update columns created with DEFAULT CURRENT TIMESTAMP if you wish the column to reflect the latest update time.
- **Global temporary tables** The temporary aspect of global temporary tables is not recognized by UltraLite. They are treated as if they were permanent base tables, which you should use instead.
- **Declared temporary tables** You cannot declare a temporary table within an UltraLite application.
- **Stored procedures** You cannot call stored procedures or user-defined functions in an UltraLite application.
- **System table access** There are no system tables in an UltraLite database.
- **SAVEPOINT statement** UltraLite databases support transactions, but not savepoints within transactions.
- **SET OPTION statement** You can determine the option settings in an UltraLite database by setting them in the reference database, but you cannot use the SET OPTION statement in an UltraLite application to change option settings.
- **System functions** You cannot use Adaptive Server Anywhere system functions, including property functions, in UltraLite applications.

- **Functions** Not all SQL functions are available for use in UltraLite applications. For example, the ISDATE and ISNUMERIC functions are not available for use in UltraLite databases.

  Use of an unsupported function gives a `Feature not available in UltraLite` error.

- **Triggers** Triggers are not available in UltraLite databases.

- **Java in the database** You cannot include Java methods in your queries or make any other use of Java in the database.

- **Schema modification** To modify the schema of a UltraLite database, you must build a new version of your application.

  ☞ For more information, see "Schema changes in remote databases" [*MobiLink Synchronization User's Guide,* page 100].

- **Limited dynamic SQL** If you are using dynamic SQL in your UltraLite application, the range of SQL available is less than in Adaptive Server Anywhere.

  ☞ For more information, see "Dynamic SQL" on page 125

UltraLite tables must have primary keys

Each table in a static UltraLite application must include a primary key.

The UltraLite generator uses primary keys from your reference database to generate primary keys in the UltraLite database. If the primary key columns for any table are not included in the data required in the UltraLite database, the UltraLite generator looks for a uniqueness constraint on the table, and promotes the columns with such a constraint to a primary key in the UltraLite database. If there are no unique columns, the generator reports an error.

Primary keys are required not only for UltraLite applications, but also during MobiLink synchronization, to associate rows in the UltraLite database with rows in the consolidated database.

The UltraLite runtime library

The UltraLite runtime library is the code that manages UltraLite databases and synchronization. The runtime library is available for Windows, Windows CE, and the Palm OS. With the exception of the static Java API, the versions of the runtime library for these distinct target platforms are based on a single codebase. The runtime library is linked into each of the UltraLite components.

Development notes

♦ The same query cannot be executed more than once at a time. As a result, you cannot access more than one instance at a time of a result set for a given query in an UltraLite application.

Static Java API

The runtime library used by the static Java API is entirely separate from the runtime library

Threading in UltraLite applications

The UltraLite runtime library is threadsafe, so that you can develop multi-threaded applications for this library as long as the development tool and the target platform both support multi-threaded applications. The exceptions are as follows:

♦ You cannot develop multi-threaded applications using UltraLite for MobileVB, because of limitations in the underlying development tools.

♦ You cannot develop multi-threaded applications using UltraLite ActiveX, because of limitations in eMbedded Visual Basic and JScript.

♦ You cannot develop multi-threaded applications for the Palm OS, because of limitations in the operating system.

Multi-threaded UltraLite applications can access UltraLite databases in a read-only fashion during synchronization. During the download phase of synchronization, read-write access is permitted. You can disable access to data during synchronization by setting the

Static Java API

The runtime library used by the UltraLite static Java API is is thread-safe. Users of the Sun Java VM must use version 1.2 or later to run multi-threaded UltraLite applications. Users of the Jeode VM on Pocket PC and the IBM Java VM can run multi-threaded UltraLite applications even though these VMs are based on JDK 1.1.8.

The entire runtime is treated as a single critical section, only allowing one thread to enter it at a time.

CHAPTER 4

Connection Parameters

About this chapter

This chapter provides a reference for the parameters that establish and describe connections from client applications to a database.

Contents

Overview

You must supply connection parameters for an application to connect to an UltraLite database. The connection parameters must specify the database to which the connection is to be established, usually by providing a database filename and path.

As an application may be compiled for more than one platform, a separate parameter for each target platform is available to identify the database. If user authentication is enabled, the connection parameters must also specify a user name and password.

☞ For more information, see "Database identification parameters" on page 54, and "User authentication parameters" on page 58.

UltraLite databases are typically created on the first connection attempt, at which time the connection parameters must include a schema file (for UltraLite components) as well as optional parameters to adjust database features. If you are using embedded SQL, the static C++ API, or the static Java API, the database is created from information already stored in the application and no schema file is needed.

☞ For more information, see "Database schema parameters" on page 61, and "Additional connection parameters" on page 65.

All connection parameters are case insensitive.

The following table lists the available connection parameters. Connection parameters used only when creating a database are marked with an asterisk (*).

Parameter	**Description**
Additional Parms	A placeholder for additional connection parameters. See "Additional Parms connection parameter" on page 65
Cache Size	Defines the size of the cache. See "Cache Size connection parameter" on page 66.
Connection Name	Specifies a connection name. See "Connection Name connection parameter" on page 60.
Database On CE	The path and filename of the UltraLite database file to which you want to connect on Windows CE. See "Database On CE connection parameter" on page 54.
Database On Desktop	The path and filename of the UltraLite database file to which you want to connect. See "Database On Desktop connection parameter" on page 55.

Parameter	**Description**
Database On Palm	An identifier for the UltraLite database on Palm OS. See "Database On Palm connection parameter" on page 56.
Encryption Key	An encryption key for the database. See "Encryption Key connection parameter" on page 63.
Obfuscate*	Apply a simple encryption scheme to the database. See "Obfuscate connection parameter" on page 67.
Page Size	The database page size. See "Page Size connection parameter" on page 67.
Palm Allow Backup	Controls HotSync backup behavior on Palm OS devices. See "Palm Allow Backup parameter" on page 68.
Password	A password for the user. See "Password connection parameter" on page 58.
Reserve Size	Defines the reserve size. See "Reserve Size connection parameter" on page 68.
Schema On CE*	The path and filename of the UltraLite schema on Windows CE. See "Schema On CE connection parameter" on page 61.
Schema On Desktop*	The path and filename of the UltraLite schema. See "Schema On Desktop connection parameter" on page 62
Schema On Palm*	The UltraLite schema for the Palm OS. See "Schema On Palm connection parameter" on page 63.
User ID	The user ID with which you connect to the database. See "User ID connection parameter" on page 59.
VFS On Palm*	Identifies the Palm card as using the virtual file system. See "VFS On Palm parameter" on page 56.

Specifying file paths

Filenames and paths in connection parameters are subject to the following requirements, depending on the UltraLite Component you are using:

Target platform	Requirement
Java	All backslashes must be escaped. For example, `"file_name=\\UltraLite\\MyFile.udb"`.
Windows CE	Paths are absolute.
Windows	Paths may be absolute or relative.

Specifying connection parameters

Each connection parameter can be specified in the following ways:

♦ **As a property of a Connection Parameters object** The following interfaces provide a connection parameters object. This object has properties that are individual connection parameters.

- UltraLite for MobileVB
- UltraLite ActiveX
- Native UltraLite for Java
- UltraLite.NET

♦ **In the AdditionalParms property** The interfaces that supply a connection parameters object include an Additional Parms property. This property takes a connection string as its value.

♦ **As a keyword in a connection string** All UltraLite interfaces can supply a connection string when connecting. The keywords in the connection string are individual connection parameters.

♦ **In the UL_STORE_PARMS macro** Embedded SQL and the static C++ API both use the UL_STORE_PARMS macro to hold connection parameters that affect database file. The parameters are used only on the initial connection attempt, when the database is created. The UL_STORE_PARMS macro takes a connection string.

For example, the following definition sets a connection parameter.

```
#define UL_STORE_PARMS    UL_TEXT( "reserve_size=2m" )
```

Where possible, it is recommended that you use the Connection Parameters object. It provides easier checking and a more systematic interface than using a connection string.

Connection strings and connection parameters

Some less commonly used parameters can be specified only in a connection string. Depending on the interface, the connection string can be supplied in the AdditionalParms property, in the UL_STORE_PARMS macro, or in an Open method that takes a connection string as argument.

If a parameter is specified both in a property and in a connection string, the value in the property takes precedence.

Database identification parameters

The connection parameters in this section are used to identify the UltraLite database. At least one of these parameters must be specified on each connection attempt.

For more information

- **UltraLite for MobileVB** See "OpenConnection method" [*UltraLite ActiveX User's Guide*, page 126], and "OpenConnectionWithParms method" [*UltraLite ActiveX User's Guide*, page 126].
- **UltraLite ActiveX** See "OpenConnection method" [*UltraLite ActiveX User's Guide*, page 126], and "OpenConnectionWithParms method" [*UltraLite ActiveX User's Guide*, page 126].
- **Native UltraLite for Java** See **ianywhere.native_ultralite.DatabaseManager** in the Native UltraLite for Java API Reference.
- **UltraLite.NET** See **DatabaseManager class** in the UltraLite.NET API Reference.
- **UltraLite for C++** See **UltraLite_DatabaseManager class** in the C++ API Reference.
- **UltraLite for embedded SQL** See "Adding user authentication to your application" [*UltraLite Embedded SQL User's Guide*, page 52].
- **UltraLite static C++** See "Adding user authentication to your application" [*UltraLite Static C++ User's Guide*, page 28].
- **UltraLite static Java** See "Adding user authentication to your application" [*UltraLite Static Java User's Guide*, page 38].

Database On CE connection parameter

Function

The path and filename of the UltraLite database file to which you want to connect on Windows CE.

Syntax

Interface	Connection parameter
UltraLite for MobileVB	DatabaseOnCE
UltraLite ActiveX	DatabaseOnCE
UltraLite.NET	DatabaseOnCE
Native UltraLite for Java	databaseOnCE
Connection string	**ce_file**

☞ For information about using the connection parameter, see "Specifying connection parameters" on page 52.

Values *String*

Default *\UltraLiteDB\ulstore.udb*

Description When creating a database, this parameter names the new database file.

When opening a connection to an existing database, the parameter identifies the database.

- If the filename does not include an extension, the file of extension *.udb* is presumed.
- The full path of the file must be specified. No substitutions are performed on this value.
- The schema file is not required if a *.udb* file already exists.
- Database On CE is required if you use a database with any name other than the default.
- You must ensure that this directory exists when the connection parameter is used. UltraLite does not create the directory automatically.

Example To create and connect to the sample database, *udemo.udb*:

```
"schema_file=MyOrders.usm;CE_FILE=udemo.udb"
```

See also "Specifying file paths" on page 51

Database On Desktop connection parameter

Function The database file to which you want to connect in the desktop development environment.

Syntax

Interface	**Connection parameter**
UltraLite for MobileVB	DatabaseOnDesktop
UltraLite ActiveX	DatabaseOnDesktop
UltraLite.NET	DatabaseOnDesktop
Native UltraLite for Java	databaseOnDesktop
Connection string	{ **file_name** | **DBF** }

☞ For information about using the connection parameter, see "Specifying connection parameters" on page 52.

Values *String*

Default *ulstore.udb*

Description When creating a database, this parameter names the new database file.

When opening a connection to an existing database, the parameter identifies the database.

If the filename does not include an extension, the file of extension *.udb* is assumed.

Example

♦ To create and connect to the sample database, *udemo.udb*, installed in the directory *c:\Program Files\Sybase\SQL Anywhere 9*, use the following connection string:

```
"schema_file=MyOrders.usm;DBF=udemo.udb"
```

See also "Specifying file paths" on page 51

Database On Palm connection parameter

Function The Palm creator ID of the database to which you want to connect.

Syntax

Interface	Connection parameter
UltraLite for MobileVB	DatabaseOnPalm
Native UltraLite for Java	databaseOnPalm
Connection string	**palm_db**

☞ For information about using the connection parameter, see "Specifying connection parameters" on page 52.

Values *String*

Default The creator ID of the application.

Description When creating a database, this parameter names the new database file.

When opening a connection to an existing database, the parameter identifies the database.

See also "Specifying file paths" on page 51

VFS On Palm parameter

Function Identifies the Palm card as using the virtual file system.

This parameter is available only in UltraLite for MobileVB. To use the virtual file system from an embedded SQL or static C++ API application, use the EnablePalmFileDB function.

Syntax

Interface	**Connection parameter**
UltraLite for MobileVB	VFSOnPalm
Connection string (UltraLite for MobileVB)	PALM_FS=VFS

☞ For information about using the connection parameter, see "Specifying connection parameters" on page 52.

Values

As a parameter, VFSOnPalm is a boolean value.

In the connection string, the parameter must be specified as follows:

```
PALM_FS=VFS
```

Description

The palm_fs=vfs parameter needs to be specified both for CreateDatabase and OpenConnection if you are using the VFS card for Palm devices and you want the database stored on the card.

To create, drop or connect to a database on a memory card, the following connection parameter must be specified in the parameter string:

```
Palm_fs=vfs
```

The default database filename is *ul_udb_YYYY.udb*, where *YYYY* is the creator ID of the application. You can control the filename by specifying a different creator ID in the DatabaseOnPalm parameter. For example, the following connection string references a database on the card with filename *ul_udb_XXXX.udb*:

```
palm_db=XXXX;palm_fs=vfs
```

Even when the VSFOnPalm parameter is specified, the palm_db parameter (Database on Palm) must be set to a valid creator ID.

If the VSF On Palm parameter is not specified, the database is created (or dropped from or to which you are connecting) on the device and not the card.

User authentication parameters

User authentication parameters are used to identify the user as authorized to use the database.

For more information

- **UltraLite for MobileVB** See “OpenConnection method” [*UltraLite ActiveX User’s Guide*, page 126], and “OpenConnectionWithParms method” [*UltraLite ActiveX User’s Guide*, page 126].
- **UltraLite ActiveX** See “OpenConnection method” [*UltraLite ActiveX User’s Guide*, page 126], and “OpenConnectionWithParms method” [*UltraLite ActiveX User’s Guide*, page 126].
- **Native UltraLite for Java** See **ianywhere.native_ultralite.DatabaseManager** in the Native UltraLite for Java API Reference.
- **UltraLite.NET** See **DatabaseManager class** in the UltraLite.NET API Reference.
- **UltraLite for C++** See **UltraLite_DatabaseManager class** in the C++ API Reference.
- **UltraLite for embedded SQL** See “Adding user authentication to your application” [*UltraLite Embedded SQL User’s Guide*, page 52].
- **UltraLite static C++** See “Adding user authentication to your application” [*UltraLite Static C++ User’s Guide*, page 28].
- **UltraLite static Java** See “Adding user authentication to your application” [*UltraLite Static Java User’s Guide*, page 38].

Password connection parameter

Function A password for the user. Passwords are case insensitive if the database is case insensitive and case sensitive if the database is case sensitive.

Syntax

Interface	**Connection parameter**
UltraLite for MobileVB	Password
UltraLite ActiveX	Password
UltraLite.NET	Password
Native UltraLite for Java	password
Connection string	{ **password** | PWD }

☞ For information about using the connection parameter, see "Specifying connection parameters" on page 52.

Usage — Anywhere

Values — *String*

Default — SQL

Description — Every user of a database has a password. The password must be supplied for the user to be allowed to connect to the database.

The Password (PWD) connection parameter is not encrypted.

Example

♦ The following connection string fragment supplies the user ID DBA and password SQL.

```
"UID=DBA;PWD=SQL;schema_file=MyOrders.usm"
```

User ID connection parameter

Function — The user ID with which you log on to the database. An authenticated user for the database. User ID's are case-insensitive if the database is case-insensitive and case sensitive if the database is case sensitive.

Databases are created with a single authenticated user DBA whose initial password is SQL. By default, connections are opened using the UID=DBA and the PWD=SQL. To disable the default user, use

```
connection.revokeConnectionFrom.
```

To add a user or change a user's password, use

```
connection.grantConnectTo.
```

Syntax

Interface	Connection parameter
UltraLite for MobileVB	UserID
UltraLite ActiveX	UserID
UltraLite.NET	UserID
Native UltraLite for Java	userID
Connection string	{ **userid** | **UID** }

☞ For information about using the connection parameter, see "Specifying connection parameters" on page 52.

Usage — Anywhere

Values	*String*
Default	DBA
Description	You must always supply a user ID when connecting to a database, unless you leave the database using the default user ID and password of DBA and SQL.

Example

♦ The following connection string fragment supplies the user ID DBA and password SQL:

```
"schema_file=MyOrders.usm;uid=DBA;pwd=SQL"
```

Connection Name connection parameter

Function

Specifies a name for the connection. This is only needed if you create more than one connection to the database.

Syntax

Interface	Connection parameter
UltraLite for MobileVB	ConnectionName
UltraLite ActiveX	ConnectionName
UltraLite.NET	ConnectionName
Native UltraLite for Java	connectionName
Connection string	**con**

☞ For information about using the connection parameter, see "Specifying connection parameters" on page 52.

Database schema parameters

The following keywords are used to specify a schema for an UltraLite database. Thus, schema parameters are vital database creation parameters, as your schema determines which tables and columns exist in your database. Only one file value is used with the platform specific keyword taking precedence over the generic keyword.

For more information

- **UltraLite for MobileVB** See "CreateDatabase method" [*UltraLite ActiveX User's Guide*, page 120], and "CreateDatabaseWithParms method" [*UltraLite ActiveX User's Guide*, page 121].
- **UltraLite ActiveX** See "CreateDatabase method" [*UltraLite ActiveX User's Guide*, page 120], and "CreateDatabaseWithParms method" [*UltraLite ActiveX User's Guide*, page 121].
- **Native UltraLite for Java** See **ianywhere.native_ultralite.DatabaseManager** in the Native UltraLite for Java API Reference.
- **UltraLite.NET** See **DatabaseManager class** in the UltraLite.NET API Reference.
- **UltraLite for C++** See **UltraLite_DatabaseManager class** in the C++ API Reference.
- **UltraLite for embedded SQL** See "Macros and compiler directives for UltraLite C/C++ applications" on page 215.
- **UltraLite static C++** See "Macros and compiler directives for UltraLite C/C++ applications" on page 215.

Schema On CE connection parameter

Function

To identify the schema filename deployed to Windows CE.

Syntax

Interface	**Connection parameter**
UltraLite for MobileVB	SchemaOnCE
UltraLite ActiveX	SchemaOnCE
UltraLite.NET	SchemaOnCE
Native UltraLite for Java	schemaOnCE
Connection string	**ce_schema**

☞ For information about using the connection parameter, see "Specifying connection parameters" on page 52.

Values *String*

Default The recommended file extension is *.usm.*

Description Used only when you create a database.

The path and filename of the UltraLite schema file on Windows CE. The default extension for UltraLite schema files is .usm. This is a required parameter when using CreateDatabase for CE.

Example

- The following connection string fragment supplies the ce_schema and schema_file parameters.

```
"CE_SCHEMA=orders.usm;SCHEMA_FILE=MyOrders.usm"
```

Schema On Desktop connection parameter

Function To identify the schema file in the desktop development environment.

Syntax

Interface	**Connection parameter**
UltraLite for MobileVB	SchemaOnDesktop
UltraLite ActiveX	SchemaOnDesktop
UltraLite.NET	SchemaOnDesktop
Native UltraLite for Java	schemaOnDesktop
Connection string	**schema_file**

☞ For information about using the connection parameter, see "Specifying connection parameters" on page 52.

Values *String*

Default The recommended file extension is *.usm.*

Description Used only when you create a database.

The path and filename of the UltraLite schema in the development environment.

Example

- The following connection string fragment supplies the ce_schema and schema_file parameters.

```
"CE_SCHEMA=orders.usm;SCHEMA_FILE=MyOrders.usm"
```

Schema On Palm connection parameter

Function
To identify the schema file deployed to a Palm OS device.

Syntax

Interface	Connection parameter
UltraLite for MobileVB	SchemaOnPalm
Native UltraLite for Java	schemaOnPalm
Connection string	**palm_schema**

☞ For information about using the connection parameter, see "Specifying connection parameters" on page 52.

Values
String

Default
The Palm file extension on the desktop is *.pdb*.

Description
Used only when you create a database.

The filename of the UltraLite schema for Palm.

Although *.pdb* is the extension on the desktop, do not supply *.pdb* in your connection parameter string.

Example
♦ The following connection string fragment supplies the palm_schema and schema_file parameters.

```
"PALM_SCHEMA=orders;SCHEMA_FILE=MyOrders.usm"
```

Encryption Key connection parameter

Function
An encryption key for the database. You can define an encryption key for your UltraLite database when CreateDatabase is called.

Syntax

Interface	Connection parameter
UltraLite for MobileVB	EncryptionKey
UltraLite ActiveX	EncryptionKey
UltraLite.NET	EncryptionKey
Native UltraLite for Java	encryptionKey
Connection string	{ **key** | **dbkey** }

☞ For information about using the connection parameter, see "Specifying connection parameters" on page 52.

Values	*String*
Default	No key is provided.
Description	Used only when you create a database.

If a database is created using an encryption key, the database file is strongly encrypted using the AES 128-bit algorithm, which is the same algorithm used to encrypt Adaptive Server Anywhere databases. Use of strong encryption provides security against skilled and determined attempts to gain access to the data, but may have a significant performance impact.

Example

```
"schema_file=MyOrders.usm;KEY=MyKey"
```

See also "Encrypting UltraLite databases" on page 36

Additional connection parameters

These are optional parameters to configure a database when it is created. Some of these parameters can influence performance, so it is suggested that you test these parameters to find the optimal performance for your application.

For more information

- **UltraLite for MobileVB** See "CreateDatabase method" [*UltraLite ActiveX User's Guide*, page 120], and "CreateDatabaseWithParms method" [*UltraLite ActiveX User's Guide*, page 121].
- **UltraLite ActiveX** See "CreateDatabase method" [*UltraLite ActiveX User's Guide*, page 120], and "CreateDatabaseWithParms method" [*UltraLite ActiveX User's Guide*, page 121].
- **Native UltraLite for Java** See **ianywhere.native_ultralite.DatabaseManager** in the Native UltraLite for Java API Reference.
- **UltraLite.NET** See **DatabaseManager class** in the UltraLite.NET API Reference.
- **UltraLite for C++** See **UltraLite_DatabaseManager class** in the C++ API Reference.
- **UltraLite for embedded SQL** See "Macros and compiler directives for UltraLite C/C++ applications" on page 215.
- **UltraLite static C++** See "Macros and compiler directives for UltraLite C/C++ applications" on page 215.

Additional Parms connection parameter

Function

Permits additional connection parameters to be specified.

Syntax

Interface	Connection parameter
UltraLite for MobileVB	AdditionalParms
UltraLite ActiveX	AdditionalParms
UltraLite.NET	AdditionalParms
Native UltraLite for Java	additionalParms

☞ For information about using the connection parameter, see "Specifying connection parameters" on page 52.

Usage	Some less commonly used connection parameters do not have properties associated with them in the UltraLite components. These parameters can be specified as a connection string in AdditionalParms.
Values	A connection string.
Default	None.
See also	“Specifying connection parameters” on page 52

Cache Size connection parameter

Function — Defines the size of the database cache.

Syntax

Interface	Connection parameter
Connection string	**cache_size**

☞ For information about using the connection parameter, see “Specifying connection parameters” on page 52.

Usage — Used when you configure a database. Use k or K, m or M to denote kilobytes or megabytes, respectively.

Values — The minimum cache size is 4K.

Default — The default is 16 x page_size. Actual value used is rounded down to the nearest multiple of page_size.

Description — Defines the size of the cache. You can specify the size in units of bytes. Use the suffix k or K to indicate units of kilobytes and use the suffix M or m to indicate megabytes

The default cache size is sixteen pages. Using the default page size of 4 K, the default cache size is therefore 64 K. The minimum cache size is platform dependent.

The default cache size is conservative. If your testing shows the need for better performance, you should increase the cache size.

Increasing the cache size beyond the size of the database itself provides no performance improvement. Also, large cache sizes may interfere with the number of other applications you can use.

On the Palm Computing Platform, the parameter applies only to virtual file system (VFS) databases. The cache itself resides in record storage, not VFS storage.

Example — For example, the following string sets the cache size to 128 K.

```
"cache_size=128k"
```

Obfuscate connection parameter

Function Obfuscates the database. Obfuscation is a form of simple encryption.

Syntax

Interface	Connection parameter
Connection string	**obfuscate**

☞ For information about using the connection parameter, see "Specifying connection parameters" on page 52.

Values **0** or **1**. A value of 1 indicates that the database should be obfuscated.

Usage Used only when you create a database.

Embedded SQL and static C++ API developers can also use the UL_ENABLE_OBFUSCATION macro to obfuscate a database. See "UL_ENABLE_OBFUSCATION macro" on page 215.

Default By default, databases are not obfuscated.

See also "Encrypting UltraLite databases" on page 36

Page Size connection parameter

Function Defines the database page size.

Syntax

Interface	Connection parameter
Connection string	**page_size**

☞ For information about using the connection parameter, see "Specifying connection parameters" on page 52.

Usage Used only when you create a database.

Used when you configure a database. Use k or K to denote kilobytes.

Default The default page size for UltraLite databases is 4 K. The range of size is 2 K to 4 K.

Description UltraLite databases are stored in pages. I/O operations are carried out a page at a time. It can be used on any target platform. Setting a page size of 2 K reduces the maximum number of tables to approximately 500.

This parameter is ignored when starting an existing database.

Example

You can specify 2 kb pages using the following storage parameters string:

```
"schema_file=MyOrders.usm;PAGE_SIZE=2K"
```

Palm Allow Backup parameter

Function

Control backup behavior over HotSync on Palm devices.

Syntax

Interface	Connection parameter
Connection string	**palm_allow_backup**

☞ For information about using the connection parameter, see "Specifying connection parameters" on page 52.

Usage

Used when you configure a database.

Values

yes or **no**.

Description

If the backup bit is set on the UltraLite database, and if this parameter is set to **yes**, the entire Palm database is backed up every time the device is synchronized using HotSync. If this parameter is not set, UltraLite ensures that the backup bit is cleared. In most applications, data is backed up by synchronization, so there is no need to set this parameter.

The backup bit is set when a database file is deployed by HotSync, and can also be set by the ULUtil utility. For more information, see "The UltraLite Palm utility" on page 103.

Example

The following string sets the parameter.

```
#define UL_STORE_PARMS UL_TEXT("palm_allow_backup=yes")
```

Reserve Size connection parameter

Function

Reserves file system space for storage of UltraLite persistent data.

Syntax

Interface	Connection parameter
Connection string	**reserve_size**

☞ For information about using the connection parameter, see "Specifying connection parameters" on page 52.

Usage

Use k or K, m or M to denote kilobytes or megabytes, respectively.

Values

Values can be expressed in kb or mb.

Description

The reserve_size parameter allows you to pre-allocate the file system space required for your UltraLite database without actually inserting any data. Reserving file system space can improve performance slightly and also prevent out of memory failures. By default, the persistent storage file only grows when required as the application updates the database.

Reserve_size reserves file system space, which includes the metadata in the persistent store file, and not just the raw data. The metadata overhead as well as data compression must be considered when deriving the required file system space from the amount of database data. Running the database with test data and observing the persistent store file size is recommended.

The reserve_size parameter reserves space by growing the persistent store file to the given reserve size on startup, regardless of whether the file previously existed. The file is never truncated.

This parameter does not apply to the Palm Computing Platform unless the application uses the Virtual File System (VFS).

Example

Use the reserve_size parameter to pre-allocate space as follows:

```
"CE_SCHEMA=orders;RESERVE_SIZE=128K"
```

This example ensures that the persistent store file is at least 128 K upon startup.

CHAPTER 5

Utility Programs

About this chapter

This chapter provides reference information about UltraLite utility programs.

Contents

Introduction to UltraLite utilities

The database **schema** is the database without the data. It is the collection of tables, indexes, and so on within the database, and all the relationships between them. The **schema file** stores schema information. You do not alter the schema of an UltraLite database directly. Instead, you modify a schema file (which typically has the extension *.usm*) and upgrade the database schema from that file using a built-in UltraLite function in your application.

You can create an UltraLite schema file in the following ways:

- **Generate the schema from an Adaptive Server Anywhere database** If you have the Adaptive Server Anywhere database management system, you can generate an UltraLite schema file using the *ulinit* command line utility.

- **UltraLite Schema Painter** The UltraLite Schema Painter is a graphical utility for creating and editing UltraLite schema files.

  To start the Schema Painter, choose Start ➤ Programs ➤ SQL Anywhere 9 ➤ UltraLite ➤ UltraLite Schema Painter, or double-click a schema file (with extension *usm*) in Windows Explorer.

- **The ulxml command line utility** The ulxml command line utility allows you to open *usm* files and save them to XML format, open XML files and save them as *usm* files, and to export XML files to a format suitable for Palm. For more information, see "The ULXML utility" on page 89.

The UltraLite Schema Painter

Applies to	UltraLite components.
Function	The UltraLite Schema Painter allows you to create a new UltraLite schema file or edit an existing one. Thus, even if you do not have Adaptive Server Anywhere installed you can:

- ♦ Create a new schema, or edit an existing schema
- ♦ Add, edit, or delete a new table by double-clicking Add Table
- ♦ Add, edit, and delete publications, columns, foreign keys, and indexes
- ♦ Export the schema as a *.pdb* file suitable for Palm OS devices
- ♦ Save as a *.usm* file or Open a *.usm* file for Pocket PC devices

Starting the UltraLite Schema Painter

❖ To start the UltraLite Schema Painter

1. Start the UltraLite Schema Painter:

   Choose Start ➤ Programs ➤ SQL Anywhere 9 ➤ UltraLite Schema Painter.

Create, save and export schema files

❖ To create a new schema file

1. Open the Tools folder and double-click Create UltraLite Schema.
2. In the New Schema dialog, type in a file name.
3. Click OK to create the schema.

❖ To save a file

1. Choose File ➤ Save to save the file.
2. You can select to Save in *.xml* or *.usm* format.

❖ To export a Palm schema file

1. Right-click the schema icon and choose Export Schema for Palm from the popup menu.
2. Enter a Creator ID.
3. Click OK.

Managing schema files

When you first rename a table or column in your schema UltraLite stores the original name of the table or column. For example, if you create a table named cust, and later rename it to customer, cust is saved as the old name. If you then renamed the table a second time, to customer_info, the old name remains cust.

The scheme is designed so that a schema file can be used to alter the schema of an existing database. For example, assume that version one of your application shipped with a table named cust. As part of the changes for version two, you modify your version one schema file by renaming the table to customer. This automatically saves cust as the old name. If you now apply this schema file to a version one database file, UltraLite looks for a table named cust, the old name, and rename it customer. The same applies to columns in a table.

It is therefore important for futue compatibility that you clear the old names from a schema file after a schema file is deployed.

☞ For more information, see "Altering the schema of UltraLite databases" on page 30.

❖ To clear all of the old names in the schema file after deployment

1. Open the schema file in the UltraLite Schema Painter
2. Right-click the database
3. Select Clear Upgrade Information

This sets all of the old names for tables and columns to empty values. You can then safely edit your schema file for the next version of your application.

Manual renaming old names

Sometimes it may be desirable to manually alter the old names of tables and columns. For example, you may have versions one and two of your application deployed and wish to create a single UltraLite schema file that can upgrade both versions one and two of this database to version three.

❖ **To manually change old names**

1. Open your schema in the UltraLite Schema Painter
2. Right-click the database
3. Choose “Prepare Schema for Deployment

You can use this feature to inspect the current old names in your schema. If you useulxml, you can explicitly set the old name of tables and columns in the <table> and <column> XML elements.

UltraLite Schema Painter dialogs

The following sections discuss the various dialogs, and the controls on these dialogs, that are available to you when you use the UltraLite Schema Painter.

UltraLite Schema Painter Options dialog

This dialog provides options for running the UltraLite Schema Painter.

The UltraLite Schema Painter Options dialog has the following components:

- ♦ **Informational messages** Select the warnings and messages you wish to receive.
  - **Warn before closing an open schema file.** Select this option for a warning when closing a schema file that is in use.
  - **Show post-deployment info.** Select this option for post-deployment messages.
  - **Warn when attempting to edit an indexed column.** Select this option for a warning when editing a column already in an index.
  - **Warn before changing the Nullability of a column being added to a primary key.** Select this option for a warning when adding a column that allows null values to a primary key.
- ♦ **Recent files options** Click Clear Now to delete the list of recently accessed schema files.

New UltraLite Schema dialog

This dialog provides options for the creation of a new UltraLite schema file.

The New UltraLite Schema dialog has the following components:

- ♦ **What should the new filename for the new UltraLite schema be?** Enter a name for your UltraLite schema or click Browse to select an existing schema.

  If you choose to browse to an existing schema, click Save to open it.

- **What collation sequence should be used for your new UltraLite schema?** Select a collation sequence from the dropdown list. A collation sequence is a character set and sorting order. For languages based on the Roman alphabet, such as most European languages, the default sequence, **1252LATIN1 - Code Page 122, Windows Latin 1, Western**, will suffice.

- **Check this to make your UltraLite database case-sensitive.** Select this option if you want your UltraLite database to be case-sensitive.

New Table dialog

This dialog provides options for the creation of a new table in your UltraLite database.

The New Table dialog has the following components:

- **Name** Enter a name for the table.

- **Columns** Lists all the columns currently in the table. Select a column by clicking on its name in the table.

- **Add** Click Add to add a new column to the table.

- **Edit** Click Edit to edit the selected column. This button is only available when a column is selected from the Columns table. Columns included in an index or primary key cannot be edited without first removing them from the index or primary key.

- **Remove** Click Remove to remove the selected column. This button is only available when a column is selected from the Columns table. Columns included in an index or primary key cannot be removed without first removing them from the index or primary key.

- **Remove All** Click Remove All to delete all columns from the table.

- **Primary Key** Click Primary key to designate one or more of the columns in the table as its primary key or to edit an existing primary key.

- **Synchronize** Select synchronization options for the table.
  - **Only changed rows (default)** Synchronizes only those rows that have been changed since the last synchronization.
  - **All rows** Synchronizes all rows in the table.
  - **No rows** This table is not synchronized.

New Column property sheet: General tab

The General tab of the New Column property sheet has the following components:

- **Name** Enter a name for the column.
- **Type** Select a datatype from the dropdown list.
  - **Size/Precision** Specify the size of the column for binary or char types, or the precision for numeric types.
  - **Scale** Specify the scale value for a numeric column.
- **Column allows null values** Check this option to allow null values. Null values are not permitted in primary keys.
- **Column default** Set a default value for the column.
  - **Default** Select a default value for the column from the list, or choose Specify value... to set a constant value.
  - **Value** Specify the column value. This option is only available when Specify value is selected as the default value.
  - **Partition size (optional)** Specify the partition size. The partition size restricts the maximum value that global autoincrement can use. This option is only available when Global autoincrement is selected as the default value.

New Column property sheet: Indexes tab

The Indexes tab of the New Column property sheet has the following components:

- **Indexes** Lists the indexes that contain the selected column.

Set Primary Key dialog

This dialog provides options for the creation and editing of a primary key.

The Set Primary Key dialog has the following components:

- **Index information** Provides information on the index.
  - **Index name** Enter a name for the index. For primary key indexes, the name is set to primary and cannot be changed.
  - **Unique index** Select Unique index to ensure that values are unique and not null. For primary key indexes, this option is checked and cannot be changed.

    - **Unique key** Select Unique key to ensure that values are unique. They may be null. For primary key indexes, this option is checked and cannot be changed.

- **Indexed Columns** Allows you to add or remove columns from the primary key index.
    - **Columns in the table** Lists all columns in the table that are not in the primary key index. Click on a column to select it, or hold down Control to select more than one column at a time.
    - **Asc.>>** Click Asc.>> to add one or more columns, chosen from the Columns in table list, to the index in ascending order.
    - **Desc.>>** Click Desc.>> to add one or more columns, chosen from the Columns in table list, to the index in descending order.
    - **<<** Click << to remove one or more columns, chosen from the Columns in the index list, from the index.
    - **Columns in the index** Lists all columns in the primary key index. Click on a column to select it, or hold down Control to select more than one column at a time.

Publication dialog

This dialog provides options for the creation of a new publication for your UltraLite database. A publication is a database object describing data to be replicated.

The Publication dialog has the following components:

- **Publication name** Enter a name for the publication.
- **Available tables** Lists all the tables in your database not yet in the publication. Click on a table to select it, or hold down Control to select more than one table at a time.
- **>>** Click >> to add one or more tables, chosen from the Available tables list, to the publication.
- **<<** Click << to add one or more tables, chosen from the Tables in the publication list, to the publication.
- **Tables in the publication** Lists all the tables in the publication. Click on a table to select it, or hold down Control to select more than one table at a time.

Database Schema property sheet: General tab

The General tab of the Database Schema property sheet has the following components:

- **Name** The name of the schema.
- **Type** The file type (usually UltraLite Schema).
- **Location** The path of the schema file.
- **Collation sequence** The collation sequence used in the file. A collation sequence is a character set and sorting order.
- **Case-sensitive** Yes if the database is case-sensitive, no if it is not.
- **Database properties** Lists various database properties and the values they are set to. Select a database property from the list to edit its value.
- **Edit** Click Edit to edit the selected database property.
- **Set Defaults** Click Set Defaults to set all database properties to their default values.

Database Schema property sheet: Certification tab

The Certification tab of the Database Schema property sheet has the following components:

- **Set Trusted Roots Certificate** Click Set to browse to a Certicom encryption certificate.
- **Clear Trusted Roots Certificate** Click Clear to remove the encryption certificate associated with your schema.
- **Save/View Trusted Roots Certificate** Save and view the certificate associated with your schema.

Save Certificate dialog

This dialog allows you to save and view the trusted root certificate associated with your schema.

The Save Certificate dialog has the following components:

- **What should the filename for the certificate be?** Enter a filename and path or click Browse to select one.
- **View certificate after saving** Select this option to view the certificate once you have saved it.

Database Property Editor dialog

This dialog allows you to edit the values of database properties.

The Database Property Editor dialog has the following components:

- **Please enter a new value for the database option** Enter a new value for the selected database property. Most database properties are very specific about the formatting of their values. If a database property is incorrectly formatted, an error will appear when trying to apply the changes in the Database Schema property sheet.

Save Schema to PDB file dialog

This dialog provides options for deploying your schema to a Palm device.

The Save Schema to PDB file dialog has the following components:

- **Palm Creator ID** Enter a Palm Creator ID.

  A Palm creator ID is assigned to you by Palm. You can use **Syb3** as your creator ID when you make sample applications for your own learning. However, when you create your commercial application, you should use your own creator ID.

- **What should the filename for the PDB be?** Enter a filename and path for your Palm database file, or click Browse to select a path and filename.

Schema Deployment dialog

This dialog provides options for modifying your database upon deployment.

The Schema Deployment dialog has the following components:

- **Tables in this schema** Lists the tables in the schema.

  The current table names are listed in the New Name column; the original table names are in the Old Name column. If a table name has been changed more than once, Old Name will record the name originally given to the table upon creation.

  To edit a table's old name, select the table from the list by clicking on it and then click on its old name.

- **Columns for table** Lists all the columns in a selected table.

  The current column names are listed in the New Name column; the original column names are in the Old Name column. If a column name has been changed more than once, Old Name will record the name originally given to the column upon creation.

  To edit a column's old name, select the column from the list by clicking on it and then click on its old name.

Set Index dialog

This dialog provides options for the creation of a new index.

The Set Index dialog has the following components:

- **Index information** Provides information on the index.
  - **Index name** Enter a name for the index.
  - **Unique index** Select Unique index to ensure that values are unique and not null.
  - **Unique key** Select Unique key to ensure that values are unique. They may be null.
- **Indexed Columns** Allows you to add or remove columns from the index.
  - **Columns in the table** Lists all columns in the table that are not in the index. Click on a column to select it, or hold down Control to select more than one column at a time.
  - **Asc.**>> Click Asc.>> to add one or more columns, chosen from the Columns in table list, to the index in ascending order.
  - **Desc.**>> Click Desc.>> to add one or more columns, chosen from the Columns in table list, to the index in descending order.
  - << Click << to remove one or more columns, chosen from the Columns in the index list, from the index.
  - **Columns in the index** Lists all columns in the index. Click on a column to select it, or hold down Control to select more than one column at a time.

Index property sheet: General tab

The General tab of the Index property sheet has the following components:

- **Index name** The name of the index.
- **Table** The name of the table containing the index.
- **Unique** If yes, values in the index must be unique and not null.
- **Unique key** If yes, values in the index must be unique, but they may be null.
- **This index includes the following columns:** Lists the columns contained in the index and their directions.

Create Foreign Key dialog

This dialog provides options for the creation of a foreign key.

The Create Foreign Key dialog has the following components:

- **Foreign key name** Enter a name for the foreign key.
- **Primary table** Select the primary table from the list of tables in your database.
- **Eligible indexes** Select an index from the list of indexes in your primary table.
- **Only check values on commit** Select this option to check values at commit, rather than upon insertion.
- **Allow Null in the foreign key columns** Allows null values in the columns of your foreign key.
- **Map** Click Map to map a column in your primary table, selected from the table above, to a column in your current table, selected in a subsequent dialog.

Map Primary Column dialog

This dialog provides options for the mapping of a column in your primary table to a column in your current table.

The Map Primary Column dialog has the following components:

- **Name** The name of your primary column.
- **Type** The data type of your primary column.
- **Default** The default value for your primary column.
- **Direction** Select ascending to sort your column in ascending order or descending to sort your column in descending order.
- **Foreign column** Select a column in your current table from the dropdown list to which to map the selected column in your primary table.
- **Properties** Displays the properties of the selected foreign column.

Foreign Key property sheet: General tab

The General tab of the Foreign Key property sheet has the following components:

- **Foreign key name** The name of the foreign key.

- **Table** The table that contains the foreign key.
- **Referenced table** The table referenced by the foreign key.
- **Referenced index** The index referenced by the foreign key.
- **Check on commit** If yes, checks values at commit, rather than upon insertion.
- **Nullable** If yes, allows null values in the columns of the foreign key.
- **This foreign key contains the following columns:** Lists columns in the foreign key and the columns they map to in the primary table.

Table property sheet: General tab

The General tab of the Table property sheet has the following components:

- **Table** The name of the referecing table.
- **Synchronization** The synchronization scheme of the referencing table, one of Only changed rows, All rows, or No rows.
- **Columns** Lists the names, types, and inclusion in the primary key for all the columns in the referencing table.

UltraLite Schema Painter Tutorial

In this tutorial, you build a single-table database schema and export it to Palm.

☞ For more information on UltraLite schemas, see "Databases and schema files" on page 28.

When creating UltraLite schemas for a Palm device, the following information is necessary:

- A way to identify the database so an application can connect to it. This is done with the Palm creator ID.
- A way to identify the schema on the development machine so it can be copied to the device.
- A way to identify the schema on the device.

To complete this tutorial you need a directory to hold the files you create. This directory is assumed to be *C:\tutorial\*. If you create your tutorial directory elsewhere, supply the path to your location instead of *c:\tutorial\* throughout.

❖ To create a schema file using the UltraLite Schema Painter

1. Start the UltraLite Schema Painter:

   Click Start ➤ Programs ➤ SQL Anywhere 9 ➤ UltraLite ➤ UltraLite Schema Painter.

2. Create a new schema file called *tutCustomer*.
   - ♦ From the File menus, select New ➤ UltraLite Schema...
   - ♦ In the file dialog box, type **c:\tutorial\tutCustomer.usm** or Browse to the folder and enter **tutCustomer**.
   - ♦ Click OK to create the schema.

3. Create a table called customer.
   - ♦ Expand the *tutCustomer* item in the left pane of the UltraLite Schema Painter and select the Tables folder.
   - ♦ Open the Tables folder and double-click Add Table. The New Table dialog appears.
   - ♦ Enter the name customer.
   - ♦ In the New Table dialog, add columns with the following properties.

Column name	**Data type (Size)**	**Column Allows NULL values?**	**Default value**
Id	integer	No	autoincrement
Fname	char (15)	No	None
Lname	char (20)	No	None
City	char (20)	Yes	None
Phone	char (12)	Yes	555-1234

   - ♦ Set Id as the primary key: click Primary Key and add Id to the index, marking it as ascending.
   - ♦ Check your work and click OK to complete the table definition and dismiss the New Table dialog.

4. Click File ➤ Save to save the *tutcustomer.usm* file.

5. Export a Palm schema file.
   - ♦ Right click on the database icon and select Export Schema for Palm from the popup menu.

- Enter a Palm Creator ID of **Syb3**.

  A note on Palm Creator IDs
  A Palm creator ID is assigned to you by Palm. You can use Syb3 as your creator ID when you make sample applications for your own learning. However, when you create your commercial application, you should use your own creator ID.

- Leave the filename at its default setting to save the PDB file in your tutorial directory. Click OK.
- Exit the UltraLite Schema Painter.

You have now defined the schema of an UltraLite database. Although this database contains only a single table, you can use many tables in UltraLite databases.

The UltraLite initialization utility

Applies to — UltraLite components.

Function — The *ulinit* utility lets you create a.*usm* file for use with any UltraLite component. The utility connects to an Adaptive Server Anywhere database. Consequently, SQL Anywhere Studio (version 8.0.2 or later) is required in order to use it.

Syntax — **ulinit -f** *schema_file* **-n** *pub_name* [*options*]

Option	Description
-c *"connection_string"*	Supply database connection parameters in the form *keyword=value*, separated by semi-colons. You supply these so you may connect to an Adaptive Server Anywhere database.
-f *schema_file*	Specify the name of the output file. This option is required.
-m *version*	Specify the version string for generated MobiLink scripts.
-n *pubname*	Add tables to the UltraLite database schema. *pubname* specifies a publication in the reference database. Tables in the publication are added to the UltraLite database schema. Specify the option multiple times to add multiple publications in to the UltraLite database schema. To add all tables in the reference database to the UltraLite schema, specify **-n** **. This option is required.
-o *"keyword=value;. . . "*	Supply schema creation options.
-palm *id*	Create a schema file compatible with PalmOS. Id is the four digit Palm creator id that identifies the database.
-q	Quiet operation — only report errors and warnings.

Option	Description
-s *pubname*	Specify a publication for synchronization. *pubname* specifies a publication in the reference database that is added as a named publication to the UltraLite database. If -s is not supplied, the UltraLite schema has no named publications. This option can be used multiple times.
-t *file*	Specify the file containing the trusted root certificates.
-w	Do not display warnings.
-z *ordering*	Specify table ordering (for example, *-z table1,table2*).

Remarks

The `-n` and `-s` options both take publication names in the reference database as arguments, but serve different purposes:

- The `-n` option defines the tables to be included in the UltraLite database schema. It does not create named publications in the UltraLite database, and is not used for synchronization.
- The `-s` option defines named publications in the UltraLite database. These named publications are used for synchronization. The -s option does not define which tables are included in the UltraLite database schema.

Examples

The following example creates a file called *customer.usm* that contains the tables in TestPublication:

```
ulinit -c "uid=dba;pwd=sql" -f customer.usm -n TestPublication
```

The following example creates a schema with two distinct publications:

```
ulinit -c "dsn=dsn-name" -f schema.usm -n Pub1 -n Pub2 -s Pub1 -
        s Pub2
```

For example, one of the publications may contain a small subset of data for priority synchronization, while the other would contain the bulk of the data.

Synchronization of publications is managed with a bitmask in the UltraLite schema. For more information, see "Designing sets of data to synchronize separately" on page 156.

When creating an UltraLite schema for Palm with *ulinit*, use the `-palm` option. This generates a *.pdb* file.

```
ulinit -c "uid=dba;pwd=sql;dsn=ASA 9.0 Sample"
-f tutcustomer.usm -n TutCustomersPub -palm Syb3
```

Note
Syb3 is the four digit Palm registered creator ID that matches the creator ID of your application. For MobileVB developers, this must be set in your MobileVB project settings.

The PDB file generated by *ulinit* must be loaded to the Palm device. When an UltraLite application needs to connect to the database, it should include the creator ID in the parameters of the call to Open. For example:

```
DatabaseManager.OpenConnection( "palm_db=Syb3" )
```

The ULXML utility

Applies to

UltraLite components.

Function

The *ulxml* utility lets you convert data file formats. For example, you can create a *.usm* file based on an XML file. It can be used with any UltraLite component.

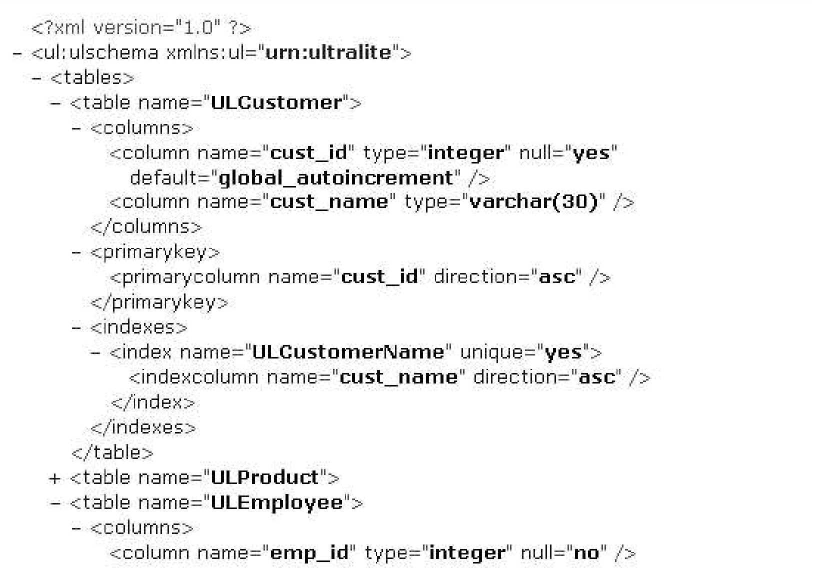
Syntax

ulxml [*options*] *input-file output-file*

Option	Description
-y	Overwrite output file if it already exists.
-to<*type*> *where type=xml|usm|pdb* Note: pdb files require a CreatorID.	Converts the file to one of these standard formats. Use *toxml* to convert an UltraLite schema to XML. Use *tousm* to convert an XML file to an UltraLite schema Use *topdb* to convert an XML file to an UltraLite schema for Palm.

The return code from ULXML is set to 0 on success and less than 0 on failure.

You can export your UltraLite schema so that you can work in XML format:



```
  <?xml version="1.0" ?>
- <ul:ulschema xmlns:ul="urn:ultralite">
  - <tables>
    - <table name="ULCustomer">
      - <columns>
          <column name="cust_id" type="integer" null="yes"
            default="global_autoincrement" />
          <column name="cust_name" type="varchar(30)" />
        </columns>
      - <primarykey>
          <primarycolumn name="cust_id" direction="asc" />
        </primarykey>
      - <indexes>
        - <index name="ULCustomerName" unique="yes">
            <indexcolumn name="cust_name" direction="asc" />
          </index>
        </indexes>
      </table>
    + <table name="ULProduct">
    - <table name="ULEmployee">
      - <columns>
          <column name="emp_id" type="integer" null="no" />
```

You can view and use the documented sample located in Samples\NativeUltraLiteForJava\sample.xml, Samples\UltraLiteActiveX\sample.xml, and Samples\UltraLiteForMobileVB\sample.xml.

Note
The UltraLite Schema Painter by default creates, opens and saves UltraLite schema files in their native USM file format. However, you are given the option to create, open and save XML files as well by choosing UltraLite XML Schema Files in any file type dropdown box.

The HotSync conduit installation utility

Function

The utility installs or removes a HotSync conduit onto the current machine.

Syntax

dbcond9 [*options*] *id*

Option	Description
id	The creator ID of the application to use the conduit
-n *name*	The name displayed by the HotSync manager.
-x	Remove the conduit for the specified creator ID

Description

Install a HotSync conduit onto the current machine. The HotSync manager must be installed in order for this to be run.

Options

id The application user ID who is to use the conduit. If a conduit already exists for the specified *creatorID*, it is replaced by the new conduit. This is a required option.

-n name The name displayed by the HotSync manager. This is also the name of the subdirectory where the conduit stores data. Do not use this option together with `-x`. The default value is **MobiLink conduit**.

-x Remove the conduit for the named *creatorID*. If `-x` is not specified, a conduit is installed.

Examples

The following command line installs the conduit for the CustDB sample application, which has a creator ID of Syb2:

```
dbcond9 -n CustDB Syb2
```

The SQL preprocessor

Applies to	Embedded SQL static development model only.
Function	The SQL preprocessor processes a C or C++ program containing embedded SQL, before the compiler is run.
Syntax	**sqlpp** [*options*] *sql-filename* [*output-filename*]

Option	Description
–c *“keyword=value;. . . ”*	Supply database connection parameters for your reference database
-d	Generate code that favors small data size
–e *level*	Flag non-conforming SQL syntax as an error
-g	Do not display UltraLite warnings
–h *line-width*	Limit the maximum line length of output
-k	Include user
-m *version*	Specify the version name for generated synchronization scripts
–n	Line numbers
-o *operating-sys*	Target operating system: WIN32, WINNT, NETWARE, or UNIX
–p *project-name*	UltraLite project name
–q	Quiet mode—do not print banner
–s *string-len*	Maximum string length for the compiler
–w *level*	Flag non-conforming SQL syntax as a warning
–x	Change multi-byte SQL strings to escape sequences.
–z *sequence*	Specify collation sequence

See also	“Introduction” [*ASA Programming Guide*, page 136]
Description	The SQL preprocessor processes a C or C++ source file that contains embedded SQL, before the compiler is run. This preprocessor translates the SQL statements in the *input-file* into C/C++. It writes the result to the *output-file*. The normal extension for source files containing embedded SQL

is *sqc*. The default output filename is the *SQL-filename* base name with an extension of *c*. However, if the *SQL-filename* already has the *.c* extension, the default output extension is *.cc*.

When preprocessing files that are part of an UltraLite application, the SQL preprocessor requires access to an Adaptive Server Anywhere reference database. You must supply the connection parameters for the reference database using the **–c** option.

If you specify *no* project name, the SQL preprocessor also runs the UltraLite generator and appends additional code to the generated C/C++ source file. This code contains a C/C++ language description of your database schema as well as the implementation of the SQL statements in the application.

Customizing UltraLite generator operations The UltraLite analyzer provides hooks that you can use to customize the code generation process. These hooks are stored procedure names. If you supply stored procedures with the following names, the UltraLite analyzer invokes them before and after the analysis process:

- **sp_hook_ulgen_begin( )**
- **sp_hook_ulgen_end( )**

These hooks are defined in the reference database and are used only during the analyzer analysis phase. The hooks can be created as follows:

```
CREATE PROCEDURE sp_hook_ulgen_begin ()
BEGIN
// actions here
END
CREATE PROCEDURE sp_hook_ulgen_end ()
BEGIN
// actions here
END
```

Options

-c Required when preprocessing files that are part of an UltraLite application. The connection string must give the SQL preprocessor access to read and modify your reference database.

-d Generate code that reduces data space size. Data structures are reused and initialized at execution time before use. This increases code size.

-e This option flags any Embedded SQL that is not part of a specified set of SQL/92 as an error.

The allowed values of *level* and their meanings are as follows:

- **e** flag syntax that is not entry-level SQL/92 syntax
- **i** flag syntax that is not intermediate-level SQL/92 syntax

- **f** flag syntax that is not full-SQL/92 syntax
- **t** flag non-standard host variable types
- **u** flag features not supported by UltraLite
- **w** allow all supported syntax

-g Do not display warning specific to UltraLite code generation.

-h num Limits the maximum length of lines output by *sqlpp* to NUM characters. The continuation character is a backslash (\), and the minimum value of NUM is ten.

-k Notifies the preprocessor that the program to be compiled includes a user declaration of SQLCODE.

-m version Specify the version name for generated synchronization scripts. The generated synchronization scripts can be used in a MobiLink consolidated database for simple synchronization.

-n Generate line number information in the C file. This consists of *#line* directives in the appropriate places in the generated C code. If your compiler supports the *#line* directive, this option will make the compiler report errors on line numbers in the **SQL-filename**, as opposed to reporting errors on line numbers in the C/C++ output file. Also, the *#line* directives will indirectly be used by the source-level debugger so that you can debug while viewing the **SQL-filename**.

-o Specify the target operating system. Note that this option must match the operating system where you will run the program. A reference to a special symbol will be generated in your program. This symbol is defined in the interface library. If you use the wrong operating system specification or the wrong library, an error will be detected by the linker. The supported operating systems are:

- **WIN32** Microsoft Windows 95/98/Me and Windows CE
- **WINNT** Microsoft Windows NT/2000/XP
- **NETWARE** Novell NetWare
- **UNIX** UNIX

-p project-name Identifies the UltraLite project to which the embedded SQL files belong. Applies only when processing files that are part of an UltraLite application.

-q Operate quietly. Do not print the banner.

-s string-len Set the maximum size string that the preprocessor will put into the C file. Strings longer than this value will be initialized using a list of characters (*'a'*,*'b'*,*'c'*, etc). Most C compilers have a limit on the size of string literal they can handle. This option is used to set that upper limit. The default value is 500.

-w level This option flags any Embedded SQL that is not part of a specified set of SQL/92 as a warning.

The allowed values of *level* and their meanings are as follows:

- **e** flag syntax that is not entry-level SQL/92 syntax
- **i** flag syntax that is not intermediate-level SQL/92 syntax
- **f** flag syntax that is not full-SQL/92 syntax
- **t** flag non-standard host variable types
- **u** flag features not supported by UltraLite
- **w** allow all supported syntax

-x Change multi-byte strings to escape sequences so that they can pass through compilers.

-z sequence This option specifies the collation sequence or filename. For a listing of recommended collation sequences, type **dbinit –l** at the command prompt.

The UltraLite generator

Applies to

Static interfaces only.

Function

The UltraLite generator implements your application database and generates additional C/C++ or Java source files, which must be compiled and linked into your application.

Syntax

ulgen [*options*] [*project* [*output-filename*]]

Option	**Description**
-a	Uppercase SQL string names [Java]
-c "*keyword=value;...*"	Supply database connection parameters for your reference database
-e	Replace SQL strings with generated constants [Java]
-f *filename*	Specify output file name
-g	Do not display warnings
-i	Generate inner classes [Java]
-j *project-name*	Project name
-l *type*	Log the execution plan for each statement to a file. The type must be one of the following: ♦ xml ♦ short ♦ long
-m *version*	Specify the version name for generated synchronization scripts
-o *table-name,...*	Specify the order in which tables are uploaded during synchronization
-p *package-name*	Package name for generated classes [Java]
-q	Do not print the banner
-r *filename*	The file containing the trusted root certificates
-s *filename*	Generate a list of SQL strings in an interface definition [Java]

Option	Description
-t *target*	Target language. Must be one of the following: ♦ c ♦ c++ ♦ java
-u *pub-name*	The publication to use (C++ API only)
-v *pub-name*	The publication to use for synchronization
-x	Generate more and smaller C/C++ files.

Description

The UltraLite generator creates code that you compile and make part of an UltraLite application. Its output is based on the schema of the Adaptive Server Anywhere reference database and the specific SQL statements or tables that you use in your embedded SQL source files.

You must ensure that all your statements and tables are defined in the dbo.ul_statement table before running the generator. You do this as follows:

- ♦ In embedded SQL, run the SQL preprocessor on each file.
- ♦ In the C/C++ API and Java, add statements to the database using ul_add_statement, and/or define publications in the database.

In this table, statements are associated with projects. By specifying a project name on the generator command line, you determine which statements are included in your generated database.

You can include multiple projects, and also mix projects with a publication, on the generator command line. You must run the generator only once for each generated database.

If you do not specify an output file name, the generated code is written to a file with a name of *project*. It is recommended that you specify an output file name using the `-f` command line option.

Customizing UltraLite generator operations The UltraLite analyzer provides hooks that you can use to customize the code generation process. These hooks are stored procedure names. If you supply stored procedures with the following names, the UltraLite analyzer invokes them before and after the analysis process:

- ♦ **sp_hook_ulgen_begin( )**

♦ **sp_hook_ulgen_end( )**

These hooks are defined in the reference database and are used only during the analyzer analysis phase. The hooks can be created as follows:

```
CREATE PROCEDURE sp_hook_ulgen_begin ()
BEGIN
// actions here
END
CREATE PROCEDURE sp_hook_ulgen_end ()
BEGIN
// actions here
END
```

Options

project The project name determines the set of statements that are to be included in the generated database. For a more precise specification of the filename, use the `-j` option.

output-filename The name for the generated file, without extension. For a more precise specification of the filename, use the `-f` option.

In Java, this name is also the database name, which you must supply on connection.

-a If you are developing a Java application, the names of the SQL statements in the project are used as constants in your application. By convention, constants are upper case, with underscore characters between words. The `-a` option makes the names of SQL statements fit this convention by uppercasing the characters and inserting an underscore whenever an uppercase character in the original name is found if not already preceded by an underscore or an uppercase character. For example, a statement named `MyStatement` becomes `MY_STATEMENT`, and a statement named `AStatement` becomes `ASTATEMENT`.

The generated names have spaces and non-alphanumeric characters replaced with an underscore, regardless of whether –a is used.

-c connection-string The connection string must give the generator permission to read and modify your reference database. This parameter is required.

-e The SQL strings in the generated database are replaced by smaller, generated strings. This option is useful when you are trying to reduce the footprint of a database with a lot of statements.

-f filename This is the recommended way to specify the output file. Do not specify an extension.

-g Suppress the display of warning messages. Error messages are still displayed.

The UltraLite generator provides warnings to indicate that some generated code may, under some circumstances, cause problems. For example, it generates a warning for SQL statements that include temporary tables.

-i By default, generated classes are written as top-level non-public classes except for the main database class. If you use `-i`, the generated classes are written as inner classes. If you use this option, you must use a Java compiler that can correctly compile inner classes.

-j project-name This is the recommended way to specify the project. You can specify multiple projects using this option as follows:

```
ulgen -j project1 -j project2 ...
```

-l type Log the execution plan for queries in the application. These plans can be viewed in Interactive SQL. The types available are:

- **xml** Description in XML format. Use the Interactive SQL File ➤ Open command to display the plan.
- **short** Brief description of the plan in a file named <*statement*>*.txt*. The content is that generated by the EXPLANATION function
- **long** Detailed description of the plan in a file named <*statement*>*.txt*. The content is that generated by the PLAN function.

-m version Specify the version name for generated synchronization scripts. The generated synchronization scripts can be used in a MobiLink consolidated database for simple synchronization.

-o table-name,... Specify the order in which tables are uploaded during synchronization. This option can be used to avoid referential integrity errors during upload. Each table to be uploaded must be specified exactly once. The option cannot be used when there are circular foreign key relationships among the tables.

-p package-name A package name for generated files when generating Java output.

-q Do not display output messages.

-r filename The file containing the trusted root certificates used for secure synchronization using Certicom security software.

The generator embeds these trusted roots into the UltraLite application. When the application receives a certificate chain from a MobiLink synchronization server, it checks if its root is among the trusted roots, and only accepts a connection if it is.

The generator checks the expiry dates of all the certificates in the trusted root certificate file and issues the following warning for any certificate that expires in less than 6 months (180 days):

```
Warning:  Certificate will expire in %1 days"
```

The generator issues a `Certificate has expired` error for any certificate that has already expired.

☞ For more information, see "Synchronization parameters reference" on page 162, and "Transport-Layer Security" [*MobiLink Synchronization User's Guide*, page 337].

-s filename Generate an interface that contains the SQL statements as constants. This option is for use with Java only. The interface file has a format similar to the following example:

```
package com.sybase.test;
public interface EmpTestSQL {
    String EMPLOYEE = "select emp_fname, emp_lname
      from employee where emp_id = ?";
    String UPDATE_EMPLOYEE = "update employee
      set emp_fname = ?, emp_lname = ?
      where emp_id = ?";
}
```

Do not supply the *.java* extension in *filename*. The `-a` option controls the case of the statement names.

-t target Specifies the kind and extension of the generated file.

♦ If you are using Java, you must use a *target* of **java**. If you are using embedded SQL or the C++ API, you can use a *target* of either **c** or **c++**. Which one you choose decides the extension of the file name, and has nothing to do with whether you are using the C++ API or embedded SQL.

♦ If you specify **c++**, the following files are generated:
  - **filename.cpp** The code for the generated API.
  - **filename.h** A header file. You do not need to look at this file.
  - **filename.hpp** The C++ API definition for your application.

♦ If you specify a *target* of **c**, *filename.c* is generated.

-u pub-name If you are generating a C++ API for a publication, specify the publication name with the -u option.

-v pub-name Specifies a publication to synchronize. If you do not use publications to define which changes are to be synchronized, all changes are synchronized.

If columns or tables specified in publications are not referenced by SQL statements in your application, they are not included in the UltraLite database.

To specify multiple publications, repeat the `-v` option. For example:

```
ulgen -v pub1 -v pub2 ...
```

The maximum number of publications is 32.

☞ For more information, see "Synchronization for UltraLite Applications" on page 143.

-x This option is intended for use in situations where the file containing the generated code is too large for the C/C++ compiler to compile.

This option causes the UltraLite generator to produce more and smaller files. When `-x` is used, the UltraLite generater writes out one C/C++ file for the database and one for each SQL statement.

This option has no effect when generating Java code.

The UltraLite segment utility

Applies to

Embedded SQL and C++ API static development models together with the GCC PRC-Tools chain for the Palm Computing Platform.

Function

The UltraLite segment utility writes a set of segment identifiers in a definition file as required by the GCC PRC-Tools chain.

Syntax

dbulseg *generated-source-file definition-file app-name creator-id*

Option	**Description**
generated-source-file	The name of the source code file written by the UltraLite generator.
definition-file	The name of the definition file to be written out. It should end in the extension *.def*.
app-name	The name of the application.
creator-id	The application creator ID

Description

The GCC PRC-Tools suite requires a set of segment identifiers in a definition file. The **dbulseg** utility reads the UltraLite generated code (in file *generated-source-file*) and writes out the definition file *definition-file*.

The segment definition file also includes the Palm application name and application creator ID. These Palm-specific identifiers must be supplied in the command line.

Example

The command line included in the *build.bat* file that compiles the UltraLite CustDB sample application is as follows:

```
dbulseg custdb.c custdb.def CustDB Syb2
```

The resulting output file is as follows:

```
application { "CustDB" Syb2 }
multiple code { ULRT1 ULRT2 ULRT3 ULRT4 ULRT5 ULRT6
ULRT7 ULRT8 ULRT9 ULRT10 ULRT11 ULRT12 ULRT13 ULRT14
ULRT15 ULRT16 ULRT17 ULG512 ULG513 ULG514 ULG515 ULG516
ULG517 ULG518 ULG519 ULG520 ULG521 ULG522 ULG523 ULG524
ULG525 ULG526 ULG527 ULG528 ULG529 ULG530 ULG531 ULG532
ULG131 }
```

The file contents are on two lines: the second line is wrapped for display purposes.

The UltraLite Palm utility

Function

The UltraLite Palm utility is a Palm Computing Platform application that deletes all of the data stored in an UltraLite application's remote database.

Description

The UltraLite Palm utility is installed as the following file:

```
%ASANY9%\UltraLite\Palm\68k\ULUtil.prc
```

ULUtil is useful in deployments where devices are shared between different users. When a different user gets a device, they may want to clear out the previous user's data, to save storage space. Also, the previous user might want to clear out their data because it is confidential. Without **ULUtil**, the only way to clear out an application's data would be to delete and re-install the application.

You can set **ULUtil** to back up the Palm store to the PC on subsequent synchronization. You can use this feature to perform an initial synchronization and then backup the store which can be deployed on other devices so they do not need to perform an initial synchronization. The backup option is automatically turned off by the UltraLite runtime to prevent subsequent backups. If you explicitly want to require the database to be backed up on every synchronization, you must add the palm_allow_backup parameter in UL_STORE_PARMS.

☞ For more information, see "UL_STORE_PARMS macro" on page 216.

Once **ULUtil** is installed on the device, you can delete an UltraLite application's data as follows:

1. Switch to **ULUtil**.

2. Select an application from the list of UltraLite Applications.

3. Tap the Delete button.

On devices with expansion cards, ULUtil provides access to both file-based and record-based stores.

PART II

ULTRALITE SQL

This part describes the range of SQL available to UltraLite applications.

UltraLite components can construct queries and other SQL statements at runtime (dynamic SQL).

The static interfaces support a wider range of SQL, but the statements used in the application must be specified at compile time.

CHAPTER 6

SQL Language Elements

About this chapter

This chapter describes the building blocks of SQL statements and data management in UltraLite databases. These building blocks are common to all UltraLite databases.

Contents

Overview of SQL support in UltraLite

In UltraLite, both the data types available to represent data and the SQL features available to access that data depend on the development model you adopt.

If you use a static interface (embedded SQL, static C++ API, or static Java API), the range of SQL available is wider, but all statements used by the application must be specified at compile time. If you develop your application using an UltraLite component, dynamic SQL provides a narrower range of SQL, but the SQL statements can be constructed at runtime.

When an UltraLite program attempts to use a SQL statement or feature that is not supported in UltraLite, the SQL error message `Feature not available in UltraLite` is reported. Dynamic SQL may also return syntax errors.

- **Data types** UltraLite supports a subset of the data types available in Adaptive Server Anywhere.

  If you create a database from an Adaptive Server Anywhere reference database, you can use a wide range of data types. Those Adaptive Server Anywhere data types not supported in UltraLite are converted by the UltraLite generator into a smaller set of base types. If you create an UltraLite database using the Schema Painter, you are restricted to the smaller set of base types.

  For a listing of the UltraLite base types, see "Data types in UltraLite" on page 111.

  ☞ For a complete listing of Adaptive Server Anywhere data types, see "SQL Data Types" [*ASA SQL Reference*, page 51].

- **Identifiers** Identifiers are the names of database objects, such as columns and tables. UltraLite supports the same rules for identifiers as Adaptive Server Anywhere.

  For information about identifiers, see "Identifiers" [*ASA SQL Reference*, page 7].

- **Strings** Strings are used to hold character data in the database. UltraLite supports the same rules for strings as Adaptive Server Anywhere.

  If you create an UltraLite database from an Adaptive Server Anywhere reference database, the rules for strings are determined by the database options in effect in the reference database when the UltraLite generator is run. The QUOTED_IDENTIFIER option is particularly important in setting rules for strings. Dynamic SQL alwaus operates as if this option is ON (the default in Adaptive Server Anywhere).

☞ For information about strings, see “Strings” [*ASA SQL Reference*, page 8].

The results of comparisons on strings, and the sort order of strings, depends on both the case sensitivity of the database and the character set. These properties are set when the database is created.

For more information, see “UltraLite database characteristics” on page 29.

- **Functions** UltraLite supports the same range of functions as Adaptive Server Anywhere, with a few minor exceptions. The functions supported are the same for static interfaces such as embedded SQL as they are for dynamic SQL.

  For a list of supported functions, see “UltraLite SQL functions” on page 114.

- **Expressions** Expressions are formed by combining data, often in the form of column references, with operators or functions.

  Adaptive Server Anywhere provides a wide range of operators that it uses to form expressions. These operators are available if you develop your UltraLite application using a static interface (embedded SQL, static C++ API, or static Java API). One exception is that in Adaptive Server Anywhere you can use SQL variables to form expressions. You cannot use SQL variables (including global variables) in UltraLite applications. The @@identity global variable is an exception, and can be used within UltraLite applications.

  ☞ For information about expressions in Adaptive Server Anywhere, see “Expressions” [*ASA SQL Reference*, page 15].

  Dynamic SQL is more limited in the range of expressions it supports than is static SQL. For example, dynamic SQL does not support subqueries.

  ☞ For information about the expressions available in dynamic SQL, see “Dynamic SQL language elements” on page 128.

- **Search conditions** Search conditions or predicates are used in the WHERE clause, the HAVING clause, and the ON clause of SELECT statements.

  Dynamic SQL is more limited in the range of conditions that it supports than is static SQL. For example, dynamic SQL does not support EXISTS conditions.

  ☞ For information about search conditions available in dynamic SQL, see “Search conditions” on page 132.

Static interfaces have the entire range of conditions supported in Adaptive Server Anywhere available.

☞ For information about search conditions in Adaptive Server Anywhere, see "Search conditions" [*ASA SQL Reference*, page 22].

♦ **Statements** SQL statements are constructed from the building blocks listed above.

For a list of SQL statements available in dynamic SQL, see "Dynamic SQL statements" on page 134.

The following SQL statements can be used in static UltraLite applications:

- **Data Manipulation Language** SELECT, INSERT, UPDATE, and DELETE statements can be included. You can use placeholders in these statements that are filled in at runtime.

  For more information, see "Writing UltraLite SQL statements" on page 207.

- **TRUNCATE TABLE statement** You can use this statement to rapidly delete entire tables.

- **Transaction control** You can use COMMIT and ROLLBACK statements to provide transaction control within your UltraLite application.

- **START/STOP SYNCHRONIZATION DELETE statements** These statements are used to temporarily suspend synchronization of delete operations.

  For more information, see "Temporarily stopping synchronization of deletes" [*MobiLink Synchronization User's Guide*, page 193].

☞ For information on other UltraLite limitations, see "UltraLite database limitations" on page 44.

Data types in UltraLite

The following are the SQL data types supported in UltraLite databases.

If you create an UltraLite database from an Adaptive Server Anywhere reference database, you can use other data types, including user-defined data types, in the reference database. The UltraLite generator casts those data types into a data type supported in UltraLite databases. You cannot use user-defined data types that include DEFAULT values or CHECK constraints.

If you use dynamic SQL, or if you design an UltraLite database using the Schema Painter, you are limited to the use of the types listed here.

☞ For data types in Adaptive Server Anywhere, see “SQL Data Types” [*ASA SQL Reference*, page 51].

Data type	Remarks
`{CHAR|CHARACTER} [( max-length )]`	Character data of maximum length *max-length* characters. The maximum length is 2048 bytes. See “CHAR data type [Character]” [*ASA SQL Reference*, page 53]
`{VARCHAR	CHARACTER VARYING} [(max-length)]`	In UltraLite, VARCHAR is implemented identically to CHAR. In other databases, VARCHAR is used for variable-length character data of maximum length *max-length*. See “CHARACTER VARYING (VARCHAR) data type [Character]” [*ASA SQL Reference*, page 53]
`[UNSIGNED] BIGINT`	An integer requiring 8 bytes of storage. See “BIGINT data type [Numeric]” [*ASA SQL Reference*, page 56]
`{DECIMAL	DEC} [(precision [,scale])]`	A decimal number with *precision* total digits and with *scale* of the digits after the decimal point. See “DECIMAL data type [Numeric]” [*ASA SQL Reference*, page 57]
`NUMERIC [( precision [,scale] )]`	Same as DECIMAL. See “NUMERIC data type [Numeric]” [*ASA SQL Reference*, page 60]
`DOUBLE [PRECISION]`	A double-precision floating-point number. See “DOUBLE data type [Numeric]” [*ASA SQL Reference*, page 58]

Data type	Remarks
FLOAT [(*precision* **)**]	A floating point number, which may be single or double precision. See “FLOAT data type [Numeric]” [*ASA SQL Reference*, page 58]
[**UNSIGNED**] {**INT**|**INTEGER**}	An integer requiring 4 bytes of storage. See “INT or INTEGER data type [Numeric]” [*ASA SQL Reference*, page 59]
REAL	A single-precision floating-point number stored in 4 bytes. See “REAL data type [Numeric]” [*ASA SQL Reference*, page 61]
[**UNSIGNED**] **SMALLINT**	An integer requiring 2 bytes of storage. See “SMALLINT data type [Numeric]” [*ASA SQL Reference*, page 61]
[**UNSIGNED**] **TINYINT**	An integer requiring 1 byte of storage. See “TINYINT data type [Numeric]” [*ASA SQL Reference*, page 62]
DATE	A calendar date, such as a year, month and day. See “DATE data type [Date and Time]” [*ASA SQL Reference*, page 69]
TIME	The time of day, containing hour, minute, second and fraction of a second. See “TIME data type [Date and Time]” [*ASA SQL Reference*, page 70]
DATETIME	Identical to TIMESTAMP. See “DATETIME data type [Date and Time]” [*ASA SQL Reference*, page 70]
TIMESTAMP	The point in time, containing year, month, day, hour, minute, second and fraction of a second. See “TIMESTAMP data type [Date and Time]” [*ASA SQL Reference*, page 71]
BINARY [(*max-length* **)**]	Binary data of maximum length *max-length* bytes. The maximum length is 2048 bytes. See “BINARY data type [Binary]” [*ASA SQL Reference*, page 72]
VARBINARY [(*max-length* **)**]	Identical to BINARY. See “VARBINARY data type [BINARY]” [*ASA SQL Reference*, page 73]

Data type	Remarks
`LONG VARCHAR`	Arbitrary length character data. Conditions in SQL statements (such as in the WHERE clause) cannot operate on LONG VARCHAR columns. The only operations allowed on LONG VARCHAR columns are to insert, update, or delete them, or to include them in the *select-list* of a query. The maximum size of LONG VARCHAR values is 64 kb. See "LONG BINARY data type [BINARY]" [*ASA SQL Reference*, page 72]
`LONG BINARY`	Arbitrary length binary data. Conditions in SQL statements (such as in the WHERE clause) cannot operate on LONG VARCHAR columns. The only operations allowed on LONG VARCHAR columns are to insert, update, or delete them, or to include them in the *select-list* of a query. The maximum size of LONG BINARY values is 64 kb. See "LONG BINARY data type [BINARY]" [*ASA SQL Reference*, page 72]

UltraLite SQL functions

The following is a convenient reference for finding functions in dynamic SQL. Each function is listed, and the function type (numeric, character, and so on) is indicated next to it.

☞ For information about functions in Adaptive Server Anywhere, see “SQL Functions” [*ASA SQL Reference*, page 83].

Function	Remarks
ABS (*numeric-expression* **)**	See “ABS function [Numeric]” [*ASA SQL Reference*, page 97]
ACOS (*numeric-expression* **)**	See “ACOS function [Numeric]” [*ASA SQL Reference*, page 97]
ARGN (*integer-expression*, *expression* [, ...] **)**	See “ARGN function [Miscellaneous]” [*ASA SQL Reference*, page 98]
ASCII (*string-expression* **)**	See “ASCII function [String]” [*ASA SQL Reference*, page 98]
ASIN (*numeric-expression* **)**	See “ASIN function [Numeric]” [*ASA SQL Reference*, page 99]
ATAN (*numeric-expression* **)**	See “ATAN function [Numeric]” [*ASA SQL Reference*, page 99]
{ **ATN2** | **ATAN2** } **(** *numeric-expression1*, *numeric-expression2* **)**	See “ATN2 function [Numeric]” [*ASA SQL Reference*, page 100]
AVG (*numeric-expression* | **DISTINCT** *column-name* **)**	**DISTINCT** *column-name* cannot be used from dynamic SQL. See “AVG function [Aggregate]” [*ASA SQL Reference*, page 100]
BYTE_LENGTH (*string-expression* **)**	See “BYTE_LENGTH function [String]” [*ASA SQL Reference*, page 101]
BYTE_SUBSTR (*string-expression*, *start* [, *length*] **)**	See “BYTE_SUBSTR function [String]” [*ASA SQL Reference*, page 101]

Function	Remarks
`CAST ( expression AS data type )`	See “CAST function [Data type conversion]” [*ASA SQL Reference*, page 102]
`CEILING ( numeric-expression )`	See “CEILING function [Numeric]” [*ASA SQL Reference*, page 103]
`CHAR ( integer-expression )`	See “CHAR function [String]” [*ASA SQL Reference*, page 103]
`CHARINDEX ( string-expression1, string-expression2 )`	See “CHARINDEX function [String]” [*ASA SQL Reference*, page 104]
`CHAR_LENGTH ( string-expression )`	See “CHAR_LENGTH function [String]” [*ASA SQL Reference*, page 104]
`COALESCE ( expression, expression [ , ...] )`	See “COALESCE function [Miscellaneous]” [*ASA SQL Reference*, page 105]
`CONVERT ( data-type, expression [ , format-style ] )`	See “CONVERT function [Data type conversion]” [*ASA SQL Reference*, page 107]
`COS ( numeric-expression )`	See “COS function [Numeric]” [*ASA SQL Reference*, page 109]
`COT ( numeric-expression )`	See “COS function [Numeric]” [*ASA SQL Reference*, page 109]
`COUNT ( * | expression | DISTINCT { expression column-name } )`	**DISTINCT** *column-name* cannot be used from dynamic SQL. See “COUNT function [Aggregate]” [*ASA SQL Reference*, page 110]
`DATALENGTH ( expression )`	See “DATALENGTH function [System]” [*ASA SQL Reference*, page 113]
`DATE ( expression )`	See “DATE function [Date and time]” [*ASA SQL Reference*, page 113]

Function	Remarks
DATEADD (*date-part, numeric-expression, date-expression* **)**	See “DATEADD function [Date and time]” [*ASA SQL Reference*, page 114]
DATEDIFF (*date-part, date-expression1, date-expression2* **)**	See “DATEDIFF function [Date and time]” [*ASA SQL Reference*, page 114]
DATEFORMAT (*datetime-expression, string-expression* **)**	See “DATEFORMAT function [Date and time]” [*ASA SQL Reference*, page 116]
DATENAME (*date-part, date-expression* **)**	See “DATENAME function [Date and time]” [*ASA SQL Reference*, page 117]
DATEPART (*date-part, date-expression* **)**	See “DATEPART function [Date and time]” [*ASA SQL Reference*, page 117]
DATETIME (*expression* **)**	See “DATETIME function [Date and time]” [*ASA SQL Reference*, page 118]
DAY (*date-expression* **)**	See “DAY function [Date and time]” [*ASA SQL Reference*, page 118]
DAYNAME(*date-expression* **)**	See “DAYNAME function [Date and time]” [*ASA SQL Reference*, page 118]
DAYS ([*datetime-expression*,] *datetime-expression* **)**	See “DAYS function [Date and time]” [*ASA SQL Reference*, page 119]
DAYS (*datetime-expression, integer-expression* **)**	See “DAYS function [Date and time]” [*ASA SQL Reference*, page 119]
DEGREES (*numeric-expression* **)**	See “DEGREES function [Numeric]” [*ASA SQL Reference*, page 123]

Function	Remarks
`DIFFERENCE ( string-expression-1, string-expression-2 )`	See "DIFFERENCE function [String]" [*ASA SQL Reference*, page 123]
`DOW ( date-expression )`	See "DOW function [Date and time]" [*ASA SQL Reference*, page 123]
`EXP ( numeric-expression )`	See "EXP function [Numeric]" [*ASA SQL Reference*, page 131]
`FLOOR ( numeric-expression )`	See "FLOOR function [Numeric]" [*ASA SQL Reference*, page 133]
`GETDATE ()`	See "GETDATE function [Date and time]" [*ASA SQL Reference*, page 135]
`GREATER ( expression1, expression2 )`	See "GREATER function [Miscellaneous]" [*ASA SQL Reference*, page 138]
`HEXTOINT ( hexadecimal-string )`	See "HEXTOINT function [Data type conversion]" [*ASA SQL Reference*, page 139]
`HOUR ( datetime-expression )`	See "HOUR function [Date and time]" [*ASA SQL Reference*, page 139]
`HOURS ( [ datetime-expression, ] datetime-expression )`	See "HOUR function [Date and time]" [*ASA SQL Reference*, page 139]See "HOURS function [Date and time]" [*ASA SQL Reference*, page 140]
`HOURS ( datetime-expression, integer-expression )`	See "HOUR function [Date and time]" [*ASA SQL Reference*, page 139]See "HOURS function [Date and time]" [*ASA SQL Reference*, page 140]
`IFNULL ( expression-1, expression-2 [ , expression-3 ] )`	See "IFNULL function [Miscellaneous]" [*ASA SQL Reference*, page 142]
`INSERTSTR ( integer-expression, string-expression-1, string-expression-2 )`	See "INSERTSTR function [String]" [*ASA SQL Reference*, page 143]

Function	Remarks
INTTOHEX (*integer-expression* **)**	See “INTTOHEX function [Data type conversion]” [*ASA SQL Reference*, page 144]
ISDATE (*string* **)**	See “ISDATE function [Data type conversion]” [*ASA SQL Reference*, page 144]
ISNULL (*expression*, *expression* [, ...] **)**	See “ISDATE function [Data type conversion]” [*ASA SQL Reference*, page 144]
LCASE (*string-expression* **)**	See “LCASE function [String]” [*ASA SQL Reference*, page 146]
LEFT (*string-expression*, *para-expression*	See “LEFT function [String]” [*ASA SQL Reference*, page 147]
LENGTH (*string-expression* **)**	See “LENGTH function [String]” [*ASA SQL Reference*, page 147]
LESSER (*expression1*, *expression2* **)**	See “LESSER function [Miscellaneous]” [*ASA SQL Reference*, page 148]
LIST ({ *string-expression* | **DISTINCT** *column-name* } [, *delimiter-string*] [**ORDER BY** *order-by-expression*] **)**	**DISTINCT** *column-name* cannot be used from dynamic SQL. See “LIST function [Aggregate]” [*ASA SQL Reference*, page 148]
LOCATE (*string-expression-1*, *string-expression-2* [, *integer-expression*] **)**	See “LOCATE function [String]” [*ASA SQL Reference*, page 150]
LOG (*numeric-expression* **)**	See “LOG function [Numeric]” [*ASA SQL Reference*, page 151]
LOG10 (*numeric-expression* **)**	See “LOG10 function [Numeric]” [*ASA SQL Reference*, page 152]
LOWER (*string-expression* **)**	See “LOWER function [String]” [*ASA SQL Reference*, page 153]

Function	Remarks
LTRIM (*string-expression* **)**	See “LOWER function [String]” [*ASA SQL Reference*, page 153]
MAX (*expression* | **DISTINCT** *column name* **)**	See “MAX function [Aggregate]” [*ASA SQL Reference*, page 154]
MIN (*expression* | **DISTINCT** *column name* **)**	See “MIN function [Aggregate]” [*ASA SQL Reference*, page 154]
MINUTE (*datetime-expression* **)**	See “MINUTE function [Date and time]” [*ASA SQL Reference*, page 155]
MINUTES ([*datetime-expression*,] *datetime-expression* **)**	See “MINUTES function [Date and time]” [*ASA SQL Reference*, page 155]
MINUTES (*datetime-expression*, *integer-expression* **)**	See “MINUTES function [Date and time]” [*ASA SQL Reference*, page 155]
MOD (*dividend*, *divisor* **)**	See “MOD function [Numeric]” [*ASA SQL Reference*, page 157]
MONTH (*date-expression* **)**	See “MONTH function [Date and time]” [*ASA SQL Reference*, page 157]
MONTHNAME (*date-expression* **)**	See “MONTHNAME function [Date and time]” [*ASA SQL Reference*, page 157]
MONTHS ([*datetime-expression*,] *datetime-expression* **)**	See “MONTHNAME function [Date and time]” [*ASA SQL Reference*, page 157]
MONTHS (*datetime-expression*, *integer-expression* **)**	See “MONTHNAME function [Date and time]” [*ASA SQL Reference*, page 157]
NEWID( )	This function is not supported by the UltraLite static Java API. See “NEWID function [Miscellaneous]” [*ASA SQL Reference*, page 159]

Function	Remarks
NOW (*)	See "NOW function [Date and time]" [*ASA SQL Reference*, page 163]
NULLIF (*expression-1*, *expression-2* **)**	See "NULLIF function [Miscellaneous]" [*ASA SQL Reference*, page 163]
PATINDEX (*'%pattern%'*, *string_expression* **)**	See "PATINDEX function [String]" [*ASA SQL Reference*, page 168]
PI (*)	See "PI function [Numeric]" [*ASA SQL Reference*, page 169]
POWER (*numeric-expression-1*, *numeric-expression-2* **)**	See "POWER function [Numeric]" [*ASA SQL Reference*, page 171]
QUARTER (*date-expression* **)**	See "QUARTER function [Date and time]" [*ASA SQL Reference*, page 173]
RADIANS (*numeric-expression* **)**	See "RADIANS function [Numeric]" [*ASA SQL Reference*, page 174]
REMAINDER (*dividend*, *divisor* **)**	See "REMAINDER function [Numeric]" [*ASA SQL Reference*, page 175]
REPEAT (*string-expression*, *integer-expression* **)**	See "REPEAT function [String]" [*ASA SQL Reference*, page 175]
REPLACE (*original-string*, *search-string*, *replace-string* **)**	See "REPLACE function [String]" [*ASA SQL Reference*, page 176]
REPLICATE (*string-expression*, *integer-expression* **)**	See "REPLICATE function [String]" [*ASA SQL Reference*, page 176]
RIGHT (*string-expression*, *integer-expression* **)**	See "RIGHT function [String]" [*ASA SQL Reference*, page 179]

Function	Remarks
`ROUND (` *`numeric-expression,`* *`integer-expression`* **`)`**	See "ROUND function [Numeric]" [*ASA SQL Reference*, page 179]
`RTRIM (` *`string-expression`* **`)`**	See "RTRIM function [String]" [*ASA SQL Reference*, page 180]
`SECOND (` *`datetime-expression`* **`)`**	See "SECOND function [Date and time]" [*ASA SQL Reference*, page 180]
`SECONDS (` `[` *`datetime-expression,`* `]` *`datetime-expression`* **`)`**	See "SECONDS function [Date and time]" [*ASA SQL Reference*, page 180]
`SECONDS (` *`datetime-expression,`* *`integer-expression`* **`)`**	See "SECONDS function [Date and time]" [*ASA SQL Reference*, page 180]
`SIGN (` *`numeric-expression`* **`)`**	See "SIGN function [Numeric]" [*ASA SQL Reference*, page 182]
`SIMILAR (` *`string-expression-1,`* *`string-expression-2`* **`)`**	See "SIMILAR function [String]" [*ASA SQL Reference*, page 183]
`SIN (` *`numeric-expression`* **`)`**	See "SIN function [Numeric]" [*ASA SQL Reference*, page 183]
`SOUNDEX (` *`string-expression`* **`)`**	See "SOUNDEX function [String]" [*ASA SQL Reference*, page 187]
`SPACE (` *`integer-expression`* **`)`**	See "SPACE function [String]" [*ASA SQL Reference*, page 188]
`SQRT (` *`numeric-expression`* **`)`**	See "SQRT function [Numeric]" [*ASA SQL Reference*, page 189]
`STR (` *`numeric-expression`* `[,` *`length`* `[,` *`decimal`* `] ]` **`)`**	See "STR function [String]" [*ASA SQL Reference*, page 191]

Function	Remarks
`STRING ( string-expression [, ...] )`	See “STRING function [String]” [*ASA SQL Reference*, page 192]
`STRTOUUID ( string-expression )`	This function is not supported by the UltraLite static Java API. See “STRTOUUID function [STRING]” [*ASA SQL Reference*, page 192]
`STUFF ( string-expression1, start, length, string-expression2 )`	See “STUFF function [String]” [*ASA SQL Reference*, page 193]
`{ SUBSTRING | SUBSTR }( string-expression, start [, length ] )`	See “SUBSTRING function [String]” [*ASA SQL Reference*, page 193]
`SUM ( expression | DISTINCT column-name )`	**DISTINCT** *column-name* cannot be used from dynamic SQL. See “SUM function [Aggregate]” [*ASA SQL Reference*, page 194]
`TAN ( numeric-expression )`	See “TAN function [Numeric]” [*ASA SQL Reference*, page 195]
`TODAY ( * )`	See “TODAY function [Date and time]” [*ASA SQL Reference*, page 196]
`TRIM ( string-expression )`	See “TRIM function [String]” [*ASA SQL Reference*, page 197]
`"TRUNCATE" ( numeric-expression, integer-expression )`	See “TRUNCATE function [Numeric]” [*ASA SQL Reference*, page 197]
`TRUNCNUM ( numeric-expression, integer-expression )`	See “TRUNCNUM function [Numeric]” [*ASA SQL Reference*, page 198]
`UCASE ( string-expression )`	See “UCASE function [String]” [*ASA SQL Reference*, page 199]

Function	Remarks
UPPER (*string-expression* **)**	See “UPPER function [String]” [*ASA SQL Reference*, page 199]
UUIDTOSTR(*uuid-expression* **)**	This function is not supported by the UltraLite static Java API. See “UUIDTOSTR function [STRING]” [*ASA SQL Reference*, page 200]
WEEKS ([*datetime-expression*,] *datetime-expression* **)**	See “WEEKS function [Date and time]” [*ASA SQL Reference*, page 204]
WEEKS (*datetime-expression*, *integer-expression* **)**	See “WEEKS function [Date and time]” [*ASA SQL Reference*, page 204]
YEAR ([*datetime-expression*,] *datetime-expression* **)**	See “YEARS function [Date and time]” [*ASA SQL Reference*, page 210]
YEARS (*datetime-expression*, *integer-expression* **)**	See “YEARS function [Date and time]” [*ASA SQL Reference*, page 210]
YMD (*integer-expression*, *integer-expression*, *integer-expression* **)**	See “YMD function [Date and time]” [*ASA SQL Reference*, page 211]

CHAPTER 7

Dynamic SQL

About this chapter

Dynamic SQL is the version of SQL available to UltraLite components. This chapter describes the features of the dynamic SQL in UltraLite.

Dynamic SQL statements can be constructed at run time. This is in contrast to the static SQL available to embedded SQL, static C++ API, and static Java API applications, which must have all SQL statements specified at compile time.

Contents

Introduction to dynamic SQL

Structured Query Language (SQL) can be used by an application to perform a database task, such as retrieving information using a query or inserting a new row into a table. SQL is a relational database language standardized by the ANSI and ISO standards bodies. UltraLite dynamic SQL is a variant designed for use on small-footprint devices.

SQL statements are supplied as strings in function calls from the programming language you are using. UltraLite components provide functions for building and generating SQL statements. The programming interface delivers the SQL statement to the database. The database receives the statement and executes it, returning the required information (such as query results) back to the application.

Queries are one form of Data Manipulation Language used in SQL. In fact, the “Q” in “SQL” stands for query. You query, or retrieve, data from a database with a SELECT statement. A query produces a result set, which is a collection of rows that satisfy the query. The basic query operations in a relational system are projection, restriction, and join. The SELECT statement implements all of them.

A projection is a subset of the columns in a table. A restriction, also called selection, is a subset of the rows in a table, based on some conditions. For example, the following SELECT statement retrieves the names and prices of all products that cost more than $15:

```
SELECT name, unit_price
FROM product
WHERE unit_price > 15
```

This query uses both a projection, as shown in the SELECT clause, and a restriction, given in the WHERE clause.

You can do more with dynamic SQL than just query. It also includes statements that modify tables, the INSERT, UPDATE, and DELETE statements.

Availability

Dynamic SQL is the varant of SQL available for UltraLite components. UltraLite static interfaces use a different variant of SQL. The UltraLite components can use a table-based interface as well as dynamic SQL.

☞ For a comparison of these data access methods, see “Data access in UltraLite” on page 11.

Using dynamic SQL

Dynamic SQL can be used from UltraLite components, but not from static development models. The steps in executing dynamic SQL statements are common to all components:

1. Prepare the statement using a prepared statement method on the connection object. The name of the method varies slightly with the interface.

   Preparing a statement causes the character string representing the statement to be parsed and optimized (prepared) and returns an object representing the prepared statement. The optimization is necessarily less involved than that in Adaptive Server Anywhere.

2. Set the value of any parameters.

   Optionally, when the statement has input parameters (specified as âĂŸ?' in the statements), then your application can call methods on the prepared statement object to set the value of these parameters. Any parameters for which values are not set are set to NULL.

3. Execute the statement.

   If the statement is an INSERT, UPDATE, or DELETE, use the ExecuteStatement method. This method returns the number of rows modified by the statement.

   It the statement is a SELECT statement, use the ExecuteQuery method. This method returns an object that holds the query result set.

4. For queries, navigate the result set and access the values in the result set.
   - ♦ You can use methods on the result set object to set the position to different rows in the result set. Some examples are Next, Previous, First, Last, Relative, BeforeFirst, and AfterLast.
   - ♦ When the current position is at a row of the result set, the values of columns in the result set can be obtained by methods that get values. The names of the methods depend on the interface. The methods convert data to application data types automatically. For example, an integer result expression can automatically converted to a string if the result is assigned to a string variable.

5. For repeated execution of a prepared statement, repeat steps 2 through 4.

   The values for input variables persist after a prepared statement is executed. If you use a different value, you must reset the value of the parameter.

Dynamic SQL language elements

This section lists the expressions, operators, and search conditions supported by UltraLite dynamic SQL. These elements form the building blocks of the SQL statements listed in "Dynamic SQL statements" on page 134.

☞ For more information about SQL language elements in UltraLite, see "Overview of SQL support in UltraLite" on page 108.

Expressions

Expressions in UltraLite dynamic SQL are built from column names, constants, and operators. Expressions evaluate to a value, and so have data types associated with them.

Syntax

expression :
  constant
| *column-name*
| **-** *expression*
| *expression operator expression*
| **(** *expression* **)**
| *function-name* **(** *expression*, . . . **)**

See also

"SQL functions" on page 114

"Operators" on page 128

Aggregate expressions

An aggregate expression calculates a single value from a range of rows. For example, the following query computes the total payroll for employees in the employee table. In this query, `SUM( salary )` is an aggregate expression:

```
SELECT sum( salary )
FROM employee
```

An aggregate expression is one in which either an aggregate function is used, or in which one or more of the operands is an aggregate expression.

When a SELECT statement does not have a GROUP BY clause, the expressions in the *select-list* must be either all aggregate expressions or none of the expressions can be an aggregate expression. When a SELECT statement does have a GROUP BY clause, any non-aggregate expression in the select-list must appear in the GROUP BY list.

Operators

Operators are used to compare, combine, or modify expressions. Dynamic SQL supports the operators listed in this section. UltraLite static interfaces have access to all of the Adaptive Server Anywhere operators.

☞ For information about operators in Adaptive Server Anywhere, see "Operators" [*ASA SQL Reference*, page 10].

Binary comparison operators

The syntax for binary comparison conditions is as follows:

expression compare expression

where *compare* is a comparison operator. The following comparison operators are available:

Operator	Description
=	Equal to
[**NOT**] **LIKE**	A text comparison, possibly using regular expressions
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to
!=	Not equal to
<>	Not equal to
!>	Not greater than
!<	Not less than

- **Case sensitivity** Comparisons are carried out with the same attention to case as the database on which they are operating. By default, UltraLite databases are created as case insensitive.

- **NULL operators** Comparisons involving NULL expressions follow these rules:

  Two null values compare as equals. When exactly one of the operands being compared is NULL, the result is UNKNOWN. Thus, SQL comparisons produce one of three results (TRUE, FALSE, and UNKNOWN). Similarly, logical expressions (AND, OR, NOT) can also produce these results.

Arithmetic operators

expression + expression Addition. If either expression is NULL, the result is NULL.

expression – expression Subtraction. If either expression is NULL, the result is NULL.

–expression Negation. If the expression is NULL, the result is NULL.

expression * expression Multiplication. If either expression is NULL, the result is NULL.

expression / expression Division. If either expression is NULL or if the second expression is 0, the result is NULL.

expression % expression Modulo finds the integer remainder after a division involving two whole numbers. For example, 21 % 11 = 10 because 21 divided by 11 equals 1 with a remainder of 10.

String operators

expression || expression String concatenation (two vertical bars). If either string is NULL, it is treated as the empty string for concatenation.

expression + expression Alternative string concatenation. When using the + concatenation operator, you must ensure the operands are explicitly set to character data types rather than relying on implicit data conversion.

For example, the following query returns the integer value **579:**

```
SELECT 123 + 456
```

whereas the following query returns the character string **123456:**

```
SELECT '123' + '456'
```

You can use the CAST or CONVERT function to explicitly convert data types.

Bitwise operators

The following operators can be used on integer data types in UltraLite.

Operator	Description
&	bitwise AND
|	bitwise OR
^	bitwise exclusive OR
~	bitwise NOT

The bitwise operators &, | and ~ are not interchangeable with the logical operators AND, OR, and NOT. The bitwise operators operate on integer values using the bit representation of the values.

Example

For example, the following statement selects rows in which the correct bits are set.

```
SELECT *
FROM tableA
WHERE (options & 0x0101) <> 0
```

Operator precedence

The precedence of operators in expressions is as follows. The operators at the top of the list are evaluated before those at the bottom of the list.

1. Names, functions, constants
2. ()
3. unary operators (operators that require a single operand): +, -
4. ~
5. **&**, | , ^,
6. *****, /, **%**
7. +, **-**
8. ||
9. Comparisons: >, <, <>, !=, <=, >=, [NOT] BETWEEN, [NOT] IN, [NOT] LIKE
10. Comparisons: IS [NOT] TRUE, FALSE, UNKNOWN
11. NOT
12. AND
13. OR

When you use more than one operator in an expression, it is recommended that you make the order of operation explicit using parentheses rather than relying on an identical operator precedence in UltraLite.

Search conditions

Search conditions appear in the WHERE clause or the ON phrase in SQL queries. The following search conditions are supported in dynamic SQL.

Syntax

search-condition:
 expression compare expression
| *expression* **IS** [**NOT**] { **NULL** | **TRUE** | **FALSE** | **UNKNOWN** }
| *expression* [**NOT**] **BETWEEN** *expression* **AND** *expression*
| *expression* [**NOT**] **LIKE** *expression*
| *expression* [**NOT**] **IN** (*expression*, ... **)**
| **NOT** *search-condition*
| *search-condition* **AND** *search-condition*
| *search-condition* **OR** *search-condition*
| (*search-condition*)

Logical operators

Logical operators compare conditions (AND, OR, and NOT) or test the truth or NULL value nature of expressions (IS)

Conditions are combined using AND as follows:

condition1 **AND** *condition2*

The combined condition is TRUE if both conditions are TRUE, FALSE if either condition is FALSE, and UNKNOWN otherwise.

Conditions are combined using OR as follows:

condition1 **OR** *condition2*

The combined condition is TRUE if either condition is TRUE, FALSE if both conditions are FALSE, and UNKNOWN otherwise.

The syntax for the NOT operator is as follows:

NOT *condition*

The NOT condition is TRUE if *condition* is FALSE, FALSE if *condition* is TRUE, and UNKNOWN if *condition* is UNKNOWN.

The IS operator provides a means to test a logical value. The syntax for the IS operator is as follows:

expression **IS** [**NOT**] { *truth-value* | **NULL** }

The condition is TRUE if the *expression* evaluates to the supplied *truth-value*, which must be one of TRUE, FALSE, UNKNOWN, or NULL. Otherwise, the value is FALSE.

Dynamic SQL statements

The following are dynamic SQL statements you can use in UltraLite.

SELECT statement

Description

Use this statement to retrieve information from the database.

Syntax

SELECT [**DISTINCT**] [**FIRST** | **TOP** *n*] *select-list*
[**FROM** *table-expression*]
[**WHERE** *search-condition*]
[**GROUP BY** *group-by-expression*,...*group-by-expression*]
[**ORDER BY** *order-by-expression*,...*order-by-expression*]

table-expression :
table-name [[**AS**] *correlation-name*]
| *table-expression* { *join-operator table-expression* [**ON** *join-condition*]
,... }
| **(** *table-expression*, . . . **)**

join-operator :
, (ON condition not allowed)
| **CROSS JOIN** (ON condition not allowed)
| **INNER JOIN**
| **JOIN** (requires ON phrase)
| **LEFT OUTER JOIN**

order-by-expression :
{ *integer* | *expression* } [**ASC** | **DESC**]

Parameters

DISTINCT All (the default) returns all rows that satisfy the clauses of the SELECT statement. If DISTINCT is specified, duplicate output rows are eliminated. Many statements take significantly longer to execute when DISTINCT is specified, so you should reserve DISTINCT for cases where it is necessary.

FIRST or TOP You can explicitly retrieve only the first row of a query or the first *n* rows of a query. These keywords are principally for use with ORDER BY queries.

select-list The *select-list* is a list of expressions, separated by commas, specifying what will be retrieved from the database. An asterisk (*) means select all columns of all tables in the FROM clause. Subqueries are not allowed in the *select-list.*

An alias name can be specified following an expression in the *select-list* to represent that expression. The alias name can then be used elsewhere in the query, such as in the WHERE clause or ORDER BY clause.

FROM clause Rows are retrieved from the tables and views specified in the *table-expression*.

ON condition The ON condition is specified for a single join operation and indicates how the join is to create rows in the result set. A WHERE clause is used to restrict the rows in the result set, after potential rows have been created by a join. For INNER joins restricting with an ON or WHERE is equivalent. For OUTER joins, they are not equivalent.

WHERE clause This clause limits the rows that are selected from the tables named in the FROM clause. It can be used to restrict rows between multiple tables.

Although both the ON phrase (which is part of the FROM clause) and the WHERE clause restrict the rows in the result set, they differ in that the WHERE clause is applied at a later stage of query execution. The ON phrase is part of the join operation between tables, while the WHERE clause is applied after the join is complete. In some queries, a condition can be specified in a WHERE clause or in the ON phrase with the same net result, but in other cases the results differ. For example, for outer joins, a condition specified in a WHERE clause gives different results from the same condition specified in the ON phrase.

GROUP BY clause You can group by columns, alias names, or functions. The result of the query contains one row for each distinct set of values in the named columns, aliases, or functions. All NULL-containing rows are treated as a single set. The resulting rows are often referred to as groups since there is one row in the result for each group of rows from the table list. Aggregate functions can then be applied to these groups to get meaningful results.

A *group-by-expr* is a (non-aggregate) expression written exactly the same as one of the expressions in the *select-list*.

When GROUP BY is used, the *select-list* and ORDER BY expressions must not reference any identifier that is not named in the GROUP BY clause. The exception is that the *select-list* may contain aggregate functions.

ORDER BY clause This clause sorts the results of a query. Each item in the ORDER BY list can be labeled as ASC for ascending order (the default) or DESC for descending order. If the expression is an integer *n*, then the query results will be sorted by the *n*th item in the select list.

The only way to ensure that rows are returned in a particular order is to use ORDER BY. In the absence of an ORDER BY clause, UltraLite returns rows in whatever order is most efficient. This means that the appearance of result sets may vary depending on when you last accessed the row and other factors.

Usage — The SELECT statement is used for retrieving results from the database.

See also — "SELECT statement" [*ASA SQL Reference*, page 541]

Example — How many employees are there?

```
SELECT count(*)
FROM employee
```

INSERT statement

Description — Use this statement to insert a single row into a table (Syntax 1) or to insert the results from a SELECT statement into the table (Syntax 2).

Syntax 1 — **INSERT** [**INTO**] *table-name* [**(** *column-name*, . . . **)**]
VALUES (*expression*, . . . **)**

Syntax 2 — **INSERT** [**INTO**] *table-name* [**(** *column-name*, . . . **)**]
SELECT statement

Usage — The INSERT statement is used to add new rows to a database table.

Insert a single row with the specified expression values. If the optional list of column names is given, the values are inserted one for one into the specified columns. If the list of column names is not specified, the values are inserted into the table columns in the order they were created (the same order as retrieved with SELECT *). The row is inserted into the table at an arbitrary position.

If you specify column names, the columns from the select list are matched ordinally with the columns specified in the column list, or sequentially in the order in which the columns were created.

Character strings inserted into tables are always stored in the same case as they are entered, regardless of whether the database is case sensitive or not. Thus a string **Value** inserted into a table is always held in the database with an upper-case V and the remainder of the letters lower case. SELECT statements return the string as **Value**. If the database is not case sensitive, however, all comparisons make **Value** the same as **value**, **VALUE**, and so on. Further, if a single-column primary key already contains an entry **Value**, an INSERT of **value** is rejected, as it would make the primary key not unique.

Side effects — None.

Examples — Add an Eastern Sales department to the database.

```
INSERT
INTO department ( dept_id, dept_name )
VALUES ( 230, 'Eastern Sales' )
```

UPDATE statement

Description

Use this statement to modify existing rows in database tables.

Syntax

UPDATE *table-name*
SET *set-item*, . . .
[**WHERE** *search-condition*]

Parameters

UPDATE clause The UPDATE clause specifies the name of the table to be updated.

SET clause Each named column is set to the value of the expression on the right hand side of the equal sign. There are no restrictions on the *expression*. If the expression is a *column-name*, the old value is used.

WHERE clause If a WHERE clause is specified, only rows satisfying the search condition are updated. If no WHERE clause is specified, every row is updated.

Case sensitivity Character strings inserted into tables are always stored in the same case as they are entered, regardless of whether the database is case sensitive or not. A CHAR data type column updated with a string **Value** is always held in the database with an upper case V and the remainder of the letters lower case. SELECT statements return the string as **Value**. If the database is not case sensitive, however, all comparisons make **Value** the same as **value**, **VALUE**, and so on. Further, if a single-column primary key already contains an entry **Value**, an INSERT of **value** is rejected, as it would make the primary key not unique.

Usage

The UPDATE statement modifies values in a table.

Side effects

None.

See also

"INSERT statement" on page 136

"DELETE statement" on page 138

Example

Transfer employee Philip Chin (employee 129) from the sales department to the marketing department.

```
UPDATE employee
SET dept_id = 400
WHERE emp_id = 129
```

DELETE statement

Description

Use this statement to delete rows from the database.

Syntax

DELETE
[**FROM**] *table-name*
[**WHERE** *search-condition*]

Usage

The DELETE statement deletes all the rows that satisfy the search condition from the named table. If no WHERE clause is specified, all rows from the named table are deleted.

Side effects

None.

Example

Remove employee 105 from the database.

```
DELETE
FROM employee
WHERE emp_id = 105
```

Remove all data prior to 2000 from the fin_data table.

```
DELETE
FROM fin_data
WHERE year < 2000
```

Optimization of SELECT statements

The primary goal of optimization is choose indexes so that data can be accessed in a convenient order (GROUP BY and ORDER BY) thus avoiding temporary tables or to access only the pertinent subset of a table when joining two tables. As a development aid, you can use the GetPlan method to obtain a character string (usually called a plan) that summarizes how a prepared statement is to be executed. Thus, the statement

```
SELECT I.inv_no, I.name, T.quantity, T.prod_no
FROM Invoice I, Transactions T
WHERE I.inv_no = T.inv_no
```

produces a plan

```
join[scan(Invoice,0),index-scan(Transactions,1)]
```

The plan indicates that the join operation will be accomplished by reading all rows from the Invoice table (following index[0]) and then using the index[1] from the Transaction table to read only the row whose inv_no column matches.

In order to be usable on small devices, the optimization is not as extensive as that carried out in Adaptive Server Anywhere. The optimizer attempts to find sub-expressions in which a column (occurring by itself) is compared with expressions that involve only constants or columns from tables that occurred earlier in the FROM clause. Sometimes, you can rewrite a SELECT statement into a form that is equivalent but which executes more quickly. Often, this is known as syntax-directed optimization.

Part III

Synchronizing UltraLite Applications

This part describes how UltraLite databases can synchronize information with other databases, and how to add synchronization to an UltraLite application.

CHAPTER 8

Synchronization for UltraLite Applications

About this chapter

This chapter introduces MobiLink synchronization for users of UltraLite applications. It provides information on designing UltraLite databases to make the most of synchronization. This chapter is intended for users who have SQL Anywhere Studio.

Contents

Introduction

UltraLite developers who also have SQL Anywhere Studio can synchronize the data in UltraLite databases with a central consolidated database. This database may be a desktop database for personal applications, or a multi-user database for shared enterprise data. Synchronization requires the MobiLink synchronization software included with SQL Anywhere Studio.

This chapter describes general aspects of synchronization. Particulars of implementing synchronization can be found in the individual book for each component or static development model. For a full description of synchronization, see the *MobiLink Synchronization User's Guide*.

You can also find a working example of synchronization in the CustDB sample application.

MobiLink synchronization features

Many mobile and embedded computing applications are integrated into an information infrastructure. They require data to be uploaded to and downloaded from a **consolidated database**. This bi-directional sharing of information is **synchronization**.

MobiLink synchronization technology, included in SQL Anywhere Studio along with UltraLite, is designed to work with industry standard SQL database-management systems from Sybase and other vendors. UltraLite automatically keeps track of changes made to the UltraLite database between each synchronization with the consolidated database. When the UltraLite database is synchronized, all changes since the previous synchronization are uploaded.

Subset of the consolidated database

Mobile and embedded databases generally cannot contain all the data that exists in the consolidated database. In practice, however, the only data you need locally is that used by the particular application you wish to make mobile. UltraLite provides the ability to take such a piece of a database, and keep it synchronized with the consolidated database.

The tables in each UltraLite database can have a subset of the rows and columns in the central database. For example, a customer table might contain over 100 columns and 100 000 rows in the consolidated database, but the UltraLite database may only require 4 columns and 1000 rows. MobiLink allows you to define the exact subset to be downloaded to each remote database.

Flexibility

MobiLink synchronization is flexible. You define the subset of data using the native SQL dialect of the consolidated database-management system, Java, or a .NET programming language. Tables in the UltraLite database can correspond to tables in the consolidated database, but you can also populate an UltraLite table from a consolidated table with a different name, or from a join of one or more tables.

Conflict resolution

Mobile and embedded databases frequently share common data. They also must allow updates to the same data. When two or more remote databases simultaneously update the same row, the conflict cannot be prevented. It must be detected and resolved when the changes are uploaded to the central database. MobiLink synchronization automatically detects these conflicts. The conflict resolution logic is defined in the native SQL dialect of the consolidated database, in Java, or in a .NET programming language.

The MobiLink synchronization server

An UltraLite application synchronizes with a central, consolidated database through the **MobiLink synchronization server**. This server provides an interface between the UltraLite application and the database server.

You control the synchronization process using **synchronization scripts**. These scripts may be SQL statements or procedures written in the native language of the consolidated DBMS, or they may be Java classes. For example, you can use a SELECT statement to identify the columns and tables in the consolidated database that correspond to each column of a row to be downloaded to a table in your UltraLite application. Each script controls a particular event during the synchronization process.

Synchronization streams

UltraLite databases can synchronize with a MobiLink synchronization server over one of a set of synchronization streams, including TCP/IP, HTTP, and HTTPS. ActiveSync synchronization is available for Windows CE applications under some development models. HotSync is available for Palm OS applications.

Each synchronization stream has a set of appropriate stream parameters. These parameters set required values for the stream, such as the location of the MobiLink synchronization server, and network-specific control parameters.

Supported synchronization streams

The following synchronization streams are supported:

Component	**TCP/IP**	**HTTP**	**HTTPS**	**ActiveSync (Windows CE only)**	**HotSync (Palm OS only)**
UltraLite ActiveX	✔	✔	✔		
UltraLite for MobileVB	✔	✔	✔[1]		✔

[1]Must be separately ordered

Component	TCP/IP	HTTP	HTTPS	ActiveSync (Windows CE only)	HotSync (Palm OS only)
Native UltraLite for Java	✔	✔	✔	✔	
UltraLite.NET	✔	✔	✔	✔	
UltraLite C++	✔	✔	✔	✔	✔
Embedded SQL	✔	✔	✔	✔	✔
Static C++ API	✔	✔	✔	✔	✔
Static Java API	✔[2]	✔	✔		

Secure synchronization

To synchronize using encrypted synchronization (HTTPS) or to use encryption over TCP/IP you must obtain the separately-licensable security option. To order this option, see the card in your SQL Anywhere Studio package or see *http://www.sybase.com/detail?id=1015780.*

Adding synchronization to your UltraLite application

❖ To add synchronization to an UltraLite application

1. Prepare the synchronization stream.

   Select a synchronization stream and set other synchronization parameters as required for that stream.

   ☞ For more information, see "Selecting a synchronization stream" on page 147.

2. Call the synchronization function.

   The synchronization function depends on the development model you are using and the synchronization stream the application is using.

   ☞ For more information, see "Calling the synchronization function" on page 148.

Secure synchronization

To synchronize using encrypted synchronization (HTTPS) you must obtain the separately-licensable security option. To order this option, see the card in your SQL Anywhere Studio package or see *http://www.sybase.com/detail?id=1015780.*

[2] Use separate streams for secure synchronization

Selecting a synchronization stream

Each synchronization stream has a set of parameters that govern its behavior. You should set these synchronization stream parameters when you select a synchronization stream.

The way to select a synchronization stream and its associated synchronization stream parameters depends on the particular UltraLite development model you are using.

♦ For UltraLite for MobileVB and UltraLite ActiveX, the synchronization stream is one of the synchonization parameters set in the Stream property of the ULSyncParms object. The stream parameters are provided as a set of keyword-value pairs in the StreamParms property.

  ☞ For more information, see “SyncParms” [*UltraLite for MobileVB User’s Guide*, page 135], and “SyncParms” [*UltraLite ActiveX User’s Guide*, page 153].

♦ For Native UltraLite for Java applications, the synchronization stream is set by the **setStream** method of the **SyncParms** object.

  ☞ For more information, see **ianywhere.native_ultralite.SyncParms** in the API Reference.

♦ For embedded SQL and static C++ API applications, the synchronization stream is set in the stream member of the ul_synch_info structure. The synchronization stream parameters are supplied in the stream_parms member of the ul_synch_info structure, as a string. The following code is an example for TCP/IP synchronization:

```
ul_synch_info info;
...
info.stream = ULSocketStream();
info.stream_parms = UL_TEXT( "host=myserver" );
```

  ☞ For more information, see the following:

  • “stream synchronization parameter” [*UltraLite Embedded SQL User’s Guide*, page 146]
  • “stream_parms synchronization parameter” [*UltraLite Embedded SQL User’s Guide*, page 149]
  • “stream synchronization parameter” [*UltraLite Static C++ User’s Guide*, page 132]
  • “stream_parms synchronization parameter” [*UltraLite Static C++ User’s Guide*, page 135]

♦ For static Java applications, the synchronization stream parameters are supplied using the **setStreamParms** method. The following example illustrates how to call the method:

```
UlSynchOptions synch_options = new UlSynchOptions();
synch_opts.setStream( new UlSocketStream() );
synch_opts.setStreamParms( "host=myserver;port=2439" );
```

☞ For more information, see “stream synchronization parameter” [*UltraLite Static Java User’s Guide,* page 80] and “stream_parms synchronization parameter” [*UltraLite Static Java User’s Guide,* page 83].

Calling the synchronization function

In order to synchronize data with a MobiLink synchronization server, UltraLite applications call a synchronization function. The particular function depends on the development model you are using and on the synchronization stream you have selected.

It is helpful to distinguish the following kinds of synchronization streams:

♦ **Externally-initiated synchronization streams** ActiveSync and HotSync synchronization are initiated by an external application.

For information about calling HotSync synchronization, see the following:

- **MobileVB** See “Synchronization” [*UltraLite for MobileVB User’s Guide,* page 72].
- **Embedded SQL** See “Adding HotSync synchronization to Palm applications” [*UltraLite Embedded SQL User’s Guide,* page 81].
- **Static C++ API** See “Adding HotSync synchronization to Palm applications” [*UltraLite Static C++ User’s Guide,* page 54].

♦ **Direct synchronization streams** TCP/IP, HTTP, and HTTPS synchronization streams are initiated directly from the UltraLite.

For information about calling the synchronization function for these streams, see the following:

- **UltraLite for MobileVB** See “Synchronize method” [*UltraLite for MobileVB User’s Guide,* page 96].
- **UltraLite ActiveX** See “Synchronize method” [*UltraLite ActiveX User’s Guide,* page 116].
- **Native UltraLite for Java** See **ianywhere.native_ultralite.Connection.synchronize** in the API Reference.
- **UltraLite.NET** See Synchronize in the UltraLite.NET API Reference.

- **Embedded SQL** See “ULSynchronize function” [*UltraLite Embedded SQL User’s Guide*, page 136].
- **Static C++ API** See “Synchronize method” [*UltraLite Static C++ User’s Guide*, page 87].
- **Static Java API** See “synchronize method” [*UltraLite Static Java User’s Guide*, page 63].

Designing synchronization for your UltraLite database

UltraLite applications use MobiLink synchronization technology to share data with a consolidated database and integrate into an enterprise information system.

By default, when you add synchronization to your database, all data is synchronized. This section describes how to maintain unique primary keys across multiple databases, as well as advanced aspects of UltraLite synchronization design, including non-synchronizing tables, separate data sets for synchronization such as high-priority synchronization, and read-only tables.

See also

This section describes how certain features of MobiLink affect the design decisions you make for UltraLite applications. For a full description of MobiLink synchronization, see the *MobiLink Synchronization User's Guide*. In particular:

- ♦ For more information on synchronization, see "Introducing MobiLink Synchronization" [*MobiLink Synchronization User's Guide*, page 3].
- ♦ For an introduction to synchronization concepts, see "Synchronization Basics" [*MobiLink Synchronization User's Guide*, page 7].
- ♦ For information about synchronization techniques, see "Synchronization Techniques" [*MobiLink Synchronization User's Guide*, page 69].

Adding synchronization

Adding MobiLink synchronization to an UltraLite application is a matter of supplying arguments to a function call. The details of the call, and the synchronization options available to your application, depend on your target platform.

☞ For more information, see "Synchronization for UltraLite Applications" on page 143.

Maintaining primary key uniqueness

You can declare the default value of a column in a reference database to be of type GLOBAL AUTOINCREMENT. You can use this default for any column in which you want to maintain unique values, but it is particularly useful for primary keys. This feature simplifies the task of generating unique values in setups where data is being replicated among multiple databases, typically by MobiLink synchronization.

When you specify default global autoincrement, the domain of values for that column is partitioned. Each partition contains the same number of values. For example, if you set the partition size for an integer column in a database to 1000, one partition extends from 1001 to 2000, the next from 2001 to 3000, and so on.

☞ For information on declaring columns as global autoincrement in your reference database, see "Declaring default global autoincrement columns" on page 151.

To use global autoincrement columns in your UltraLite database, you must first assign each copy of the database a unique global database identification number. UltraLite then supplies default values for the column only from the partition uniquely identified by that database's number. For example, if you assigned a database in the above example the identity number 1, the default values in that database would be chosen in the range 1001–2000. Another copy of the database, assigned the identification number 2, would supply default value for the same column in the range 2001–3000.

☞ For information on assigning global database identification numbers, see "Setting the global database identifier" on page 152.

☞ For information on using global autoincrement values in Adaptive Server Anywhere remote databases, see "Maintaining unique primary keys using global autoincrement" [*MobiLink Synchronization User's Guide*, page 82].

Declaring default global autoincrement columns

You declare default column values in the Adaptive Server Anywhere reference database. When you build your UltraLite application, your UltraLite database inherits the default column value. You can set default values in your reference database by selecting the column properties in Sybase Central, or by including the DEFAULT GLOBAL AUTOINCREMENT phrase in a TABLE or ALTER TABLE statement.

Optionally, the partition size can be specified in parentheses immediately following the AUTOINCREMENT keyword. The partition size may be any positive integer, although the partition size is generally chosen so that the supply of numbers within any one partition will rarely, if ever, be exhausted.

For columns of type INT or UNSIGNED INT, the default partition size is $2^{16} = 65536$; for columns of other types the default partition size is $2^{32} = 4294967296$. Since these defaults may be inappropriate it is best to specify the partition size explicitly.

For example, the following statement creates a simple reference table with two columns: an integer that holds a customer identification number and a character string that holds the customer's name.

```
CREATE TABLE customer (
   id   INT          DEFAULT GLOBAL AUTOINCREMENT (5000),
   name VARCHAR(128) NOT NULL,
   PRIMARY KEY (id)
)
```

In the above example, the chosen partition size is 5000.

Default partition sizes for some data types are different in UltraLite applications than in Adaptive Server Anywhere databases. Declare the partition size explicitly if you wish the reference database to behave in the same manner as your UltraLite application.

☞ For more information on GLOBAL AUTOINCREMENT, see "CREATE TABLE statement" [*ASA SQL Reference*, page 361].

Setting the global database identifier

When deploying an application, you must assign a different identification number to each database. You can accomplish the task of creating and distributing the identification numbers by a variety of means. One method is to place the values in a table and download the correct row to each database based on some other unique property, such as user name.

The method of setting this identification number varies according to the programming interface you are using.

- **UltraLite for MobileVB** See "Properties" [*UltraLite for MobileVB User's Guide*, page 89].
- **UltraLite ActiveX** See "Properties" [*UltraLite ActiveX User's Guide*, page 111].
- **Native UltraLite for Java** See **ianywhere.native_ultralite.Connection.databaseID** in the API Reference.
- **UltraLite.NET** See Connection in the UltraLite.NET API Reference.
- **Embedded SQL** See "ULSetDatabaseID function" [*UltraLite Embedded SQL User's Guide*, page 131].
- **Static C++ API** See "SetDatabaseID method" [*UltraLite Static C++ User's Guide*, page 86].
- **Static Java API** See "setDatabaseID method" [*UltraLite Static Java User's Guide*, page 62].

How default values are chosen

The global database identifier in each deployed UltraLite application must be set to a unique, non-negative integer before default values can be assigned. These identification numbers uniquely identify the databases.

☞ For information, see "Setting the global database identifier" on page 152.

The range of default values for a particular database is $pn + 1$ to $p(n + 1)$, where p is the partition size and n is the global database identification number. For example, if the partition size is 1000 and the global database identification number is set to 3, then the range is from 3001 to 4000.

UltraLite applications choose default values by applying the following rules:

- ♦ If the column contains no values in the current partition, the first default value is $pn + 1$.
- ♦ If the column contains values in the current partition, but all are less than $p(n + 1)$, the next default value will be one greater than the previous maximum value in this range.
- ♦ Default column values are not affect by values in the column outside of the current partition; that is, by numbers less than $pn + 1$ or greater than $p(n + 1)$. Such values may be present if they have been replicated from another database via MobiLink synchronization.

Caution
Column values downloaded via MobiLink synchronization do not update the default value counter. Thus, an error can occur should one MobiLink client insert a value into another client's partition. To avoid this problem, ensure that each copy of your UltraLite application inserts values only in its own partition.

If the global database identification number is set to the default value of 2147483647, a NULL value is inserted into the column. Should NULL values not be permitted, the attempt to insert the row causes an error. This situation arises, for example, if the column is contained in the table's primary key.

Because the global database identification number cannot be set to negative values, the values chosen are always positive. The maximum identification number is restricted only by the column data type and the partition size.

Null default values are also generated when the supply of values within the partition has been exhausted. In this case, a new global database identification number should be assigned to the database to allow default values to be chosen from another partition. Attempting to insert the NULL value causes an error if the column does not permit nulls.

Should the values in a particular partition become exhausted, you can assign a new database identification number to that database. You can assign new database id numbers in any convenient manner. However, one possible technique is to maintain a pool of unused database id values. This pool is maintained in the same manner as a pool of primary keys.

☞ For information on determining whether the range of default values is becoming exhausted, see "Detecting the number of available default values" on page 154.

☞ For information on maintaining primary key uniqueness using explicit primary key pools, see "Maintaining unique primary keys" [*MobiLink Synchronization User's Guide,* page 81].

Determining the most recently assigned value

You can retrieve the value that was chosen during the most recently insert operation. Since these values are often used for primary keys, knowing the generated value may let you more easily insert rows that reference the primary key of the first row.

From embedded SQL, you can obtain the most recently assigned global autoincrement default value using the following statement.

```
select @@identity
```

From the C++ API, the value is available using the **GetLastIdentity()** method on the **ULConnection** object

The returned value is an unsigned 64-bit integer, database data type UNSIGNED BIGINT. Since this statement only allows you to determine the most recently assigned default value, you should retrieve this value soon after executing the insert statement to avoid spurious results.

Occasionally, a single insert statement may include more than one column of type global autoincrement. In this case, the return value is one of the generated default values, but there is no reliable means to determine which one. For this reason, you should design your database and write your insert statements so as to avoid this situation.

Detecting the number of available default values

Default values are chosen from the partition identified by the global database identification number until the maximum value is reached. When this state has been reached or is imminent, you must assign the database a new identification number.

The programming interfaces provide means of obtaining the proportion of numbers that have been used. The return value is a short in the range 0–100 that represents the percent of values used thus far. For example, a value of 99 indicates that very few unused values remain and the database should be assigned a new identification number.

The method of setting this identification number varies according to the programming interface you are using.

♦ **UltraLite for MobileVB** See “Properties” [*UltraLite for MobileVB User’s Guide,* page 89].

♦ **UltraLite ActiveX** See “Properties” [*UltraLite ActiveX User’s Guide,* page 111].

♦ **Native UltraLite for Java** See **ianywhere.native_ultralite.Connection.databaseID** in the API Reference.

♦ **UltraLite.NET** See Connection in the UltraLite.NET API Reference.

♦ **Embedded SQL** See “ULGlobalAutoincUsage function” [*UltraLite Embedded SQL User’s Guide,* page 118].

♦ **Static C++ API** See “GlobalAutoincUsage method” [*UltraLite Static C++ User’s Guide,* page 80].

♦ **Static Java API** See “globalAutoincUsage method” [*UltraLite Static Java User’s Guide,* page 61].

Including non-synchronizing tables in UltraLite databases

By default, all tables in an UltraLite database are synchronized to the consolidated database. You can include tables in your UltraLite database that are excluded from synchronization, but you must explicitly identify these tables when you create your reference database.

Tables with names ending in nosync are excluded from synchronization. You can use these tables for persistent data that is not related to the consolidated database. Other than being excluded from synchronization, you can use these tables in exactly the same way as other tables in the UltraLite database.

You can alternatively use publications to achieve the same effect. For more information, see “Designing sets of data to synchronize separately” on page 156.

Designing sets of data to synchronize separately

The schema of an UltraLite database is defined by the queries included in the application. You can add publications to the reference database to define sets of data that can be synchronized separately. If you do not use publications to define which changes are to be synchronized, all changes are synchronized.

Publications are used for several purposes in SQL Anywhere. A publication consists of a set of articles. In general, each article can be a whole table, or can define a subset of the data in a table.

Articles defined for UltraLite applications can use row subsets by supplying a WHERE clause, but cannot use column subsets or the SUBSCRIBE BY clause. Articles in UltraLite publications governing HotSync synchronization cannot use a WHERE clause.

❖ To synchronize subsets of data from an UltraLite database

1. Create publications representing the data you wish to synchronize.

   ☞ For more information, see "Creating publications for UltraLite databases" on page 157.

2. Run the UltraLite generator, specifying the publications on the -v command-line option.

   ☞ For more information, see "The UltraLite generator" on page 96.

3. When calling the synchronization function, specify the publication.

   If you specify no publication, all changes to the database are synchronized. If you specify one or more publications, only changes that fall within one or more of the listed publications are synchronized.

   ☞ For more information, see the following:

   ♦ **MobileVB** See "SyncParms" [*UltraLite for MobileVB User's Guide*, page 135].

   ♦ **ActiveX** See "SyncParms" [*UltraLite ActiveX User's Guide*, page 153].

   ♦ **Native UltraLite for Java** See **ianywhere.native_ultralite.SyncParms** in the API Reference.

   ♦ **Embedded SQL** See "publication synchronization parameter" [*UltraLite Embedded SQL User's Guide*, page 144].

   ♦ **Static C++ API** See "publication synchronization parameter" [*UltraLite Static C++ User's Guide*, page 130].

   ♦ **Static Java API** See "publication synchronization parameter" [*UltraLite Static Java User's Guide*, page 77].

Creating publications for UltraLite databases

Components

Publications can be added to the UltraLite database using the schema painter, the ULXML utility, or from a reference database.

❖ To publish data from an UltraLite database (Schema Painter)

1. Connect to the UltraLite database.
2. In the left pane, open the Synchronization folder.
3. Double-click Add Publication.
4. Specify a set of tables to include in the publication.
5. Click OK to save the changes.

Reference database

For UltraLite synchronization, each article in a publication may include either a complete table or may include a WHERE clause.

❖ To publish data from an UltraLite reference database (Sybase Central)

1. Connect to the database as a user with DBA authority.
2. Open the Publications folder and double-click Add Publication.
3. Type a name for the new publication. Click Next.
4. On the Tables tab, select a table from the list of Matching Tables. Click Add. The table appears in the list of Selected Tables on the right.
5. Add additional tables as required. The order of the tables is not important.
6. If necessary, click the Where tab to specify the rows to be included in the publication. You cannot specify column subsets. If you are using HotSync synchronization, do not specify a WHERE clause.
7. Click Finish.

❖ **To publish data from an UltraLite reference database (SQL)**

1. Connect to the database as a user with DBA authority.

2. Execute a CREATE PUBLICATION statement that specifies the name of the new publication and the table you want to publish.

   ☞ For more information, see “CREATE PUBLICATION statement” [*ASA SQL Reference*, page 334].

Synchronizing high-priority changes

Publications permit the synchronization of specific portions of your UltraLite database. You can combine publications with upload-only or download-only synchronization to synchronize high-priority changes efficiently. Both upload-only and download-only synchronization are less time-consuming than two-way synchronization.

☞ For more information, see “Creating publications for UltraLite databases” on page 157, and the following:

- **MobileVB** See “SyncParms” [*UltraLite for MobileVB User’s Guide*, page 135].
- **ActiveX** See “SyncParms” [*UltraLite ActiveX User’s Guide*, page 153].
- **Native UltraLite for Java** See **ianywhere.native_ultralite.SyncParms** in the API Reference.
- **Embedded SQL** See “upload_only synchronization parameter” [*UltraLite Embedded SQL User’s Guide*, page 150].
- **Static C++ API** See “upload_only synchronization parameter” [*UltraLite Static C++ User’s Guide*, page 136].
- **Static Java API** See “upload_only synchronization parameter” [*UltraLite Static Java User’s Guide*, page 84].

Including read-only tables in an UltraLite database

Some applications include tables in the UltraLite database that are not updated locally. Price lists and company policies are two examples. You can synchronize these tables efficiently by including them in a publication, and synchronizing the publication using download-only synchronization. Download-only synchronization is less time-consuming than a two-way synchronization, as no data is uploaded.

To use download-only synchronization, you must ensure that the data is not changed locally. If any data is changed locally, synchronization fails with a SQLE_DOWNLOAD_CONFLICT error.

Unlike for two-way synchronization, you do not have to commit all changes to the UltraLite database before download-only synchronization. Uncommitted changes to tables not involved in synchronization are not uploaded, and so there incomplete transactions do not cause problems.

☞ For information on download-only synchronization, see the following:

- ♦ **MobileVB** See “SyncParms” [*UltraLite for MobileVB User’s Guide,* page 135].
- ♦ **ActiveX** See “SyncParms” [*UltraLite ActiveX User’s Guide,* page 153].
- ♦ **Native UltraLite for Java** See **ianywhere.native_ultralite.SyncParms** in the API Reference.
- ♦ **Embedded SQL** See “download_only synchronization parameter” [*UltraLite Embedded SQL User’s Guide,* page 141].
- ♦ **Static C++ API** See “download_only synchronization parameter” [*UltraLite Static C++ User’s Guide,* page 127].
- ♦ **Static Java API** See “download_only synchronization parameter” [*UltraLite Static Java User’s Guide,* page 73].

Using client-specific data to control synchronization

Some UltraLite applications require client-specific data that control synchronization, but which are not needed on the consolidated database. For example, you may wish your UltraLite applications to indicate which of a number of channels or topics they are interested in, and use this information to download the appropriate rows.

If you create a table in your UltraLite database with a name ending in allsync, all rows of that table are synchronized at each synchronization, whether or not they have been changed since the last synchronization.

You can store user-specific or client-specific data in allsync tables. If you upload the data in the table to a temporary table in the consolidated database on synchronization, you can use the data to control synchronization by your other scripts without having to be maintained in the consolidated database.

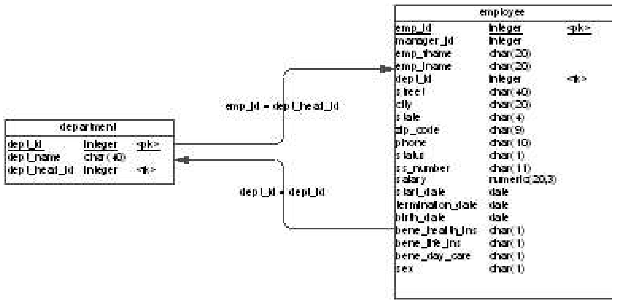
Foreign key cycles

This section describes a specific limitation in UltraLite synchronization that results from a series of tables linked together by foreign keys so that a cycle is formed.

MobiLink synchronization from an UltraLite remote database requires that all changes be committed to the consolidated database in one transaction. To facilitate this single transaction for multiple tables, the inserts, updates, and deletes for each table must be ordered so that operations for a primary table come before the associated foreign table. This ensures that the insert in the foreign table will have its foreign key referential integrity constraint satisfied (likewise for other operations like delete).

The UltraLite analyzer automatically orders all the tables in the remote database so those primary tables are uploaded before foreign tables based on the schema in the reference database. The ordering is always possible as long as there are no foreign key cycles in the schema.

The figure illustrates a simple foreign key cycle between two tables.



If a foreign key cycle is detected by the UltraLite analyzer, the cycle must be broken for the analyzer to successfully complete without any errors. The foreign key cycle must be broken on both the reference database and the consolidated database in order for synchronization transactions to be successfully applied.

For an Adaptive Server Anywhere consolidated and reference database, one of the foreign keys can be made to **check on commit** so that foreign key referential integrity is checked during the commit phase rather than when the operation is initiated. Other database vendors may have similar methods but if not, the schema must be redesigned to eliminate the foreign key cycle.

Example

```
create table c (
    id integer not null primary key,
    c_pk integer not null
);
create table p (
    pk integer not null primary key,
    c_id integer not null,
    foreign key p_to_c (c_id) references c(id)
);
alter table c
add foreign key c_to_p (c_pk)
references p(pk)
check on commit;
```

Synchronization parameters reference

Synchronization parameters control the synchronization between an UltraLite database and the MobiLink synchronization server. The way you set the parameters depends on the specific UltraLite interface you are using. This section describes the effects of the parameters, and provides links to other locations for information on how to set them.

Authentication Parameters synchronization parameter

Function	Supplies parameters to the authentication_parameters script.
Usage	Use this parameter to supply any values required by your authentication_parameters script. These may be a user name and password, for example. If you use this parameter, you must also supply the number of parameters. See "Number of Authentication Parameters parameter" on page 166.
Allowed values	An array of strings. Null is not allowed as a value for any of the strings, but you can supply an empty string.
See also	"Number of Authentication Parameters parameter" on page 166 "authenticate_parameters connection event" [*MobiLink Synchronization Reference*, page 98]

Interfaces

- ♦ **MobileVB** "SyncParms" [*UltraLite for MobileVB User's Guide*, page 135]
- ♦ **ActiveX** "SyncParms" [*UltraLite ActiveX User's Guide*, page 153]
- ♦ **UltraLite.NET** See ianywhere.UltraLite.SyncParms.
- ♦ **Embedded SQL** See "auth_parms parameter" [*UltraLite Embedded SQL User's Guide*, page 139].
- ♦ **Static C++ API** See "auth_parms parameter" [*UltraLite Static C++ User's Guide*, page 125].
- ♦ **Static Java API** See "auth_parms parameter" [*UltraLite Static Java User's Guide*, page 71].

Authentication Status synchronization parameter

Function	Reports the status of MobiLink user authentication. The MobiLink synchronization server provides this information to the client.
Usage	If you are implementing a custom authentication scheme, the authenticate_user or authenticate_user_hashed synchronization script must return one of the allowed values of this parameter.

The parameter is set by the MobiLink synchronization server, and so is read-only.

Allowed values

The allowed values are held in an interface-specific enumeration.

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].

See also

"Authenticating MobiLink Users" [*MobiLink Synchronization User's Guide*, page 103].

Interfaces

- ♦ **MobileVB** "SyncResult" [*UltraLite for MobileVB User's Guide*, page 138]
- ♦ **ActiveX** "SyncResult" [*UltraLite ActiveX User's Guide*, page 155]
- ♦ **UltraLite.NET** See ianywhere.UltraLite.SyncResult.
- ♦ **Embedded SQL** See "auth_status parameter" [*UltraLite Embedded SQL User's Guide*, page 139].
- ♦ **Static C++ API** See "auth_status parameter" [*UltraLite Static C++ User's Guide*, page 125].
- ♦ **Static Java API** See "auth_status parameter" [*UltraLite Static Java User's Guide*, page 71].

Authentication Value synchronization parameter

Function

Reports results of a custom MobiLink user authentication script. The MobiLink synchronization server provides this information to the client.

Default

The values set by the default MobiLink user authentication mechanism are described in "Authentication Status synchronization parameter" on page 162.

Usage

The parameter is set by the MobiLink synchronization server, and so is read-only.

See also

"authenticate_user connection event" [*MobiLink Synchronization Reference*, page 100]

"authenticate_user_hashed connection event" [*MobiLink Synchronization Reference*, page 104]

"Authentication Status synchronization parameter" on page 162

Interfaces

- ♦ **MobileVB** "SyncResult" [*UltraLite for MobileVB User's Guide*, page 138]
- ♦ **ActiveX** "SyncResult" [*UltraLite ActiveX User's Guide*, page 155]
- ♦ **UltraLite.NET** See ianywhere.UltraLite.SyncResult.

- **Embedded SQL** See “auth_value synchronization parameter” [*UltraLite Embedded SQL User’s Guide,* page 140].
- **Static C++ API** See “auth_value synchronization parameter” [*UltraLite Static C++ User’s Guide,* page 126].
- **Static Java API** See “auth_value synchronization parameter” [*UltraLite Static Java User’s Guide,* page 72].

Checkpoint Store synchronization parameter

Function	Adds additional checkpoints of the database during synchronization to limit database growth during the synchronization process. The checkpoint operation adds I/O operations for the application and for the Palm conduit and so slows synchronization. The option is most useful for large downloads with many updates. Devices with slow flash memory may not want to pay the performance penalty.
Default	By default, only required checkpointing is done.

Interfaces

- **MobileVB** “SyncParms” [*UltraLite for MobileVB User’s Guide*, page 135]
- **ActiveX** “SyncParms” [*UltraLite ActiveX User’s Guide*, page 153]
- **UltraLite.NET** See ianywhere.UltraLite.SyncParms.
- **Embedded SQL** See “checkpoint_store synchronization parameter” [*UltraLite Embedded SQL User’s Guide,* page 141].
- **Static C++ API** See “checkpoint_store synchronization parameter” [*UltraLite Static C++ User’s Guide,* page 127].
- **Static Java API** Not available.

Disable Concurrency synchronization parameter

Function	Disallow database access from other threads during synchronization.
Default	The parameter is a Boolean value, and by default is false (allowing concurrent database access). Data access is read-write during the download phase, and read-only otherwise.
See also	“Threading in UltraLite applications” on page 47

Interfaces

- **MobileVB** Not available
- **ActiveX** Not available
- **UltraLite.NET** See ianywhere.UltraLite.SyncParms.

- **Embedded SQL** See “disable_concurrency synchronization parameter” [*UltraLite Embedded SQL User’s Guide*, page 141].
- **Static C++ API** See “disable_concurrency synchronization parameter” [*UltraLite Static C++ User’s Guide*, page 127].
- **Static Java API** See “disable_concurrency synchronization parameter” [*UltraLite Static Java User’s Guide*, page 73].

Download Only synchronization parameter

Function	Do not upload any changes from the UltraLite database during this synchronization.
Default	The parameter is a Boolean value, and by default is false.
See also	“Including read-only tables in an UltraLite database” on page 158.
	“Upload Only synchronization parameter” on page 176

Interfaces

- **MobileVB** “SyncParms” [*UltraLite for MobileVB User’s Guide*, page 135]
- **ActiveX** “SyncParms” [*UltraLite ActiveX User’s Guide*, page 153]
- **UltraLite.NET** See ianywhere.UltraLite.SyncParms.
- **Embedded SQL** See “download_only synchronization parameter” [*UltraLite Embedded SQL User’s Guide*, page 141].
- **Static C++ API** See “download_only synchronization parameter” [*UltraLite Static C++ User’s Guide*, page 127].
- **Static Java API** See “download_only synchronization parameter” [*UltraLite Static Java User’s Guide*, page 73].

Ignored Rows synchronization parameter

Function

This boolean parameter is set to **true** if any rows were ignored by the MobiLink synchronization server during synchronization because of absent scripts.

The parameter is read-only.

Interfaces

- **MobileVB** “SyncResult” [*UltraLite for MobileVB User’s Guide*, page 138]
- **ActiveX** “SyncResult” [*UltraLite ActiveX User’s Guide*, page 155]
- **UltraLite.NET** See ianywhere.UltraLite.SyncResult.
- **Embedded SQL** See “ignored_rows synchronization parameter” [*UltraLite Embedded SQL User’s Guide*, page 141].

- **Static C++ API** See “ignored_rows synchronization parameter” [*UltraLite Static C++ User’s Guide*, page 127].
- **Static Java API** See “ignored_rows synchronization parameter” [*UltraLite Static Java User’s Guide*, page 74].

New Password synchronization parameter

Function	Sets a new MobiLink password associated with the user name.
Default	The parameter is optional, and is a string.
See also	“Authenticating MobiLink Users” [*MobiLink Synchronization User’s Guide*, page 103].

Interfaces

- **MobileVB** “SyncParms” [*UltraLite for MobileVB User’s Guide*, page 135]
- **ActiveX** “SyncParms” [*UltraLite ActiveX User’s Guide*, page 153]
- **UltraLite.NET** See ianywhere.UltraLite.SyncParms.
- **Embedded SQL** See “new_password synchronization parameter” [*UltraLite Embedded SQL User’s Guide*, page 142].
- **Static C++ API** See “new_password synchronization parameter” [*UltraLite Static C++ User’s Guide*, page 128].
- **Static Java API** See “new_password synchronization parameter” [*UltraLite Static Java User’s Guide*, page 74].

Number of Authentication Parameters parameter

Function	Supply the number of authentication parameters being passed to the authentication_parameters script.
Default	No parameters supplied.
See also	“Authentication Parameters synchronization parameter” on page 162 “authenticate_parameters connection event” [*MobiLink Synchronization Reference*, page 98]

Interfaces

- **MobileVB** “SyncParms” [*UltraLite for MobileVB User’s Guide*, page 135]
- **ActiveX** “SyncParms” [*UltraLite ActiveX User’s Guide*, page 153]
- **UltraLite.NET** See ianywhere.UltraLite.SyncParms.
- **Embedded SQL** See “num_auth_parms parameter” [*UltraLite Embedded SQL User’s Guide*, page 142].

- **Static C++ API** See “num_auth_parms parameter” [*UltraLite Static C++ User’s Guide*, page 128].
- **Static Java API** See “num_auth_parms parameter” [*UltraLite Static Java User’s Guide*, page 74].

Observer synchronization parameter

Function	A pointer to a callback function that monitors synchronization.
See also	“User Data synchronization parameter” on page 177

Interfaces

- **MobileVB** Declare the connection object With Events. See “Connection” [*UltraLite for MobileVB User’s Guide*, page 89]
- **ActiveX** Use CreateObjectWithEvents when opening the DatabaseManager object. See “DatabaseManager” [*UltraLite ActiveX User’s Guide*, page 120]
- **UltraLite.NET** See ianywhere.UltraLite.SyncResult.
- **Embedded SQL** See “observer synchronization parameter” [*UltraLite Embedded SQL User’s Guide*, page 142].
- **Static C++ API** See “observer synchronization parameter” [*UltraLite Static C++ User’s Guide*, page 128].
- **Static Java API** See “observer synchronization parameter” [*UltraLite Static Java User’s Guide*, page 75].

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	The parameter is optional, and is a string.
See also	“Authenticating MobiLink Users” [*MobiLink Synchronization User’s Guide*, page 103].

Interfaces

- **MobileVB** “SyncParms” [*UltraLite for MobileVB User’s Guide*, page 135]
- **ActiveX** “SyncParms” [*UltraLite ActiveX User’s Guide*, page 153]
- **UltraLite.NET** See ianywhere.UltraLite.SyncParms.
- **Embedded SQL** See “password synchronization parameter” [*UltraLite Embedded SQL User’s Guide*, page 142].

- **Static C++ API** See “password synchronization parameter” [*UltraLite Static C++ User’s Guide,* page 128].
- **Static Java API** See “password synchronization parameter” [*UltraLite Static Java User’s Guide,* page 75].

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.

See also

“-pi option” [*MobiLink Synchronization Reference,* page 76]

Interfaces

- **MobileVB** “SyncParms” [*UltraLite for MobileVB User’s Guide,* page 135]
- **ActiveX** “SyncParms” [*UltraLite ActiveX User’s Guide,* page 153]
- **UltraLite.NET** See ianywhere.UltraLite.SyncParms.
- **Embedded SQL** See “ping synchronization parameter” [*UltraLite Embedded SQL User’s Guide,* page 143].
- **Static C++ API** See “ping synchronization parameter” [*UltraLite Static C++ User’s Guide,* page 129].

- **Static Java API** See "ping synchronization parameter" [*UltraLite Static Java User's Guide*, 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

When synchronizing, set the publication parameter to a **publication mask**: an OR'd list of publication constants.

See also

"The UltraLite generator" on page 96

"Designing sets of data to synchronize separately" on page 156

Interfaces

- **MobileVB** "PublicationSchema" [*UltraLite for MobileVB User's Guide*, page 116]
- **ActiveX** "PublicationSchema" [*UltraLite ActiveX User's Guide*, page 135]
- **UltraLite.NET** See ianywhere.UltraLite.PublicationSchema.
- **Embedded SQL** See "publication synchronization parameter" [*UltraLite Embedded SQL User's Guide*, page 144].
- **Static C++ API** See "publication synchronization parameter" [*UltraLite Static C++ User's Guide*, page 130].
- **Static Java API** See "publication synchronization parameter" [*UltraLite Static Java User's Guide*, page 77].

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].

This parameter is not used in the static Java API.

To use secure synchronization from UltraLite Java applications, choose a separate stream. For more information, see "UlSecureRSASocketStream synchronization parameters" on page 188 and "UlSecureSocketStream synchronization parameters" on page 189.

Default

The 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]

Interfaces

- ♦ **Embedded SQL** See "security synchronization parameter" [*UltraLite Embedded SQL User's Guide,* page 144].
- ♦ **Static C++ API** See "security synchronization parameter" [*UltraLite Static C++ User's Guide,* page 130].
- ♦ **Static Java API** Use a separate synchronization stream. See "UlSecureRSASocketStream synchronization parameters" on page 188, and "UlSecureSocketStream synchronization parameters" on page 189.

Security Parameters 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 169.

This parameter is not required in static Java applications. To use secure synchronization from UltraLite static Java applications, choose a separate stream. For more information, see "UlSecureRSASocketStream synchronization parameters" on page 188 and "UlSecureSocketStream synchronization parameters" on page 189.

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 96.

Interfaces

- **Embedded SQL** See "security_parms synchronization parameter" [*UltraLite Embedded SQL User's Guide*, page 145].

- **Static C++ API** See "security_parms synchronization parameter" [*UltraLite Static C++ User's Guide*, page 131].

- **Static Java API** Use a separate synchronization sream.

Send Column Names synchronization parameter

Function

When set to 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.

This parameter is not available for UltraLite static Java applications.

See also

"-za option" [*MobiLink Synchronization Reference*, page 28]

Interfaces

- **MobileVB** "SyncParms" [*UltraLite for MobileVB User's Guide*, page 135]

- **ActiveX** "SyncParms" [*UltraLite ActiveX User's Guide*, page 153]

- **UltraLite.NET** See ianywhere.UltraLite.SyncParms.

- **Embedded SQL** See “send_column_names synchronization parameter” [*UltraLite Embedded SQL User’s Guide*, page 146].
- **Static C++ API** See “send_column_names synchronization parameter” [*UltraLite Static C++ User’s Guide*, page 132].
- **Static Java API** See “send_column_names synchronization parameter” [*UltraLite Static Java User’s Guide*, page 79].

Send Download Acknowledgement 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 a 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.

Interfaces

- **MobileVB** “SyncParms” [*UltraLite for MobileVB User’s Guide*, page 135]
- **ActiveX** “SyncParms” [*UltraLite ActiveX User’s Guide*, page 153]
- **UltraLite.NET** See ianywhere.UltraLite.SyncParms.
- **Embedded SQL** See “send_download_ack synchronization parameter” [*UltraLite Embedded SQL User’s Guide*, page 146].
- **Static C++ API** See “security_parms synchronization parameter” [*UltraLite Static C++ User’s Guide*, page 131].
- **Static Java API** This parameter is not available for static Java applications.

Stream Error synchronization parameter

Function

Set a structure to hold communications error reporting information.

This feature is not available for UltraLite static Java applications.

Default

The parameter is set by the MobiLink synchronization server, and so is read-only. It is set only if a communication error occurs during synchronization.

Interfaces

- **MobileVB** “SyncResult” [*UltraLite for MobileVB User’s Guide*, page 138]
- **ActiveX** “SyncResult” [*UltraLite ActiveX User’s Guide*, page 155]
- **UltraLite.NET** See ianywhere.UltraLite.SyncResult.

- ♦ **Embedded SQL** See “stream_error synchronization parameter” [*UltraLite Embedded SQL User’s Guide*, page 147].
- ♦ **Static C++ API** See “stream_error synchronization parameter” [*UltraLite Static C++ User’s Guide*, page 133].
- ♦ **Static Java API** This feature is not available in static Java.

Stream Type synchronization parameter

Function

Set the MobiLink synchronization stream to use for synchronization.

Most synchronization streams require parameters to identify the MobiLink synchronization server address and other behavior. These parameters are supplied in the **stream_parms** parameter.

☞ For more information, see “Stream Parameters synchronization parameter” on page 175.

Default

The parameter has no default value, and must be explicitly set.

When the type of stream requires a parameter, pass that parameter using the Stream Parameters parameter; otherwise, set the Stream Parameters parameter to null.

The following stream functions are available, but not all are available on all target platforms:

Stream	**Description**
ActiveSync	ActiveSync synchronization (Windows CE only). ☞ For a list of stream parameters, see “ActiveSync synchronization stream parameters” on page 179.
HTTP	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” on page 184.

Stream	Description
HTTPS	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” on page 186.
TCP/IP	Synchronize via TCP/IP. ☞ For a list of stream parameters, see “TCP/IP stream parameters” on page 182.
UlSecureSocketStream()	TCP/IP or HTTP synchronization with transport-layer security using elliptic curve encryption. This stream is available for static Java applications only. ☞ For a list of stream parameters, see “UlSecureSocketStream synchronization parameters” on page 189.
UlSecureRSASocket-Stream()	TCP/IP or HTTP synchronization with transport-layer security using RSA encryption. This stream is available for static Java applications only. ☞ For a list of stream parameters, see “UlSecureSocketStream synchronization parameters” on page 189.

Interfaces

- **MobileVB** “SyncParms” [*UltraLite for MobileVB User’s Guide*, page 135]
- **ActiveX** “SyncParms” [*UltraLite ActiveX User’s Guide*, page 153]
- **UltraLite.NET** See ianywhere.UltraLite.SyncParms.
- **Embedded SQL** See “stream synchronization parameter” [*UltraLite Embedded SQL User’s Guide*, page 146].

- **Static C++ API** See "stream synchronization parameter" [*UltraLite Static C++ User's Guide*, page 132].
- **Static Java API** See "stream synchronization parameter" [*UltraLite Static Java User's Guide*, page 80].

Stream Parameters 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 synchronization stream parameters" on page 179
- "HotSync synchronization stream parameters" on page 181
- "HTTP stream parameters" on page 184
- "HTTPS stream parameters" on page 186
- "TCP/IP stream parameters" on page 182
- "UlSecureRSASocketStream synchronization parameters" on page 188
- "UlSecureSocketStream synchronization parameters" on page 189

Default

The parameter is optional, is a string, and by default is null.

See also

"Stream parameters reference" on page 179.

Interfaces

- **MobileVB** "SyncParms" [*UltraLite for MobileVB User's Guide*, page 135]
- **ActiveX** "SyncParms" [*UltraLite ActiveX User's Guide*, page 153]
- **UltraLite.NET** See ianywhere.UltraLite.SyncParms.
- **Embedded SQL** See "stream_parms synchronization parameter" [*UltraLite Embedded SQL User's Guide*, page 149].
- **Static C++ API** See "stream_parms synchronization parameter" [*UltraLite Static C++ User's Guide*, page 135].
- **Static Java API** See "stream_parms synchronization parameter" [*UltraLite Static Java User's Guide*, page 83].

Upload OK synchronization parameter

Function	Reports the status of data uploaded to the MobiLink synchronization server.
Usage	The MobiLink synchronization server sets this parameter, and so it is read-only. After synchronization, the parameter holds **true** if the upload was successful, and **false** otherwise. You can check this parameter if there was a synchronization error, to know whether data was successfully uploaded before the error occurred.
Interfaces	♦ **MobileVB** "SyncResult" [*UltraLite for MobileVB User's Guide*, page 138] ♦ **ActiveX** "SyncResult" [*UltraLite ActiveX User's Guide*, page 155] ♦ **UltraLite.NET** See ianywhere.UltraLite.SyncResult. ♦ **Embedded SQL** See "upload_ok synchronization parameter" [*UltraLite Embedded SQL User's Guide*, page 149]. ♦ **Static C++ API** See "upload_ok synchronization parameter" [*UltraLite Static C++ User's Guide*, page 135]. ♦ **Static Java API** See "upload_ok synchronization parameter" [*UltraLite Static Java User's Guide*, page 83].

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.
See also	"Synchronizing high-priority changes" on page 158 "Download Only synchronization parameter" on page 165
Interfaces	♦ **MobileVB** "SyncParms" [*UltraLite for MobileVB User's Guide*, page 135] ♦ **ActiveX** "SyncParms" [*UltraLite ActiveX User's Guide*, page 153] ♦ **UltraLite.NET** See ianywhere.UltraLite.SyncParms. ♦ **Embedded SQL** See "upload_only synchronization parameter" [*UltraLite Embedded SQL User's Guide*, page 150].

- **Static C++ API** See “upload_only synchronization parameter” [*UltraLite Static C++ User’s Guide*, page 136].
- **Static Java API** See “upload_only synchronization parameter” [*UltraLite Static Java User’s Guide*, page 84].

User Data synchronization parameter

Function	Make application-specific information available to the synchronization observer.
Usage	When implementing the synchronization observer callback function , you can make application-specific information available by providing information using the User Data parameter.
See also	“Observer synchronization parameter” on page 167

Interfaces

- **MobileVB** “Connection” [*UltraLite for MobileVB User’s Guide*, page 89]
- **ActiveX** “DatabaseManager” [*UltraLite ActiveX User’s Guide*, page 120]
- **UltraLite.NET** See ianywhere.UltraLite.SyncResult.
- **Embedded SQL** See “user_data synchronization parameter” [*UltraLite Embedded SQL User’s Guide*, page 150].
- **Static C++ API** See “user_data synchronization parameter” [*UltraLite Static C++ User’s Guide*, page 136].
- **Static Java API** See “user_data synchronization parameter” [*UltraLite Static Java User’s Guide*, page 84].

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	The user name is required unless the MobiLink synchronization server is being run with user authentication turned off. For more information, see “-zu option” [*MobiLink Synchronization Reference*, page 31].
See also	“Authenticating MobiLink Users” [*MobiLink Synchronization User’s Guide*, page 103].
	“MobiLink users” [*MobiLink Synchronization User’s Guide*, page 20].

Interfaces

- **MobileVB** “SyncParms” [*UltraLite for MobileVB User’s Guide*, page 135]

- **ActiveX** “SyncParms” [*UltraLite ActiveX User’s Guide*, page 153]
- **UltraLite.NET** See ianywhere.UltraLite.SyncParms.
- **Embedded SQL** See “user_name synchronization parameter” [*UltraLite Embedded SQL User’s Guide*, page 150].
- **Static C++ API** See “user_name synchronization parameter” [*UltraLite Static C++ User’s Guide*, page 136].
- **Static Java API** See “user_name synchronization parameter” [*UltraLite Static Java User’s Guide*, page 85].

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.

See also

“Script versions” [*MobiLink Synchronization User’s Guide*, page 49].

Interfaces

- **MobileVB** “SyncParms” [*UltraLite for MobileVB User’s Guide*, page 135]
- **ActiveX** “SyncParms” [*UltraLite ActiveX User’s Guide*, page 153]
- **UltraLite.NET** See ianywhere.UltraLite.SyncParms.
- **Embedded SQL** See “version synchronization parameter” [*UltraLite Embedded SQL User’s Guide*, page 151].
- **Static C++ API** See “version synchronization parameter” [*UltraLite Static C++ User’s Guide*, page 137].
- **Static Java API** See “version synchronization parameter” [*UltraLite Static Java User’s Guide*, page 85].

Stream parameters reference

This section lists the stream parameters for each synchronization stream. The stream parameters provide information such as addressing information (host and port) and protocol-specific information to ensure that the client can locate and properly communicate with the MobiLink synchronization server.

ActiveSync synchronization stream parameters

The ActiveSync synchronization stream is accessible only from Native UltraLite for Java, embedded SQL, and static C++ API applications running on Windows CE.

To choose ActiveSync synchronization:

- In Native UltraLite for Java, supply StreamType.ACTIVE_SYNC as the argument to the syncParms.setStream method. For example:

  ```
  _conn.syncParms.setStream( StreamType.ACTIVE_SYNC );
  ```

  ☞ For more information, see **ianywhere.native_ultralite.StreamType** and **ianywhere.native_ultralite.SyncParms** in the Native UltraLite for Java API Reference.

- In embedded SQL and the static C++ API, supply ULActiveSyncStream() as the stream synchronization parameter. For example:

  ```
  ul_synch_info info;
  ...
  info.stream = ULActiveSyncStream();
  ```

  ☞ For more information, see "ULActiveSyncStream function" [*UltraLite Embedded SQL User's Guide*, page 105].

Meaning of synchronization stream parameters

The stream parameters control the connection from the MobiLink ActiveSync provider, running on the desktop machine, to the MobiLink synchronization server.

The stream parameters take the following form:

```
stream=stream_name;provider_stream_parameters
```

where *stream_name* indicates the protocol for the conduit to use when communicating from the conduit to the MobiLink synchronization server. It must be one of the following:

- **tcpip**
- **http**

♦ **https**

and where *provider_stream_parameters* is a set of stream parameters for use by the ActiveSync provider, and has the same form as the stream parameters for the protocol in use. For the given stream, the *provider_stream_parameters* adopts the same defaults as the stream parameters for the protocol. The default value for the *stream_name* is **tcpip**.

For example, the following static C++ code uses an HTTP synchronization stream:

```
ULInitSynchInfo( &info );
info.stream = ULActiveSyncStream();
info.stream_parms = "stream=http";
ULSynchronize( &sqlca, &info );
```

☞ For more information on *provider_stream_parameters*, see "TCP/IP stream parameters" on page 182, "HTTP stream parameters" on page 184, and "HTTPS stream parameters" on page 186.

Adding encryption to ActiveSync synchronization

To add Certicom encryption to the stream, the root certificates must be in a file on the desktop machine. This is different from other UltraLite applications, where the encryption information is embedded in the **security** synchronization parameter.

The stream parameters need to be specified in the stream parameters in much the same way as for Adaptive Server Anywhere MobiLink clients . The format is:

security=*cipher*{ *keyword***=***value***;**. . . }

where *cipher* must be certicom_tls and the keywords are taken from the following list:

♦ **certificate_company** The organization field on the certificate.

♦ **certificate_unit** The organization unit field on the certificate.

♦ **certificate_name** The common name field on the certificate.

♦ **trusted_certificates** The location of the trusted certificates.

For example, a static C++ application may use a line such as the following:

```
info.stream_parms = "stream=tcpip;security=ecc_tls(trusted_
         certificates=trusted.crt)";
```

☞ For more information, see "CREATE SYNCHRONIZATION USER statement [MobiLink]" [*ASA SQL Reference*, page 351].

HotSync synchronization stream parameters

The HotSync synchronization stream is accessible only from UltraLite for MobileVB applications, embedded SQL applications, and static C++ API applications running on the Palm Computing Platform. Unlike HTTP or TCP/IP synchronization, HotSync synchronization is initiated externally by the HotSync Manager, rather than by a synchronization function within the UltraLite applcation.

To choose HotSync synchronization:

- In UltraLite for MobileVB, choose ulPalmConduit from the ULStreamType enumeration as the ULSyncParms.Stream.

  ☞ For more information, see “SyncParms” [*UltraLite for MobileVB User’s Guide,* page 135].

- In embedded SQL or the static C++ API, supply the ul_synch_info structure to the ULPalmExit or ULData::PalmExit method of your application. The stream parameter is ignored and may be set to UL_NULL.

  ☞ For more information, see “ULPalmExit function” [*UltraLite Embedded SQL User’s Guide,* page 124].

Meaning of synchronization stream parameters

For HotSync synchronization, the stream parameters do *not* control the connection from the device to the HotSync Manager or HotSync Server. Instead, they specify the connection from the MobiLink conduit, running at the HotSync manager or server, to the MobiLink synchronization server.

The argument has the following form:

```
stream=stream_name;conduit_stream_parameters
```

where *stream_name* indicates the protocol for the conduit to use when communicating from the conduit to the MobiLink synchronization server. It must be one of the following:

- **tcpip**
- **http**
- **https**

and where *conduit_stream_parameters* is a set of stream parameters for use by the conduit, and has the same form as the **stream_parms** argument for the protocol in use. For the given stream, the *conduit_stream_parameters* adopts the same defaults as the **stream_parms** argument for the protocol. The default value for the *stream_name* is tcpip.

For example, the following embedded SQL code uses an HTTP synchronization stream:

```
ULInitSynchInfo( &info );
info.stream_parms = "stream=http";
```

☞ For more information on *conduit_stream_parameters*, see "TCP/IP stream parameters" on page 182, "HTTP stream parameters" on page 184, and "HTTPS stream parameters" on page 186.

Null value and default settings

If you use HotSync synchronization, and do not supply stream parameters, the conduit searches in the registry for the stream name and stream parameters. If it finds no valid stream, the default stream and stream parameters is used. This default stream parameter setting is:

```
stream=tcpip;host=localhost
```

Adding encryption to HotSync synchronization

To add Certicom encryption to the stream, the root certificates must be in a file on the desktop machine. This is different from other UltraLite applications, where the encryption information is embedded in the **security** synchronization parameter.

The stream parameters need to be specified in the stream parameters in much the same way as for Adaptive Server Anywhere MobiLink clients . The format is:

security=*cipher*{ *keyword***=***value***;**. . . }

where *cipher* must be certicom_tls and the keywords are taken from the following list:

- **certificate_company** The organization field on the certificate.
- **certificate_unit** The organization unit field on the certificate.
- **certificate_name** The common name field on the certificate.
- **trusted_certificates** The location of the trusted certificates.

For example, in a static C++ application:

```
info.stream_parms = "stream=tcpip;security=ecc_tls(trusted_
          certificates=trusted.crt)";
```

☞ For more information, see "CREATE SYNCHRONIZATION USER statement [MobiLink]" [*ASA SQL Reference*, page 351].

TCP/IP stream parameters

The TCP/IP synchronization stream is accessible from all UltraLite interfaces.

Selecting the TCP/IP synchronization stream

To select TCP/IP as the synchronization stream:

♦ In UltraLite for MobileVB and UltraLite ActiveX, choose ulTCPIP from the ULStreamType enumeration as the ULSyncParms.Stream.

  ☞ For more information, see "SyncParms" [*UltraLite for MobileVB User's Guide,* page 135] and "SyncParms" [*UltraLite ActiveX User's Guide,* page 153].

♦ In Native UltraLite for Java, supply StreamType.TCPIP as the argument for SyncParms.setStream().

  ☞ For more information, see **ianywhere.native_ultralite.StreamType** and **ianywhere.native_ultralite.SyncParms** in the Native UltraLite for Java API Reference.

♦ In embedded SQL or the static C++ API, supply ULSocketStream() as the stream synchronization parameter.

  ☞ For more information, see "ULSocketStream function" [*UltraLite Embedded SQL User's Guide,* page 132].

♦ In the static Java API,

  For more information, see "stream synchronization parameter" [*UltraLite Static Java User's Guide,* page 80].

Stream parameters

Synchronization stream parameters for the TCP/IP stream are chosen from the following table:

Parameter	Description
client_port=*nnnnn* **client_port**=*nnnnn-mmmmm*	A range of client ports for communication. If only one value is specified, the end of the range is 100 greater than the initial value, for a total of 101 ports. The option can be useful for clients inside a firewall communicating with a MobiLink synchronization server outside the firewall.

Parameter	Description
host=*hostname*	The host name or IP number for the machine on which the MobiLink synchronization server is running. The default value is **localhost**, except on Windows CE. For Windows CE, the default setting corresponds to the desktop machine where the CE device's cradle is connected, which is stored as the *ipaddr* entry in the registry folder *Comm\Tcpip\Hosts\ppp_peer*. Do not use **localhost**, which refers to the device itself, on Windows CE. For the Palm Computing Platform, the default value of **localhost** refers to the device itself. You should supply an explicit host name or IP address to connect to a desktop machine.
liveness_timeout=*n*	The amount of time, in seconds, after a client stops communicating before MobiLink recovers the connection. A value of 0 means that there is no timeout. This option is only effective if download acknowledgement if set to off. The default is 120 seconds.
port=*portnumber*	The socket port number on the host machine. The port number must be a decimal number that matches the port the MobiLink synchronization server is setup to monitor. The default value for the port parameter is 2439, which is the IANA registered port number for the MobiLink synchronization server.

HTTP stream parameters

The HTTP synchronization stream is accessible from all UltraLite components.

Selecting the HTTP synchronization stream

To select HTTP as the synchronization stream:

♦ In UltraLite for MobileVB and UltraLite for ActiveX, choose ulHTTP from the ULStreamType enumeration as the ULSyncParms.Stream.

  ☞ For more information, see "SyncParms" [*UltraLite for MobileVB User's Guide*, page 135].

♦ In Native UltraLite for Java, supply StreamType.HTTP as the argument for SyncParms.setStream().

☞ For more information, see **ianywhere.native_ultralite.StreamType** and **ianywhere.native_ultralite.SyncParms** in the Native UltraLite for Java API Reference.

- In embedded SQL or the static C++ API, supply ULHTTPStream() as the stream synchronization parameter.

  ☞ For more information, see "ULHTTPStream function" [*UltraLite Embedded SQL User's Guide,* page 121].

Stream parameters

Synchronization stream parameters for the HTTP stream are chosen from the following table:

Parameter	Description
buffer_size=*nnnn*	The amount of memory allocated for sending content. The default is 1 K.
client_port=*nnnnn* **client_port**=*nnnnn-mmmmm*	A range of client ports for communication. If only one value is specified, the end of the range is 100 greater than the initial value, for a total of 101 ports. The option can be useful for clients inside a firewall communicating with a MobiLink synchronization server outside the firewall.
version= *versionnumber*	A string specifying the version of HTTP to use. You have a choice of **1.0** or **1.1**. The default value is **1.1**.
host=*hostname*	The host name or IP number for the machine on which the MobiLink synchronization server is running. The default value is **localhost**. For Windows CE, the default value is the value of *ipaddr* in the registry folder *Comm\Tcpip\Hosts\ppp_peer.* This allows a CE device to connect to a MobiLink synchronization server executing on the desktop machine where the CE device's cradle is connected. For the Palm Computing Platform, the default value of **localhost** refers to the device. It is recommended that an explicit host name or IP address be specified.

Parameter	Description
persistent={ **0** | **1** }	If this is set to 1 the client uses the same TCP connection for all HTTP requests in a synchronization. The default value is 1 on the Palm OS, and 0 elsewhere.
port=*portnumber*	The socket port number. The port number must be a decimal number that matches the port the MobiLink synchronization server is setup to monitor. The default value for the port parameter is 2439, which is the IANA registered port number for the MobiLink synchronization server.
proxy_host= *proxy_-hostname*	The host name of the proxy server.
proxy_port= *proxy_-portnumber*	The port number of the proxy server. The default value is 80.
url_suffix=*suffix*	The suffix to add to the URL on the first line of each HTTP request. When synchronizing through a proxy server, the suffix may be necessary in order to find the MobiLink synchronization server. The default value is **MobiLink**.

HTTPS stream parameters

The HTTPS synchronization stream is accessible from all UltraLite components.

Selecting the HTTPS synchronization stream

To select HTTPS as the synchronization stream:

♦ In UltraLite for MobileVB and UltraLite for eMbedded Visual Basic, choose ulHTTPS from the ULStreamType enumeration as the ULSyncParms.Stream.

☞ For more information, see "SyncParms" [*UltraLite for MobileVB User's Guide*, page 135].

♦ In Native UltraLite for Java, supply StreamType.HTTPS as the argument for SyncParms.setStream().

☞ For more information, see **ianywhere.native_ultralite.StreamType** and **ianywhere.native_ultralite.SyncParms** in the Native UltraLite for Java API Reference.

♦ In embedded SQL or the static C++ API, supply ULHTTPSStream() as the stream synchronization parameter.

☞ For more information, see "ULHTTPSStream function" [*UltraLite Embedded SQL User's Guide,* page 120].

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" in the book *Introducing SQL Anywhere Studio.*

Stream parameters

Synchronization stream parameters for the HTTPS stream are chosen from the following table:

Parameter	Description
buffer_size=*nnnn*	The amount of memory allocated for sending content. The default is 1 K.
client_port=*nnnnn* **client_port**=*nnnnn-mmmmm*	A range of client ports for communication. If only one value is specified, the end of the range is 100 greater than the initial value, for a total of 101 ports. The option can be useful for clients inside a firewall communicating with a MobiLink synchronization server outside the firewall.
host=*hostname*	The host name or IP number for the machine on which the MobiLink synchronization server is running. The default value is **localhost**. For Windows CE, the default value is the value of *ipaddr* in the registry folder *Comm\Tcpip\Hosts\ppp_peer.* This allows a CE device to connect to a MobiLink synchronization server executing on the desktop machine where the CE device's cradle is connected. For the Palm Computing Platform, the default value of **localhost** refers to the device. It is recommended that an explicit host name or IP address be specified.
persistent={ **0** | **1** }	If this is set to 1 the client uses the same TCP connection for all HTTP requests in a synchronization. The default value is 1 on the Palm OS, and 0 elsewhere.

Parameter	Description
port=*portnumber*	The socket port number. The port number must be a decimal number that matches the port the MobiLink synchronization server is setup to monitor. The default value for the port parameter is 2439, which is the IANA registered port number for the MobiLink synchronization server.
proxy_host= *proxy_-hostname*	The host name of the proxy server.
proxy_port= *proxy_-portnumber*	The port number of the proxy server. The default value is 80.
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_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.
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.
url_suffix=*suffix*	The suffix to add to the URL on the first line of each HTTP request. When synchronizing through a proxy server, the suffix may be necessary in order to find the MobiLink synchronization server. The default value is **MobiLink**.
version= *versionnumber*	A string specifying the version of HTTP to use. You have a choice of **1.0** or **1.1**. The default value is **1.1**.

UlSecureRSASocketStream synchronization parameters

Transport-layer security using Certicom RSA encryption is accessed from static Java applications as a separate stream, accessed using the

UlSecureRSASocketStream object. This is different behavior from other UltraLite applications, where a separate parameter is supplied to the synchronization structure.

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].

Stream parameters

The synchronization parameters for UlSecureRSASocketStream are identical to those for UlSecureSocketStream. For a complete listing, see “UlSecureSocketStream synchronization parameters” on page 189.

☞ For more information, see “stream synchronization parameter” [*UltraLite Static Java User’s Guide,* page 80], and “Using transport-layer security” [*UltraLite Static Java User’s Guide,* page 48].

UlSecureSocketStream synchronization parameters

Transport-layer security using Certicom elliptic-curve encryption is accessed from static Java applications as a separate stream, accessed using the UlSecureSocketStream object.

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].

☞ For more information, see “stream synchronization parameter” [*UltraLite Static Java User’s Guide,* page 80], and “Using transport-layer security” [*UltraLite Static Java User’s Guide,* page 48].

Stream parameters

The parameters for the UlSecureSocketStream are supplied in an semicolon-separated string. These parameters are chosen from the following table:

Parameter	Description
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.

Parameter	Description
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.
client_port=*nnnnn* **client_port**=*nnnnn-mmmmm*	A range of client ports for communication. If only one value is specified, the end of the range is 100 greater than the initial value, for a total of 101 ports. The option can be useful for clients inside a firewall communicating with a MobiLink synchronization server outside the firewall.
host=*hostname*	The host name or IP number for the machine on which the MobiLink synchronization server is running. The default value is localhost, except on Windows CE. For Windows CE, the default setting corresponds to the desktop machine where the CE device's cradle is connected, which is stored as the ipaddr entry in the registry folder Comm\Tcpip\Hosts\ppp_peer. Do not use localhost, which refers to the device itself, on Windows CE. For the Palm Computing Platform, the default value of localhost refers to the device itself. You should supply an explicit host name or IP address to connect to a desktop machine.

Parameter	Description
port=*portnumber*	The socket port number on the host machine. The port number must be a decimal number that matches the port the MobiLink synchronization server is setup to monitor. The default value for the port parameter is 2439, which is the IANA registered port number for the MobiLink synchronization server.

Part IV

Static Programming Interfaces

This part describes the embedded SQL, C++ API, and Java static programming interfaces.

When using a static interface, all queries must be specified at compile time.

CHAPTER 9

Using UltraLite Static Interfaces

About this chapter

This chapter presents an overview of the UltraLite static programming interfaces.

When using static interfaces, the SQL statements to be used in an application must be specified at compile time. In a dynamic model, SQL statements can be specified at run time. The static interfaces are embedded SQL, the static C++ API, and the static Java API which uses JDBC. This chapter describes aspects common to all static UltraLite interfaces.

Contents

Overview

This section describes the development environment and process for UltraLite static interfaces.

The development environment for static UltraLite applications

Developing UltraLite applications using a static interface requires the following tools.

♦ **A reference database** A reference database is an Adaptive Server Anywhere database that serves as a model of the UltraLite database you want to create. You create this database yourself, using tools such as Sybase Central.

  Your UltraLite database is a subset of the columns, tables, and indexes, in your reference database. The arrangement of tables and of the foreign key relationships between them is called the database **schema**.

  In addition to modeling the UltraLite database, you need to add the SQL statements that are to be included in your UltraLite application to the reference database.

  ☞ For more information, see "Preparing a reference database" on page 200.

♦ **A supported development tool** You use a standard development tool to develop UltraLite applications. For the non-UltraLite specific portions of your application, such as the user interface, use your development tool in the usual way. For the UltraLite-specific data-access portions, you also need to use the UltraLite development tools.

  It can be convenient to separate the data access code from the user interface and internal logic of your application.

  ☞ For information about supported application development tools, see "UltraLite host platforms" [*Introducing SQL Anywhere Studio*, page 126].

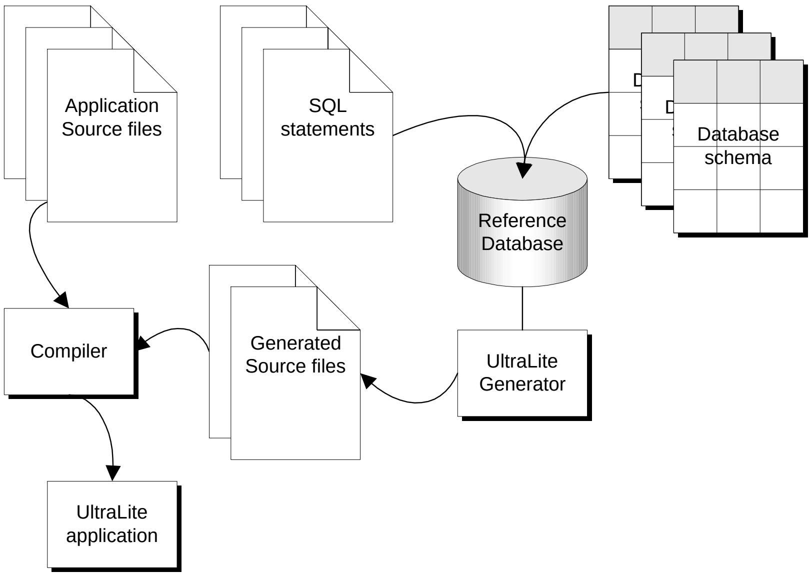
♦ **UltraLite development tools** UltraLite includes several tools for development using the static interfaces.

  - **The UltraLite generator** This application generates source code that implements the underlying query execution, data storage, and synchronization features of your application. The generator is required for all kinds of UltraLite development using static SQL.
  - **The SQL preprocessor** This application is needed only if you are developing an UltraLite application using embedded SQL. It reads your embedded SQL source files and generates standard C/C++ files. As it scans the embedded SQL source files, it also stores information in the reference database that is used by the generator.

- **UltraLite runtime libraries** UltraLite includes a runtime library for each target platform. On some platforms, this is a static library that becomes part of your application executable; on other platforms it is a dynamic link library. For Java, the runtime library is a jar file. UltraLite includes all the header files and import files needed to use the runtime libraries.

The static UltraLite development process

The basic features of the development process are common to all static interfaces. The following diagram summarizes the key features.



- Create a reference database, which contains a superset of the tables to be included in your application. It may also contain representative data for your application. This reference database is needed only as part of the development process, and is not required by your final application.

- Add the SQL statements into a special table in the reference database. The way this is accomplished is dependent on the interface you choose:
  - If you are using the C++ API or Java, these statements are added to your database using Sybase Central or a stored procedure.
  - If you are using embedded SQL, the SQL preprocessor adds the statements to the reference database for you.

- Run the UltraLite generator, which produces source files that include code needed to execute your SQL statements, and code needed to define the database schema for your UltraLite application. This generated code includes function calls into the UltraLite runtime library.

- Create application source files. If you are using embedded SQL, the SQL preprocessor reads your *.sqc* files and inserts the SQL statements into the reference database for you.
- Compile your application source files together with the generated source files to produce your UltraLite application.

Adding synchronization

Most UltraLite applications include synchronization to integrate their data with data on a consolidated database.

☞ For more information about synchronization, and the kinds of synchronization available, see "Synchronization for UltraLite Applications" on page 143.

Choosing an UltraLite static interface

There are three static interfaces for developing UltraLite applications:

- **C++ API** Development using C or C++ with data access features using a result-set based API.
- **Embedded SQL** Development using C or C++ with data access features using embedded SQL statements.
- **Static Java API** Development using the Java programming language.

The decision whether to use Java or C/C++ development will be determined primarily by your target platform.

Here are some considerations when choosing between embedded SQL and the C++ API:

- Embedded SQL is an industry standard programming method, while the C++ API is a proprietary API.
- Embedded SQL gives more control in designing your application. If you are experienced with embedded SQL development, you can design a more efficient application using this method.
- Many programmers are more familiar with API-based programming. The C++ API requires less learning for these developers.
- The C++ API generates classes and associated methods for manipulating the database. It enforces standardized function names and so can be a quicker approach in terms of development time.

Preparing a reference database

To implement the UltraLite database engine for your application, the UltraLite generator must have access to an Adaptive Server Anywhere reference database. This database must contain the following information:

- **Database schema** The database objects used in your UltraLite application, including tables and any indexes on those tables you wish to use in your application.

  ☞ For more information, see "Using an existing database as a reference database" on page 202.

- **Data** (Optional) You can fill your reference database with data that is similar in quantity and distribution to the data you expect your UltraLite database to hold. The UltraLite analyzer automatically uses this information to optimize the performance of your application.

  ☞ For more information, see "Using an existing database as a reference database" on page 202.

- **Queries** The UltraLite system tables must contain any SQL statements you wish to use in your application.

  ☞ For more information, see "Defining SQL statements for your application" on page 204.

- **Publications** If you wish to add multiple synchronization options to your application, you can do so using publications. You also add publications to your database if you wish to develop a C++ API application without defining queries.

  ☞ For information on multiple synchronization options, see "Designing sets of data to synchronize separately" on page 156.

- **Database options** Database options such as date formats and govern some aspects of database behavior that can make applications behave differently. The UltraLite database is generated with the same option settings as those in the reference database.

  For many purposes, you can leave all database options at their default settings.

  ☞ For more information, see "Setting database options in the reference database" on page 201.

Creating a reference database

The analyzer uses the reference database as a template when constructing your UltraLite application.

❖ To create a reference database

1. Start with an existing Adaptive Server Anywhere database or create a new database using the *dbinit* command.

   ☞ For more information on upgrading a database, see "Using an existing database as a reference database" on page 202.

2. Add the tables and foreign key relationships that you need within your application. You can use any convenient tool, such as Sybase Central or Sybase PowerDesigner Physical Architect (included with SQL Anywhere Studio), or a more powerful database design tool such as the complete Sybase PowerDesigner package.

> **Performance tip**
> You do not need to include any data in your reference database. However, if you populate your database tables with data representative of the data you expect to be stored by a typical user of your application, the UltraLite analyzer automatically uses this data to optimize the performance of your application.

☞ For information about designing a database and creating a schema, see "Designing Your Database" [*ASA SQL User's Guide*, page 3].

Example

1. Create a database.

   From a command prompt, execute the following statement:

   ```
   dbinit path\dbname.db
   ```

2. Use Sybase Central to add tables for your UltraLite application, based on your own needs.

3. Add your sample data. Interactive SQL includes an Import menu item that allows several common file formats to be imported.

   ☞ For more information, see "Importing data" [*ASA SQL User's Guide*, page 529].

Setting database options in the reference database

UltraLite does not support the getting or setting of option values.

When the UltraLite application is generated, certain option values in the reference database affect the behavior of the generated code. The following options have an effect:

- Date_format
- Date_order

- Nearest_century
- Precision
- Scale
- Time_format
- Timestamp_format

By setting these options in the reference database, you can control the behavior of your UltraLite database. The option setting in your reference database is used when generating your UltraLite application.

Using an existing database as a reference database

Many UltraLite applications synchronize data via MobiLink with a central, master store of data called the **consolidated database**. Do not confuse a reference database with a consolidated database. The reference database for the UltraLite application is generally a different database from the consolidated database.

Only an Adaptive Server Anywhere consolidated database can also be used as a reference database. If your consolidated database is of another type, you must create an Adaptive Server Anywhere reference database. Even if your consolidated database is Adaptive Server Anywhere, you must create a separate reference database if you wish to have a different schema or use different settings in your UltraLite application.

You can choose any of the supported ODBC-compliant database management products to create and manage the consolidated database, including Adaptive Server Enterprise, Adaptive Server Anywhere, Oracle, Microsoft SQL Server, and IBM DB2.

If you have an existing Adaptive Server Anywhere database that you will be using as a consolidated database, you could make a copy of it for your reference database.

❖ To create a reference database from a non-Adaptive Server Anywhere database

1. Create a new Adaptive Server Anywhere database.

   You can use the *dbinit* command or use Sybase Central.

2. Add the tables and foreign-key relationships that you need within your application using your consolidated database as a guide.

   You can use a tool such as Sybase Physical Data Architect to re-engineer the consolidated database.

3. Populate your database tables with representative data from your consolidated database.

   You need not transfer all the information in your consolidated database, only a representative sample. In the early stages of development, you do not need sample data at all. For production applications, you may want to use representative data because access plans of UltraLite queries are based on the distribution of data in the reference database.

☞ For more information on creating reference databases from non-Adaptive Server Anywhere databases, see "Migrating databases to Adaptive Server Anywhere" [*ASA SQL User's Guide*, page 548].

Optimizing query execution

You can improve the performance of your static UltraLite applications using the following techniques.

♦ **add an index** If you frequently retrieve information in a particular order, consider adding an index to your reference database. Primary keys are automatically indexed, but other columns are not. Particularly on slow devices, an index can improve performance dramatically.

♦ **add representative data** The Adaptive Server Anywhere optimizer automatically optimizes the performance of your queries. It chooses access plans using the information present in your reference database. To improve application performance, fill your reference database with data that is representative in size and distribution of the data you expect your application will hold once it is deployed.

Defining SQL statements for your application

All the data access instructions for your application are defined by adding SQL statements to the reference database.

If you use the C++ API, you can also use publications to define data access methods. For information on using publications, see “Defining UltraLite tables” [*UltraLite Static C++ User’s Guide*, page 21].

If you are using embedded SQL, the SQL preprocessor carries out the tasks in this section for you.

Creating an UltraLite project

When you add SQL statements to a reference database, you assign them to an UltraLite **project**. By grouping them this way, you can develop multiple applications using the same reference database.

When the UltraLite generator runs against a reference database to generate the database source code files, it takes a project name as an argument and generates the code for the SQL statements in that project.

You can define an UltraLite project using Sybase Central or by directly calling a system stored procedure.

If you are using embedded SQL, the SQL preprocessor defines the UltraLite project for you and you do not need to create it explicitly.

❖ To create an UltraLite project (Sybase Central)

1. In Sybase Central, connect to your database if you are not already connected.

2. In the left pane, open the database container.

3. In the left pane, open the UltraLite Projects folder.

4. From the File menu, choose New ➤ UltraLite Project.

   The UltraLite Project Creation wizard appears.

5. Enter an UltraLite project name and click Finish to create the project in the database.

   ☞ For information on UltraLite project naming rules, see “ul_add_project system procedure” on page 212.

❖ To create an UltraLite project (SQL)

1. From Interactive SQL or another application, enter the following command:

   ```
   call ul_add_project( 'project-name' )
   ```

   where *project-name* is the name of the project.

   ☞ For more information, see "ul_add_project system procedure" on page 212.

❖ To create an UltraLite project (embedded SQL)

1. If you are using the embedded SQL interface, specify the UltraLite project name on the SQL Preprocessor command line, and the preprocessor adds the project to the database for you.

   ☞ For more information, see "Building Embedded SQL Applications" [*UltraLite Embedded SQL User's Guide,* page 17].

Notes

UltraLite project names must conform to the rules for database identifiers. If you include spaces in the project name, do not enclose the name in double quotes, as these are added for you by Sybase Central or the stored procedure.

☞ For more information, see "Identifiers" [*ASA SQL Reference,* page 7].

Adding SQL statements to an UltraLite project

Each UltraLite application carries out a set of data access requests. These requests are implemented differently in each interface, but the data access requests are defined in the same way for each model.

You define the data access requests that an UltraLite application can carry out by adding a set of SQL statements to the UltraLite project for that application in your reference database. The UltraLite generator then creates the code for a database engine that can execute the set of SQL statements.

In the C++ API, you can also use publications to define data access methods. For information on using publications, see "Defining UltraLite tables" [*UltraLite Static C++ User's Guide,* page 21].

You can add SQL statements to an UltraLite project using Sybase Central, or by directly calling a system stored procedure. If you are using embedded SQL, the SQL preprocessor adds the SQL statements in your embedded SQL source files to the reference database for you.

❖ To add a SQL statement to an UltraLite project (Sybase Central)

1. In Sybase Central, connect to your database if you are not already connected.
2. In the left pane, open the database container.
3. In the left pane, open the UltraLite Projects folder.
4. Open the project for your application.
5. From the File menu, choose New ➤ UltraLite statement.

   The UltraLite Statement Creation wizard appears.
6. Enter a short, descriptive name for the statement, and click Next
7. Enter the statement itself, and click Finish to add the statement to the project.

   You can test the SQL statements against the database by right-clicking the statement and choosing Execute From Interactive SQL from the popup menu.

   ☞ For information on what kinds of statement you can use, see "Writing UltraLite SQL statements" on page 207.

❖ To add a SQL statement to an UltraLite project (SQL)

1. From Interactive SQL or another application, enter the following command:

   ```
   call ul_add_statement( 'project-name',
      'statement-name',
      'sql-statement' )
   ```

   where *project-name* is the name of the project, *statement-name* is a short descriptive name, and *sql-statement* is the actual SQL statement.

   ☞ For more information, see "ul_add_statement system procedure" on page 212.

❖ To add a SQL statement to an UltraLite project (embedded SQL)

1. If you are using the embedded SQL interface, specify the UltraLite project name on the SQL Preprocessor command line.

   No statement name is used in embedded SQL development.

   ☞ For more information, see "Building Embedded SQL Applications" [*UltraLite Embedded SQL User's Guide*, page 17].

Notes

Statement names should be short and descriptive. They are used by the UltraLite generator to identify the statement for use in Java or in the C++ API. For example, a statement named **ProductQuery** generates a C++ API class named **ProductQuery** and a Java constant named **PRODUCT_QUERY**. Names should be valid SQL identifiers.

The SQL statement syntax is checked when you add the statement to the database, and syntax errors give an error message to help you identify mistakes.

You can use Sybase Central or ul_add_statement to update a statement in a project, in just the same way as you add a statement. If a statement already exists, it is overwritten with the new syntax. You must regenerate the UltraLite code whenever you modify a statement.

Writing UltraLite SQL statements

This section describes what SQL statements you can add to an UltraLite project, and describes how to use placeholders in your SQL statements.

☞ For information on the range of SQL that you can use, see "Overview of SQL support in UltraLite" on page 108.

How to supply double quotes

The SQL statement that you enter, whether into Sybase Central or as an argument to **ul_add_statement**, is added to the reference database as a string. It must therefore conform to the rules for SQL strings.

You must escape some characters in your SQL statements using the backslash character.

☞ For information on SQL strings, see "Strings" [*ASA SQL Reference*, page 8].

Using variables with statements

For most insert or update statements, you do not know the new values ahead of time. You can use question marks as placeholders for variables, and supply values at run time:

```
call ul_add_statement(
   ’ProductApp’,
   ’AddCap’,
   ’INSERT INTO \"DBA\".product ( id, name, price )
     VALUES( ?, ?, ? )’
)
```

Placeholders can also be used in the WHERE clause of queries:

```
call ul_add_statement(
   ’ProductApp’,
   ’ProductQuery’,
   ’SELECT id, name, price
    FROM \"DBA\".product
    WHERE price > ?’
)
```

The backslash characters are used to escape the double quotes.

In embedded SQL, you use **host variables** as placeholders. For more information, see “Using host variables” [*UltraLite Embedded SQL User’s Guide*, page 30].

For SQL statements containing placeholders, an extra parameter on the **Open** or **Execute** method of the generated C++ class is defined for each parameter. For Java applications, you use the JDBC set methods to assign values for the parameters.

Generating the UltraLite data access code

To generate the code for storing and accessing the UltraLite database, the **UltraLite generator** analyzes your reference database and the SQL statements you use in your application. The UltraLite generator is a command-line application. It takes a set of command-line options to customize the behavior for each project. For example, it can generate either C/C++ or Java code, depending on the command-line options you supply.

The data storage code includes only those tables and columns of the reference database that you use in your application. Additionally, the UltraLite generator includes indexes present in your reference database whenever they improve the efficiency of your application.

The data access code includes only those SQL statements that you have added to the project in the reference database.

The result is a custom database engine tailored to your application. The engine is much smaller than a general-purpose database engine because the UltraLite generator includes only the features your application uses.

☞ For more information about the UltraLite generator, see "The UltraLite generator" on page 96.

Configuring development tools for static UltraLite development

Most 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.

☞ For specific instructions on adding UltraLite projects to a dependency-based development environment, see “Configuring development tools for embedded SQL development” [*UltraLite Embedded SQL User’s Guide*, page 24].

CHAPTER 10

Static Development Model Reference

About this chapter

This chapter provides reference information about for static development models.

Contents

Reference database stored procedures

This section describes system stored procedures in the Adaptive Server Anywhere reference database, which can be used to add SQL statements to a project.

For each SQL statement added in this way, the UltraLite generator defines a C++ or Java class.

These system procedures are owned by the built-in user ID **dbo**.

ul_add_statement system procedure

Function

Adds a SQL statement to an UltraLite project.

Syntax

ul_add_statement (in @**project** char(128),
in @**name** char(128),
in @**statement** text)

Permissions

DBA authority required

Side effects

None

See also

"ul_add_project system procedure" on page 212

"ul_delete_statement system procedure" on page 213

Description

Adds or modifies a statement to an UltraLite project.

project The UltraLite project to which the statement should be added. The UltraLite generator defines classes for all statements in a project at one time.

name The name of the statement. This name is used in the generated classes.

statement A string containing the SQL statement.

If a statement of the same name in the same project exists, it is updated with the new syntax. If *project* does not exist, it is created.

Examples

The following call adds a statement to the TestSQL project:

```
call ul_add_statement(
'TestSQL', 'TestQuery',
'select prod_id, price, prod_name from ulproduct where price <
         ?' )
```

ul_add_project system procedure

Function

Creates an UltraLite project.

Syntax	**ul_add_project** (in @**project** char(128))
Permissions	DBA authority required
Side effects	None
See also	"ul_delete_statement system procedure" on page 213
Description	Adds an UltraLite project to the database. The project acts as a container for the SQL statements in an application, and the project name is supplied on the UltraLite generator command line so that it can define classes for all statements in the project.
	project The UltraLite project name.
Examples	The following call adds a project named **Product** to the database:

```
call ul_add_project( 'Product' )
```

ul_delete_project system procedure

Function	Removes an UltraLite project from a database.
Syntax	**ul_delete_project** (in @**project** char(128))
Permissions	DBA authority required
Side effects	None
See also	"ul_add_project system procedure" on page 212
	"ul_delete_statement system procedure" on page 213
Description	Removes an UltraLite project from the database.
	project The UltraLite project to be deleted from the database.
Examples	The following call deletes the **Product** project:

```
call ul_delete_project( 'Product' )
```

ul_delete_statement system procedure

Function	Removes a SQL statement from an UltraLite project.
Syntax	**ul_delete_statement** (in @**project** char(128), in @**name** char(128))
Permissions	DBA authority required
Side effects	None

See also

"ul_add_project system procedure" on page 212

"ul_add_statement system procedure" on page 212

Description

Removes a statement from an UltraLite project.

project The UltraLite project from which the statement should be removed.

name The name of the statement. This name is used in the generated classes.

Examples

The following call removes a statement from the **Product** project:

```
call ul_delete_statement( 'Product', 'AddProd' )
```

ul_set_codesegment system procedure

Function

For Palm Computing Platform development using the C++ API, assigns a SQL statement from an UltraLite project to a particular segment.

Syntax

ul_set_codesegment(in **@project** char(128),
in **@name** char(128), in **@segment_name** *char(8)*)

Side effects

None

See also

"ul_add_statement system procedure" on page 212

"Explicitly assigning segments" [*UltraLite Static C++ User's Guide*, page 52]

Description

Explicitly assigns the generated code for a C++ API SQL statement to a named Palm segment.

project The UltraLite project to which the statement applies.

name The name of the statement as defined in "ul_add_statement system procedure" on page 212..

segment_name The name of the segment to which the statement is assigned.

Examples

The following call assigns the statement **mystmt** in project **myproject** to segment **MYSEG1**.

```
call ul_set_codesegment(
     'myproject', 'mystmt', 'MYSEG1' )
```

Macros and compiler directives for UltraLite C/C++ applications

This section describes compiler directives to supply for UltraLite C/C++ applications. Unless stated otherwise, directives apply to both embedded SQL and C++ API applications.

Compiler directives can be supplied on your compiler command line or in the compiler settings dialog box of your user interface. Alternatively, they can be defined in source code.

On the compiler command line, a compiler directive is commonly set by using the /D command-line option. For example, to compile an UltraLite application with user authentication, a makefile for the Microsoft Visual C++ compiler may look as follows:

```
CompileOptions=/c /DPRWIN32 /Od /Zi /DWIN32
/D__NT__ /DUL_USE_DLL /DULB_USE_BIGINT_TYPES
/DULB_USE_FLOAT_TYPES /DUL_ENABLE_USER_AUTH

IncludeFolders= \
/I"$(VCDIR)\include" \
/I"$(ASANY9)\h"

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

where *VCDIR* is your Visual C++ directory and *ASANY9* is your SQL Anywhere directory.

In source code, directives are supplied using the `#define` statement.

UL_AS_SYNCHRONIZE macro

Function	Provides the name of the callback message used to indicate an ActiveSync synchronization.
Applies to	Windows CE applications using ActiveSync only.
See also	"Adding ActiveSync synchronization to your application" [*UltraLite Embedded SQL User's Guide*, page 96]
	"Adding ActiveSync synchronization to your application" [*UltraLite Static C++ User's Guide*, page 68]

UL_ENABLE_OBFUSCATION macro

Function	By default, obfuscation is disabled. To enable obfuscation, define UL_ENABLE_OBFUSCATION when compiling the generated database.

Applies to	The generated database code.
See also	"Encrypting UltraLite databases" on page 36

UL_ENABLE_USER_AUTH macro

Function	For C++ API applications only, define this directive to enable user authentication. Without this directive, there is no user authentication on C++ API UltraLite applications.
Applies to	The *ulapi.cpp* file.
See also	"Adding user authentication to your application" [*UltraLite Static C++ User's Guide*, page 28]

UL_ENABLE_SEGMENTS macro

Function	Instructs the compiler to generate multi-segment code for Palm Computing Platform applications.
Applies to	The generated database code.
See also	"Enabling multi-segment code generation" [*UltraLite Embedded SQL User's Guide*, page 78]
	"Enabling multi-segment code generation" [*UltraLite Static C++ User's Guide*, page 51]

UL_STORE_PARMS macro

Function	Supply a set of keyword-value pairs to configure database storage.
Syntax	`#define UL_STORE_PARMS UL_TEXT( "`*keyword*`=`*value*`;. . . " )`
	All spaces in the keyword-value list are significant, except spaces at the start of the string and any spaces that immediately follow a semicolon.
Usage	Define the UL_STORE_PARMS macro in the header of your application source code so that it is visible to all **db_init()** calls.
Parameters	Keywords are case insensitive. The case sensitivity of the values depends on the application interpreting it. For example, the case sensitivity of the filename depends on the operating system.
	☞ For a list of available parameters, see "Database schema parameters" on page 61, and "Additional connection parameters" on page 65.
Examples	The following statements set the cache size to 128 kb.

```
#undef  UL_STORE_PARMS
#define UL_STORE_PARMS  UL_TEXT("cache_size=128k")
   . . .
db_init( &sqlca );
```

You can set UL_STORE_PARMS to a string, then set the value of that string programmatically before calling **db_init**, as in the following example. The UL_TEXT macro and the **_stprintf** function are used to achieve proper character encoding.

```
char store_parms[32];
#undef  UL_STORE_PARMS
#define UL_STORE_PARMS  store_parms
...
   /* Set cache_size to the correct number of bytes. */
...
_stprintf( store_parms, UL_TEXT("cache_size=%lu"),
   cache_size );
db_init( &sqlca );
```

See also

"Database schema parameters" on page 61

"Additional connection parameters" on page 65

"Configuring and managing database storage" [*UltraLite Embedded SQL User's Guide*, page 56]

"Encrypting UltraLite databases" on page 36

UL_SYNC_ALL macro

Function

Provides a publication mask that refers to all tables in the database, including those not in publications.

See also

"Publication synchronization parameter" on page 169

"ULGetLastDownloadTime function" [*UltraLite Embedded SQL User's Guide*, page 115]

"ULCountUploadRows function" [*UltraLite Embedded SQL User's Guide*, page 108]

"UL_SYNC_ALL_PUBS macro" on page 217

UL_SYNC_ALL_PUBS macro

Function

Provides a publication mask that refers to all tables in the database that are in publications.

See also

"Publication synchronization parameter" on page 169

“ULGetLastDownloadTime function” [*UltraLite Embedded SQL User’s Guide*, page 115]

“ULCountUploadRows function” [*UltraLite Embedded SQL User’s Guide*, page 108]

“UL_SYNC_ALL macro” on page 217

UL_TEXT macro

Function

Prepares constant strings to be compiled as single-byte strings or wide-character strings. In embedded SQL and C++ API applications, use this macro to enclose all constant strings so that the compiler handles these parameters correctly.

UL_USE_DLL macro

Function

For Windows CE and Windows applications only, define this directive to use the runtime library DLL, rather than a static runtime library.

Applies to

The generated database code.

UNDER_NT macro

Function

Use this macro when compiling UltraLite code for Windows NT/2000/XP only.

By default, this macro is defined in all new Visual C++ projects that target Windows NT/2000/XP.

UNDER_CE macro

Function

Use this macro when compiling UltraLite applications for Windows CE only.

By default, this macro is defined in all new eMbedded Visual C++ projects.

See also

“Developing UltraLite Applications for Windows CE” [*UltraLite Embedded SQL User’s Guide*, page 87].

UNDER_PALM_OS macro

Function

Use this macro when compiling UltraLite applications for Palm OS only.

This macro is defined in the *ulpalmXX.h* header file included in UltraLite Palm OS applications by the UltraLite plugin. For more information, see “Using the UltraLite plug-in for CodeWarrior” [*UltraLite Embedded SQL User’s Guide*, page 75].

See also

“Developing UltraLite Applications for the Palm Computing Platform”

[*UltraLite Embedded SQL User's Guide*, page 71].

Index