UltraLite™ User's Guide

Last modified: October 2002
Part Number: 38134-01-0802-01

Last modified October 2002. Part number 38134-01-0802-01.

Contents

About This Manual

Subject	This manual describes the UltraLite deployment technology for Adaptive Server Anywhere. With UltraLite, you can develop and deploy database applications to handheld, mobile, or embedded devices, such as devices running the Palm Computing Platform, Windows CE, VxWorks, or Java.
Audience	This manual is intended for all application developers writing programs that use UltraLite deployment. Familiarity with relational databases and Adaptive Server Anywhere is assumed.

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 set

The SQL Anywhere Studio documentation set 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.
- **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.

- **Adaptive Server Anywhere Error Messages** This book provides a complete listing of Adaptive Server Anywhere error messages together with diagnostic information.
- **Adaptive Server Anywhere C2 Security Supplement** 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. The book does *not* include the security features added to the product since certification.
- **MobiLink Synchronization User's Guide** This book describes all aspects of 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.
- **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.
- **UltraLite User's Guide** This book describes how to build database applications for small devices such as handheld organizers using the UltraLite deployment technology for Adaptive Server Anywhere databases.
- **UltraLite User's Guide for PenRight! MobileBuilder** This book is for users of the PenRight! MobileBuilder development tool. It describes how to use UltraLite technology in the MobileBuilder programming environment.
- **SQL Anywhere Studio Help** This book is provided online only. It includes the context-sensitive help for Sybase Central, Interactive SQL, and other graphical tools.

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

Documentation formats

SQL Anywhere Studio provides documentation in the following formats:

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

  To access the online books on Windows operating systems, choose Start➤Programs➤Sybase SQL Anywhere 8➤Online Books. You can navigate the online books using the HTML Help table of contents, index, and search facility in the left pane, and using the links and menus in the right pane.

  To access the online books on UNIX operating systems, run the following command at a command prompt:

  ```
  dbbooks
  ```

- **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 following books are included in the SQL Anywhere Studio box:

  - *Introducing SQL Anywhere Studio*.
  - *Adaptive Server Anywhere Getting Started*.
  - *SQL Anywhere Studio Quick Reference*. This book is available only in printed form.

  The complete set of books is available as the SQL Anywhere Documentation set from Sybase sales or from e-Shop, the Sybase online store, at http://e-shop.sybase.com/cgi-bin/eshop.storefront/.

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 are shown 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. 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.

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

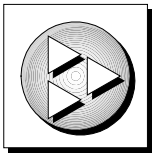
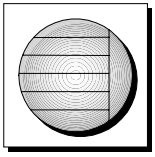
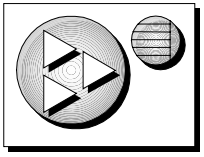
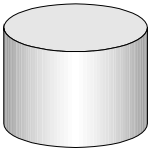
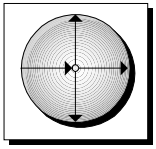
If the QUOTES option is chosen, one of ON or OFF must be provided. The brackets and braces should not be typed.

- **One or more options** If you choose more than one, separate your choices with commas.

  { **CONNECT**, **DBA**, **RESOURCE** }

Graphic icons

The following icons are used in this documentation:

Icon	Meaning
	A client application.
	A database server, such as Sybase Adaptive Server Anywhere or Adaptive Server Enterprise.
	An UltraLite application and database server. In UltraLite, the database server and the application are part of the same process.
	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, SQL Remote Message Agent, and the Replication Agent (Log Transfer Manager) for use with Replication Server.
	A Sybase Replication Server.
	A programming interface.

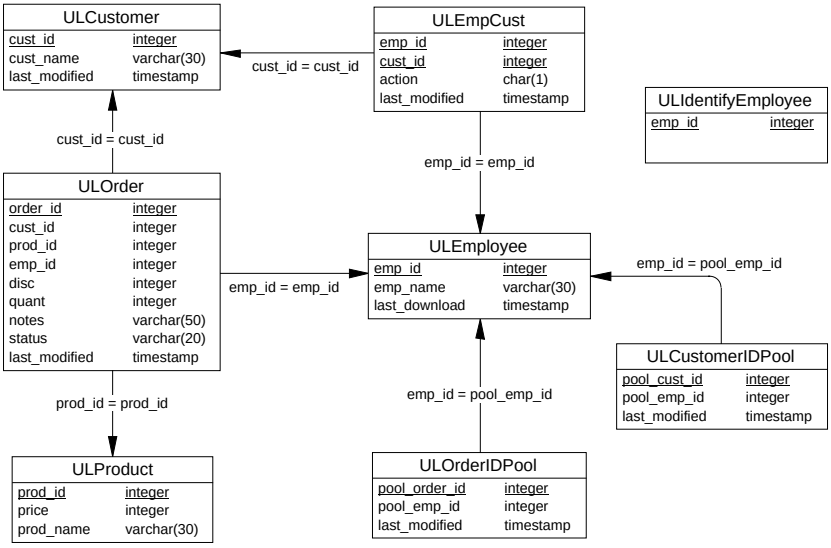
The UltraLite sample database

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

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 ONE

Introduction to UltraLite

This part introduces UltraLite, presents a sample UltraLite application, and provides general information on how to build your own UltraLite application.

CHAPTER 1

Introduction to UltraLite

About this chapter

This chapter introduces the UltraLite database deployment technology, describing the purpose and defining characteristics of UltraLite.

Contents

UltraLite features

UltraLite is a deployment technology for Adaptive Server Anywhere databases, aimed at small, mobile, and embedded devices. Supported target platforms include Windows CE, Palm OS, Java, and VxWorks devices.

UltraLite provides the following benefits for users of small devices:

- The functionality and reliability of a transaction-processing SQL database.
- The ability to synchronize data with your central database-management system.
- An extremely small memory footprint.
- C/C++ or Java development.

Full-featured SQL

UltraLite allows applications on small devices to use full-featured SQL to accomplish data storage, retrieval, and manipulation. UltraLite supports referential integrity, transaction processing, and multi-table joins of all varieties. In fact, UltraLite supports most of the same data types, runtime functions, and SQL data manipulation features as Sybase Adaptive Server Anywhere.

Synchronization to industry-standard RDBMS

UltraLite uses MobiLink synchronization technology 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.

Small footprint custom database

UltraLite provides an ultra-small footprint by generating a custom database engine for your application. This custom engine includes only the features required by your application.

Each UltraLite application has its own database, modeled after a reference Adaptive Server Anywhere database.

C/C++ UltraLite custom database engines for your application can be as small as 50 kb, depending on your deployment platform and the number of SQL statements in the application and the SQL features used.

In-memory database UltraLite target devices may have no hard disk and tend to have relatively slow processors. The UltraLite runtime employs algorithms and data structures that provide high performance and low memory use.

Supported development models

You can develop UltraLite applications in the following ways:

- **C/C++ using the UltraLite C++ API** The C++ interface exposes tables and SQL statements as objects, with methods to move through the rows of the table or a result set and to execute statements.
- **C/C++ using embedded SQL** Embedded SQL is an easy way of including SQL statements directly in C or C++ source code.
- **Java using JDBC** UltraLite Java applications support standard JDBC data access methods.

Supported platforms

Platform support for UltraLite is of the following kinds:

- **Target platforms** The **target platform** is the device and operating system on which you deploy your finished UltraLite application.
- **Host platforms** For each target platform, you develop your applications using a particular development tool and operating system. The tool and operating system comprise the **host platform**. Target platform manufacturers may supply **emulators** to ease development tasks. Emulators simulate the target platform, while running on the host platform.

  You may be able to use other tools to develop your UltraLite applications, but supporting documentation and technical assistance are provided only for the supported development platforms.

The supported target platforms for UltraLite applications fall into two categories: those supporting C/C++ UltraLite applications and those supporting Java UltraLite applications.

Supported platforms for C/C++ applications

This version of UltraLite supports the following platforms for C/C++ applications using Embedded SQL or the UltraLite C++ API:

- **Palm Computing Platform** UltraLite applications can be built for Palm Computing Platform devices running the Palm OS versions 3.*x* or version 4.*x*. For example, the Palm III, Palm V, and Palm VII organizers are suitable target platforms, as are the Handspring Visor, TR6Pro, m505, and so on. PalmPilot Personal and PalmPilot Professional devices running Palm OS version 2.x or earlier are not supported target platforms.

  UltraLite applications can be run for testing and demonstration purposes on the Palm emulator.

  The supported development platform for the Palm Computing Platform is the Metrowerks CodeWarrior version 6, 7, or 8 on Windows NT/2000/XP, and the GCC PRC Tools version 2.0.
- **CE** Windows CE 3.0 and later are supported.

  Windows CE 3.0 support includes support for Pocket PC, including Pocket PC 2002, as well as Handheld PC 2000.

The Windows CE 3.0 operating system is supported on any of the following processors:

- MIPS processor
- ARM processor
- x86 processor. UltraLite supports the x86 processor for emulation purposes only.

UltraLite applications can be run for testing and demonstration purposes in the Pocket PC and Pocket PC 2002 emulators.

The supported development platforms for Windows CE are:

- Microsoft eMbedded Visual C++ 3.0 and later.

- **WindRiver VxWorks** UltraLite applications can be built for devices running the VxWorks operating system in any of the following configurations:
  - A 386, 486 or Pentium PC running VxWorks 5.3 or 5.4 with the Intel x86 Board Support Package (BSP) version 1.1 or above.
  - A PowerPC running VxWorks 5.4. The UltraLite library has been compiled for the PowerPC860 chip and tested on an MBX860 board using TCP/IP synchronization.

  VxWorks 5.5 is not supported.

  The VxWorks version of UltraLite also runs under the VxSim emulator. The full simulator is required to carry out synchronization. VxSim-Lite can be used for testing, but does not support synchronization.

  Development for VxWorks is supported on WindRiver Tornado development environment for Windows, version 1.0.1 or above.

  UltraLite requires a dosFs (MS-DOS-compatible file system) device or a functionally equivalent device to store the database in a persistent manner in a file.

- **Windows** Windows operating systems other than Windows CE are supported for testing and demonstration purposes only.

For more information on supported platforms, see "UltraLite supported operating systems" on page 144 of the book *Introducing SQL Anywhere Studio*.

Supported platforms for Java applications

Deployment using the Sun Java 2 (JDK 1.2.*x* or later) environment is supported. Also, deployment using the Sun JDK 1.1.4 or later is supported. To use UltraLite in applets, you must use the Java Plug-in 1.2.*x* to run the applets. There is no guarantee that applets will work with the built-in Java VMs for the Internet Explorer or Netscape browsers.

Development using Sybase PowerJ or the Sun JDK is supported.

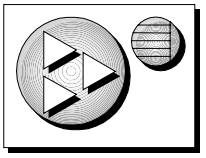
UltraLite architecture

SQL database products typically use a client/server architecture. The database is normally stored in one or more files in the file system, or directly on a storage device. The database server receives and processes SQL requests from client applications, maintaining the database according to the requests. The server protects you from failures by guaranteeing that complete transactions are recovered in case of a failure and incomplete transactions are rolled back.

UltraLite has the same client/server architecture as other SQL database systems. However, the UltraLite database engine is not a separate process, but is instead a library of functions that is called by an application. If you build your application using C/C++, this engine can be accessed either as a DLL or from a statically linked library. If you build your application in Java, the engine is accessed from Java byte code stored in a JAR file.

C/C++ deployment

If you build your application using C/C++, the UltraLite development tools generate C/C++ source code that is compiled along with your application source code. When you link your application, you link all of the compiled C/C++ together with the UltraLite runtime library or imports library. The result is a single executable file containing application logic and database logic required by the application.



When first executed on a new device, this executable automatically creates the UltraLite database for your application. This database is initially empty, but you can add data, either explicitly or through synchronization with a central database.

Java deployment

If you build a Java UltraLite application, the UltraLite development process generates Java source code that represents the database schema. The generated source file is to be compiled into classes that you deploy as part of your application with the UltraLite runtime JAR file. You may wish to package all files together into a single JAR file for ease of deployment.

When first executed on a new device, the UltraLite runtime automatically creates the database for your application. This database is initially empty, but you can add data, either explicitly or through synchronization with a central database.

Persistent memory

UltraLite provides protection against system failures. Some UltraLite target devices have no disk drive, but instead feature memory that retains content when the device is not running. The storage mechanism for the UltraLite database is platform-dependent, but is managed by the UltraLite runtime library, and does not need explicit treatment from the application developer.

Fixed schema

UltraLite does not allow the schema of an UltraLite database to be modified once the application is deployed. When a newer version of the application requires more tables or more columns, the newer version of the application is deployed and the UltraLite database is repopulated through synchronization.

UltraLite development tools

The UltraLite development tools supplement a supported C/C++ or Java development tool. They manage the generation of the data access code for your application.

During UltraLite application development you create an Adaptive Server Anywhere **reference database**, which is a model of your UltraLite database. You use the **UltraLite generator**, which uses the reference database to create the data access and data management code for your application.

Parameterized SQL statements

The SQL statements used in the application must be determined at compile time. In other words, you cannot dynamically construct a SQL statement within an UltraLite application and execute it. However, SQL statements in UltraLite applications can use placeholders or host variables to adjust their behavior.

MobiLink synchronization

Many mobile and embedded computing applications are integrated into an information infrastructure. They require data to be uploaded to a **central database**, which consolidates all the data throughout the MobiLink installation, and downloaded from a consolidated database. This bi-directional sharing of information is called **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. The UltraLite runtime 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 for synchronization.

Subset of the central database

Mobile and embedded databases may not contain all the data that exists in 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

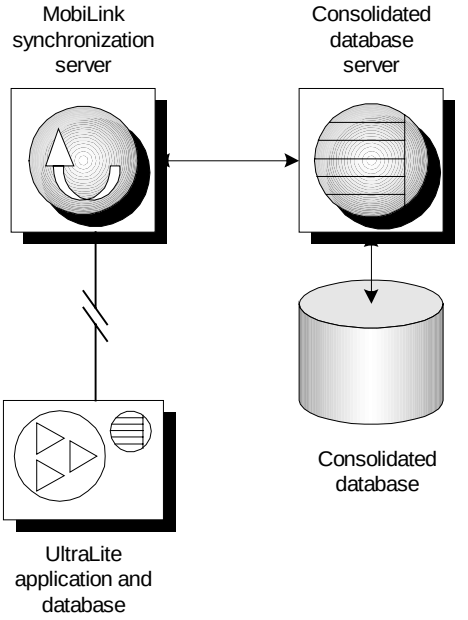
MobiLink synchronization is flexible. You define the subset of data using the native SQL dialect of the consolidated database-management system. Tables in the UltraLite database can correspond to tables in the consolidated database, but you can also populate an UltraLite table from a consolidated table 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 central DBMS.

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

Synchronization occurs through a synchronization **stream**. Supported streams include TCP/IP, HTTP, HTTPS, HotSync, Scout Sync, and ActiveSync. Regardless of the stream, you control the synchronization process using the same SQL scripts defined in your consolidated database.

For a detailed introduction to MobiLink synchronization, see "Synchronization Basics" on page 9 of the book *MobiLink Synchronization User's Guide*.

For information on adding synchronization to your UltraLite application, see "Designing synchronization for UltraLite applications" on page 55.

Enterprise productivity

UltraLite was designed for development using existing tools, skills, and components. You can thus leverage the current capabilities of your organization.

High-level programming and portability

UltraLite encourages high productivity by providing robust, high-level programming solutions on an increasing variety of devices. You can develop applications for small devices using the proven and powerful methodology of full-featured SQL. You need not become familiar with device-specific aspects (such as flash memory) and the disparate operating system interfaces that provide access to them. Similarly, synchronization can be achieved without becoming an expert in the various transmission protocols. Moreover, the database and synchronization components in your application are portable.

High-level development environments

UltraLite allows you to continue to use whatever tools you already use for productive development. It adds functionality to your development process. For example, you can develop applications using Microsoft Visual C++ and test them in the various Windows environments before deploying them, for final testing, on a specific device.

Industrial strength

MobiLink synchronization allows UltraLite applications to synchronize with many widely-used databases, not just those from Sybase. Built with the established technology of Sybase Adaptive Server Anywhere, it uses mature and proven database technology.

CHAPTER 2

Tutorial: 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.

Contents

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 "Introduction 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.

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.

The CustDB sample application

Versions of the CustDB application are supplied for each supported operating system. Also, source code for the application is provided in embedded SQL, the C++ API, and Java. This tutorial uses the compiled version of the application for Windows, the Palm Computing Platform, and Windows CE.

For information about the Java version of the sample application, see "The UltraLite Java sample application" on page 339.

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 8.0 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 different executable is supplied for each operating system. These executables are held in subdirectories of your *ultralite* directory named for the operating system. Each operating system directory has a separate subdirectory for each supported CPU, and the executable files are located in these subdirectories.
  - **ce** Windows CE applications.
  - **palm** Palm Computing Platform applications.
  - **vxw** VxWorks applications.
  - **win32** A Windows application.
  - **java** A Java application.

  This chapter uses the win32, ce, and palm versions of the application.

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 8.

  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.

- **Application source** The source code for the sample application is in two parts.

- The user interface code for each platform is held in a separate subdirectory of *Samples\UltraLite\CustDB*, named for each supported development tool.
- The data access code is help in the *Samples\UltraLite\CustDB* subdirectory of your UltraLite directory.
  - The embedded SQL data access code is held in *custdb.sqc.*
  - The C++ API data access code is held in *custdbapi.cpp.*

  For information on the UltraLite development process, see "Designing UltraLite Applications" on page 41.

For a list of supported development tools, see "Supported platforms" on page 6. For information on building the application for each supported platform, see the following locations:

- **Palm Computing Platform (CodeWarrior)** "Building the CustDB sample application from CodeWarrior" on page 258.
- **Palm Computing Platform (PRC Tools)** "Building the CustDB sample application with PRC Tools" on page 259.
- **Windows CE** "Building the CustDB sample application" on page 296.
- **VxWorks** "Building the CustDB sample application" on page 312.

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

For an overview of the synchronization process, see "The synchronization process" on page 24 of the book *MobiLink Synchronization User's Guide*.

For a description of how to write the synchronization scripts that control synchronization, see "Writing Synchronization Scripts" on page 47 of the book *MobiLink Synchronization User's Guide.*

For information on the techniques used in the CustDB sample application, see "The CustDB sample" on page 361 of the book *MobiLink Synchronization User's Guide*.

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 8➤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➤Sybase SQL Anywhere 8➤MobiLink➤Synchronization Server Sample.

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

☞ For the next step, see the section for your platform under "Lesson 2: Install the sample application to your target platform" on page 21.

Lesson 2: Install the sample application to your target platform

The UltraLite application must be installed onto a target machine in order for you to proceed. This section describes how to install the sample application to your target machine.

No installation needed for Windows
If you want to run the Windows version of the CustDB application from your desktop, you do not need to read this section.

For the next step, see "Lesson 3: Start the sample application and synchronize" on page 24.

Install the sample application (Palm Computing Platform)

You need to carry out the steps in this section only if you wish to run the tutorial on a Palm Computing device. If you wish to run the tutorial on a desktop Windows machine, proceed to "Lesson 3: Start the sample application and synchronize" on page 24.

Use the Palm Install Tool to deploy the application to your device. You must tell Install Tool the location of the application, and then use HotSync to transfer the executable file to the device.

The Adaptive Server Anywhere installation automatically sets registry entries to enable CustDB synchronization via HotSync. These entries associate the UltraLite conduit on Windows with the CustDB application on the Palm Computing device. You must have HotSync Manager 3 installed for HotSync synchronization to work properly.

❖ **To install the sample application to a Palm Computing device:**

1. Prepare your PC for running Adaptive Server Anywhere and the Palm Desktop software.

   Install Adaptive Server Anywhere onto a machine that has Palm Desktop already installed. The Adaptive Server Anywhere installation then adds the registry entries required for HotSync.

2. On your PC, start Palm Desktop.

3. Add the sample application to the list of files to install onto your handheld.

Click Install on the Palm Desktop toolbar.

Click Add. Locate the Palm Computing executable file for the sample application. This executable is the file *custdb.prc*, and it is in the *UltraLite\palm\68k* subdirectory of your SQL Anywhere directory.

Click Done.

4 HotSync your Palm Computing device.

The CustDB application is copied into the Applications view on your Palm device.

❖ **To copy the sample application to a Palm Computing Platform emulator:**

1 Start the Palm emulator.

2 Right click and select Install Application/Database from the popup menu.

3 Locate the application and click OK to install.

You may have to refresh the Applications view to see the installed application on the emulator interface.

For the next step, see "Lesson 3: Start the sample application and synchronize" on page 24.

Install the sample application (Windows CE)

You need to carry out the steps in this section only if you wish to run the tutorial on a Windows CE device. If you wish to run the tutorial on a desktop Windows machine, proceed to "Lesson 3: Start the sample application and synchronize" on page 24.

❖ **To copy the sample application to a CE device:**

1 Ensure that Windows CE Services is properly installed. You can do this by checking whether Windows Explorer can view and modify the file system on the CE device.

2 Open the Windows Explorer. Right click on My Computer and select Explore.

3 Find your SQL Anywhere installation directory. The default location is as follows:

```
c:\Program Files\Sybase\SQL Anywhere 8
```

Open the *ultralite\ce* folder so that the contents of the folder are visible. Open processor-specific folder depending on the CPU used in the CE device. The executable file *CustDB.exe* should be visible at this point.

4 Right click on *CustDB.exe* and select Copy.

5 Find the CE device in the Explorer hierarchy. It should be under the *Mobile Devices* folder. Create a folder on the CE device called *Sybase*. Open the *Sybase* folder on the device and paste the executable to copy the *CustDB.exe* file to the CE device. You may be asked whether you want the file to always be synchronized or converted before copying, both of these actions can be refused.

6 In the *Sybase* folder on the device, there should be one executable file, namely, *CustDB.exe*. Right click on *CustDB.exe* to create a shortcut and rename the shortcut to *CustDB*.

7 On the CE Device, open the root folder *Windows* followed by *Start Menu* and *Programs*. In the *Programs* folder, add a *Sybase* folder.

8 Move the shortcut for *CustDB* from *\Sybase* to the *\Windows\Start Menu\Programs\Sybase* folder by cutting and pasting the shortcut.

9 The samples are now accessible from the CE device Start button. Select Start➤Programs➤Sybase➤CustDB.

☞ For the next step, see "Start the application (Windows CE)" on page 26.

Lesson 3: 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 (Windows)

❖ **To start and synchronize the sample application:**

1. Launch the sample application.

   From the Start menu, choose Programs➤Sybase SQL Anywhere 8➤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 4: Add an order" on page 28.

Start the sample application (Palm Computing Platform)

The sample application for the Palm Computing Platform uses HotSync as the synchronization mechanism.

❖ **To start and synchronize the sample application:**

1. Place your Palm device into its cradle.

When you start the sample application for the first time it must be able to synchronize, to download an initial copy of the data. This step is required only the first time you start the application. After that, the downloaded data is stored in the UltraLite database.

2 Launch the sample application.

From the Applications view, tap CustDB. An initial dialog displays, prompting you for an employee ID.

3 Enter an employee ID.

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

A message box tells you that you must synchronize before proceeding.

4 Synchronize your application.

Use HotSync to obtain an initial copy of the data.

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

From the Applications view, tap the CustDB application. The display shows an entry sheet for a customer, with entries.

You have now synchronized your data.

For the next step, see "Lesson 4: Add an order" on page 28.

Installing the HotSync conduit

HotSync synchronization requires a MobiLink HotSync conduit to be installed on the desktop computer. The SQL Anywhere setup program automatically installs a conduit for the CustDB sample application. For other applications, you would have to install the conduit yourself using the *dbcond8* command-line utility. The command line used to install the conduit for the CustDB sample application is as follows:

```
dbcond8 -n CustDB Syb2
```

where CustDB is the name which HotSync Manager displays, and Syb2 is the Palm creator ID for the application.

❖ **To start and synchronize the sample application (Palm Emulator):**

1 Start the emulator.

2 Ensure the emulator is set up for TCP/IP synchronization:

- Right-click the emulator and choose Settings➤Properties from the popup menu.
- In the Properties dialog, check Redirect NetLib Calls to Host TCP/IP.

3 Launch the sample application.

From the Applications view, tap CustDB. An initial dialog displays, prompting you for an employee ID.

4 Enter an employee ID.

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

A message box tells you that you must synchronize before proceeding.

5 Set the application to synchronize using TCP/IP:

- Tap the Options menu.
- Set the Conduit to **Disabled**.
- Set the Synch Method to **TCP/IP**.
- Leave the Synch Parameters as **host=localhost**.
- Tap OK.

6 Synchronize your application:

- Tap the Synchronize menu. A set of data is uploaded and downloaded.

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

From the Applications view, tap the CustDB application. The display shows an entry sheet for a customer, with entries.

Start the application (Windows CE)

For synchronization to succeed, you must have the consolidated database server and a MobiLink synchronization server running when you start the sample application.

❖ **To start and synchronize the sample application:**

1 Connect your Windows CE device to your PC.

When you start the sample application for the first time, it must be able to connect to the MobiLink synchronization server and download an initial copy of the data. This step is required only the first time you start the application. Once you have downloaded the data, it is stored in the UltraLite database.

2 Launch the sample application.

On the CE device, choose Start➤Programs➤Sybase➤CustDB.

3 Enter an employee ID.

When running through this section as a tutorial, enter a value of 50 and press ENTER. The application does also allow 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 your machine.

4 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 4: Add an order" on page 28.

Lesson 4: 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 (Windows or Windows CE)

The procedure is the same for Windows CE as for other Windows operating systems.

❖ **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 5: Act on some existing orders" on page 31.

Add an order (Palm Computing Platform)

❖ **To add an order:**

1 Scroll through the outstanding orders.

Tap the Down arrow in the bottom right corner to display the next customer.

2 Open the window to enter a new order.

Tap 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 Add the new order.

Tap Add to add the order.

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 5: Act on some existing orders" on page 31.

Lesson 5: 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 6: Synchronize your changes" on page 32.

Lesson 6: 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 20.

Synchronize your changes (Windows, Windows CE)

❖ **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 7: Confirm the synchronization at the consolidated database" on page 34.

Synchronize your changes (Palm Computing Platform)

❖ **To synchronize your changes:**

1. Place the Palm device in its cradle.
2. Press the HotSync button to synchronize.
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 7: Confirm the synchronization at the consolidated database" on page 34.

Lesson 7: 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 8.0 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.

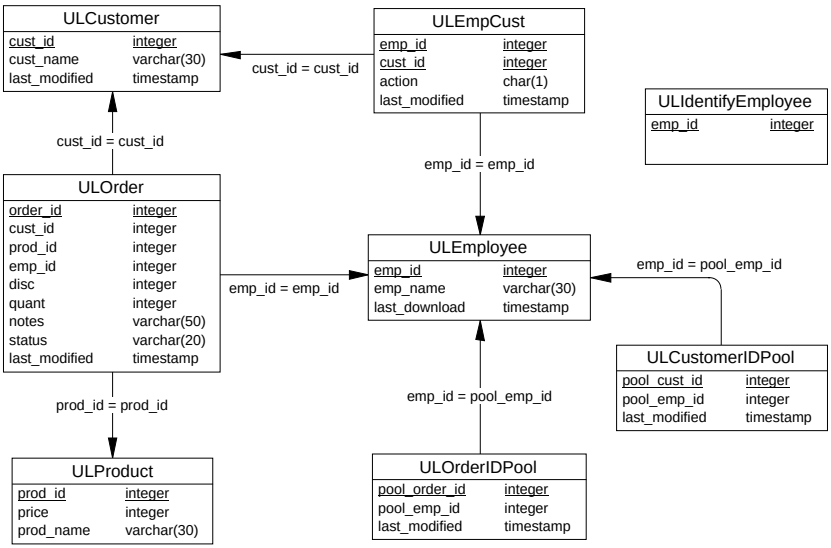
Lesson 8: Browse the consolidated database

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.

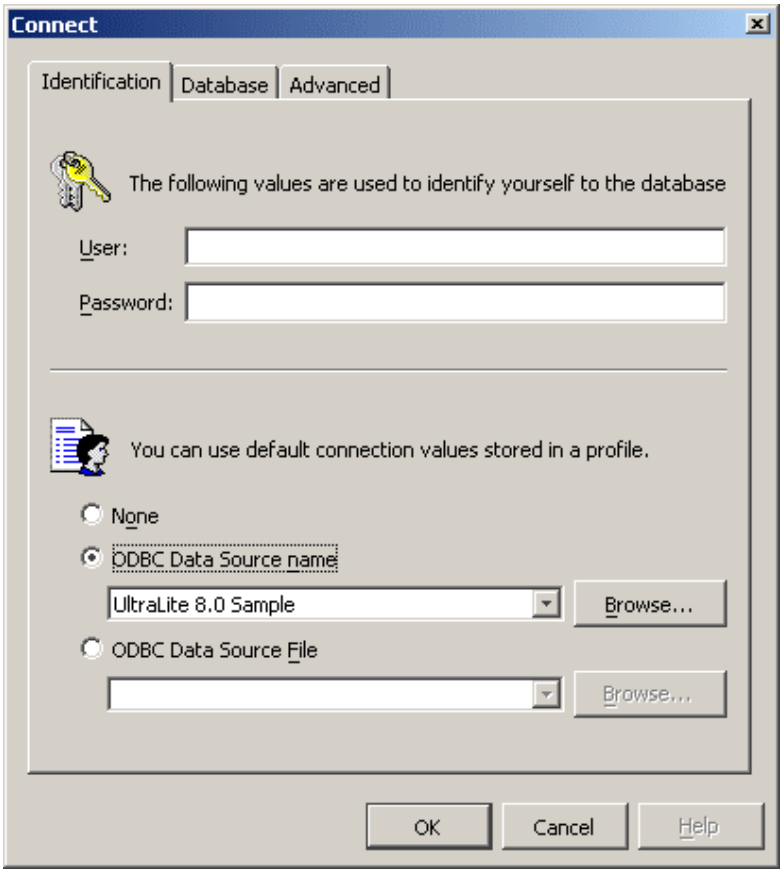
- **ULCustomer** A list of customers.
- **ULProduct** A list of products.
- **ULEmployee** A list of sales employees. This table is not present in the UltraLite database.
- **ULEmpCust** A many-to-many relationship between employees and customers. This table is not present in the UltraLite database.
- **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 8➤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 8➤ 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.

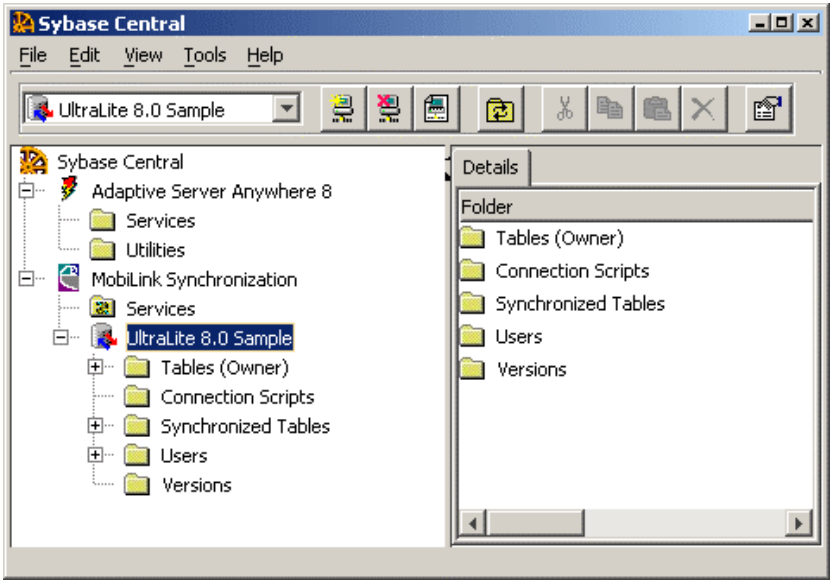


Select ODBC, enter **UltraLite 8.0 Sample** in the Data Source box. Click OK to connect.

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" on page 47 of the book *MobiLink Synchronization User's Guide*.

Summary

In this tutorial you learned how to:

- Copy an UltraLite application to a Palm or Windows CE device.
- Start and work with the sample application on a Palm or Windows CE device, or on Windows.
- Synchronize changes with a consolidated database, via the MobiLink synchronization server.
- Use Sybase Central to view the scripts controlling the synchronization process.

The CustDB sample application is used throughout the current book and in the *MobiLink Synchronization Guide* to illustrate programming methods and synchronization techniques.

CHAPTER 3

Designing UltraLite Applications

About this chapter

This chapter describes the features you can include in UltraLite applications, as well as some general design features that apply to all UltraLite applications.

The chapter also provides a description of UltraLite internals as background information for application design.

Contents

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 44.

UltraLite database internals

This section gives an outline of how data is stored in an UltraLite database, and how UltraLite maintains data in a database.

Database storage mechanism

Each UltraLite application has its own database. The UltraLite runtime creates the database when you first start your application. The physical storage of the UltraLite database depends on the target platform. In all cases, except for Java, the database is persistent by default: it continues to exist when the application is not running.

- On the Palm Computing Platform, the UltraLite 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.
- On Windows, the UltraLite 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.
- On VxWorks, UltraLite requires a dosFs (MS-DOS-compatible file system) device or a functionally equivalent device to store the persistent data file. UltraLite defaults to using a device named *ULDEV:*, and a filename *ul_<project>.udb*, where *<project>* is the UltraLite project name, and the filename is truncated to an 8.3 format. You can configure a storage device with this name and it will be used to store persistent data for the application, or you can override the default filename and specify a different device.
- In Java applications, the database is either transient, or is stored as a file in the file system. By default, it is transient.

For information on configuring UltraLite databases, see "Configuring and managing database storage" on page 45. For information on configuring UltraLite Java databases, see "UltraLite JDBC URLs" on page 346.

The information in an UltraLite database

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.

In C/C++ UltraLite 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.

Configuring and managing database storage

You can control several aspects of UltraLite persistent storage for C/C++ applications. The following aspects can be configured:

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

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

For more information, see "UL_STORE_PARMS macro" on page 428.

Encrypting UltraLite databases

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

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

  For more information, see "Obfuscating an UltraLite database" on page 46.

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

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

For more information, see "Encrypting an UltraLite database" on page 46, and "Changing the encryption key for a database" on page 49.

Obfuscating an UltraLite database

❖ **To obfuscate an UltraLite database (C/C++):**

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

  For more information, see "UL_ENABLE_OBFUSCATION macro" on page 427.

❖ **To obfuscate an UltraLite database (Java):**

- Add the following line to your code before creating the database (that is, before connecting to the database for the first time):

```
UlDatabase.setDefaultObfuscation( true );
```

Encrypting an UltraLite database

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

Encryption for C/C++ programs

❖ **To strongly encrypt an UltraLite database (C/C++):**

1. Load the encryption module.

   Call **ULEnableStrongEncryption** before opening the database.

   You open a database by calling **db_init** (embedded SQL) or **ULData::Open** (C++ API).

   On the Palm Computing Platform, you open a database by calling **ULPalmLaunch** or **ULData::PalmLaunch**.

2 Specify the encryption key.

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

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

As with most passwords, it is best to choose a key value that cannot be easily guessed. The key can be of arbitrary length, but generally the longer the key, the better because a shorter key is easier to guess than a longer one. As well, including a combination of numbers, letters, and special characters decreases the chances of someone guessing the key.

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

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

For more information on UL_STORE_PARMS, see "UL_STORE_PARMS macro" on page 428.

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

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

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

Here is a code snippet from the sample:

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

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

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

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

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

Encryption for Java programs

❖ **To strongly encrypt an UltraLite database (Java):**

1 Set a property named key before creating a database object for the first time.

Here is a code fragment that reads the encryption key from the command line.

```
InputStreamReader isr = new InputStreamReader(
System.in );
BufferedReader br = new BufferedReader( isr );
String key = null ;
System.out.print( "Enter encryption key:" );
key = br.readLine() ;
System.out.println( "The key is: " + key );

// (3) Connect to the database
java.util.Properties p = new java.util.Properties();
p.setProperty( "persist", "file" );
p.setProperty( "key", key );
SampleDB db = new SampleDB( p );
```

Here, SampleDB is the database filename as supplied in the UltraLite generator `-f` command-line option.

For more information, see "The UltraLite generator" on page 419, and "Using a Properties object to store connection information" on page 347.

2 Create the database object using the properties.

For example:

```
Connection conn = db.connect();
```

After the first connection attempt, subsequent attempts to access the database produce an Incorrect or missing encryption key SQLException if the wrong key is supplied.

You can find a sample Java application demonstrating encryption in the directory *\Samples\UltraLite\JavaSecurity*. The encryption code is held in *\Samples\UltraLite\JavaSecurity\Sample.java*.

Here is a code snippet from the sample:

```
// Obtain the encryption key
InputStreamReader isr = new InputStreamReader( System.in );
BufferedReader br = new BufferedReader( isr );
String key = null ;
System.out.print( "Enter encryption key:" );
key = br.readLine() ;
System.out.println( "The key is: " + key );

java.util.Properties p = new java.util.Properties();
p.setProperty( "persist", "file" );
p.setProperty( "key", key );
SampleDB db = new SampleDB( p );
Connection conn = db.connect();
```

Changing the encryption key for a database

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

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

❖ **To change the encryption key on an UltraLite database (C/C++):**

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

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

  For more information, see "ULChangeEncryptionKey function" on page 233.

❖ **To change the encryption key on an UltraLite database (Java):**

- Call **changeEncryptionKey** on the database object, supplying the new key as an argument.

```
db.changeEncryptionKey( "new key" );
```

  For more information, see "changeEncryptionKey method" on page 369.

Using the encryption key on the Palm Computing Platform

If you encrypt an UltraLite database on the Palm Computing Platform, you are prompted to re-enter the key each time you launch the application. This section describes how to add code that circumvents the re-entering of the key. The feature is currently available only for embedded SQL applications.

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

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

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

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

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

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

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

For more information, see "ULClearEncryptionKey function" on page 233, "ULRetrieveEncryptionKey function" on page 247, and "ULSaveEncryptionKey function" on page 248.

Defragmenting UltraLite databases

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

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

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

For embedded SQL reference information, see "ULStoreDefragFini function" on page 249, "ULStoreDefragInit function" on page 249, and "ULStoreDefragStep function" on page 250. The embedded SQL functions can also be called from the C++ API.

For the Java interface to this feature, see "Class JdbcDefragIterator" on page 373.

❖ **To defragment UltraLite databases (C/C++):**

1 Obtain a p_ul_store_defrag_info information block. For example,

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

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

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

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

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

❖ **To defragment UltraLite databases (Java):**

1 Cast a **Connection** to a **JdbcConnection** object. For example,

```
…
Connection conn = db.connect();
JdbcConnection jconn = (JdbcConnection)conn ;
```

2 Call **getDefragIterator()** to obtain a **JdbcDefragIterator** object. Foe example:

```
JdbcDefragIterator defrag =
jconn.getDefragIterator();
```

3 During idle time, call **ulStoreDefragStep()** to defragment a piece of the database.

```
defrag.ulStoreDefragStep();
```

Example

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

```
p_ul_store_defrag_info  DefragInfo;

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

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

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

Choosing an UltraLite development model

There are three methods of 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.
- **Java** 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 programmming. 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.

Designing synchronization for UltraLite applications

UltraLite applications use MobiLink synchronization technology to share data with a consolidated database and integrate into an enterprise information system.

In the simplest scenario, an UltraLite application synchronizes all its data with the consolidated database. However, additional options are possible:

- **Synchronize subsets of your data** You can identify portions of the data named **publications** in your UltraLite application to be synchronized separately. Publications allow you to separate high-priority items from lower-priority data.

  For more information, see "Designing sets of data to synchronize separately" on page 76.

- **Mark data for download only** You can carry out download-only synchronizations. By combining this with publications, UltraLite applications can get updates from the consolidate database efficiently, and upload changes at a convenient time.

- **Exclude tables from synchronization** You can maintain data on the UltraLite database that is excluded from synchronization.

  For more information, see "Including non-synchronizing tables in UltraLite databases" on page 76.

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" on page 3 of the book *MobiLink Synchronization User's Guide*.
- For an introduction to synchronization concepts, see "Synchronization Basics" on page 9 of the book *MobiLink Synchronization User's Guide*.
- For information about synchronization techniques, see "Synchronization Techniques" on page 83 of the book *MobiLink Synchronization User's Guide*.

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 "Adding synchronization to your application" on page 94.

Supported synchronization streams

The following synchronization streams are supported:

Synchronization stream	Supported languages and platforms
TCP/IP	C/C++ and Java. All platforms.
HTTP	C/C++ and Java. All platforms.
HotSync	Palm Computing Platform only.
ScoutSync	Palm Computing Platform only.

☞ For a list of the supported target platforms under C/C++ and Java, see "Supported platforms" on page 6.

The HotSync synchronization stream is the method used by many Palm OS applications. ScoutSync is a synchronization technology licensed by Palm Computing for incorporation into their HotSync Server.

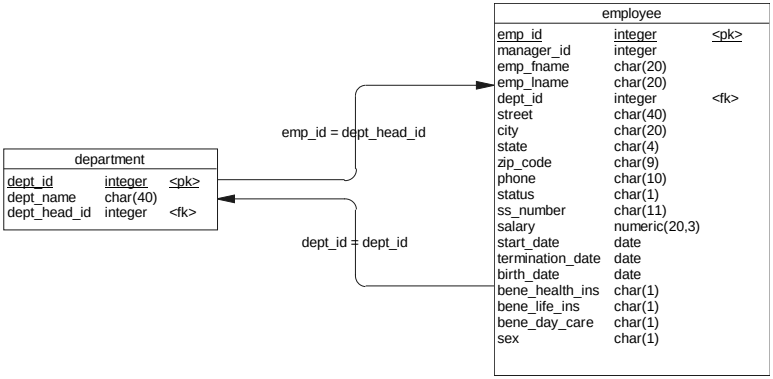
Foreign key cycles

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;
```

Global autoincrement default column values

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 58.

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 59.

For information on using global autoincrement values in Adaptive Server Anywhere remote databases, see "Maintaining unique primary keys using global autoincrement" on page 96 of the book *MobiLink Synchronization User's Guide*.

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" on page 350 of the book *ASA SQL Reference Manual*.

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.

❖ **To set the global database identifier (embedded SQL):**

- Call the **ULSetDatabaseID** function. This function takes an argument that indicates the identification number.

```
int n = 123;
ULSetDatabaseID( &sqlca, n );
```

For more information, see "ULSetDatabaseID function" on page 248.

❖ **To set the global database identifier (C++ API):**

- Call the **ULConnection::SetDatabaseID** method. This method takes a single integer argument that indicates the identification number.

```
int n = 123;
conn.SetDatabaseID( n );
```

For more information, see "SetDatabaseID method" on page 142.

❖ **To set the global database identifier (Java):**

1. Call the **JdbcConnection.setDatabaseID** method. This method takes a single argument, which is the integer global identification value.

```
int n = 123;
conn.setDatabaseID( n);
```

For more information, see "setDatabaseID method" on page 368.

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 59.

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 62.

For information on maintaining primary key uniqueness using explicit primary key pools, see "Maintaining unique primary keys" on page 95 of the book *MobiLink Synchronization User's Guide*.

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.

❖ **To find out the percent of values used (embedded SQL):**

- Retrieve the proportion of used default values by calling the **ULGlobalAutincUsage** function. This procedure takes no arguments. It returns the maximum percent of used default values as a short in the range 0–100.

  ```
  short p;
  p = ULGlobalAutincUsage( );
  ```

  For more information, see "ULGlobalAutoincUsage function" on page 241.

❖ **To find out the percent of values used (C++ API):**

- Retrieve the proportion of used default values by calling the **ULConnection::GlobalAutincUsage** method. This method takes no arguments. It returns the maximum percent of used default values as a short in the range 0–100.

```
short p;
p = conn.GlobalAutincUsage( );
```

For more information, see "GlobalAutoincUsage method" on page 136.

❖ **To find out the percent of values used (Java):**

- Retrieve the proportion of used default values by calling the **JdbcConnection.globalAutincUsage** method. This method takes no arguments. It returns the maximum percent of used default values as a short in the range 0–100.

For more information, see "globalAutoincUsage method" on page 367.

Character sets in UltraLite

An UltraLite application uses 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.

Multi-byte platforms

An UltraLite application uses 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.

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.

Synchronization

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.

Palm Computing Platform

At the time of printing, all single-byte Palm Computing Platform devices uses a character set based on code page 1252 (the Windows US code page). The default Adaptive Server Anywhere collation sequence (1252Latin1) is appropriate for developing applications for the Palm Computing Platform. 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.

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.

CHAPTER 4

Developing UltraLite Applications

About this chapter

This chapter presents an overview of the UltraLite development process.

UltraLite applications can be developed using either C/C++ or using Java. Later chapters in the book describe the specifics of each approach. This chapter describes aspects common to all UltraLite application development.

Contents

Introduction

UltraLite development models

UltraLite supports the following **development models**:

- C++ applications using the UltraLite C++ API.
- C/C++ applications using embedded SQL.
- Java applications.

The overall development process for each model is similar, but the details are different. This chapter describes those aspects of development that are similar among the development models. It should be used together with the chapter on the particular development model you are using:

- "Developing C++ API Applications" on page 121
- "Developing Embedded SQL Applications" on page 193
- "Developing UltraLite Java Applications" on page 337

Host and target platforms

UltraLite applications are developed on a **host platform**, and deployed on a **target platform**. The host platform is PC-based, and the target platform is generally a handheld or embedded device.

To create an UltraLite application, you need to use a development tool or compiler that supports your target platform, together with the UltraLite development tools. For example, you may want to use Metrowerks CodeWarrior for Palm OS development, or Microsoft Visual C++ for Windows CE development.

For information on supported host platforms, target platforms, and development tools, see "Supported platforms" on page 6.

For information specific to each target platform, see the following chapters:

- "Developing Applications for the Palm Computing Platform" on page 253.
- "Developing Applications for Windows CE" on page 293.
- "Developing Applications for VxWorks" on page 309.
- "Developing UltraLite Java Applications" on page 337.

Multi-threaded applications

You can develop multi-threaded UltraLite applications on those C/C++ platforms that support it (Windows, Windows CE, and VxWorks). You cannot develop multi-threaded UltraLite Java applications.

For more information, see "Developing multi-threaded applications" on page 93.

The UltraLite development environment

When developing UltraLite applications, you will be working with 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 72.

- **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 on supported application development tools, see "Supported platforms" on page 6.

- **UltraLite development tools** UltraLite includes several tools for development.

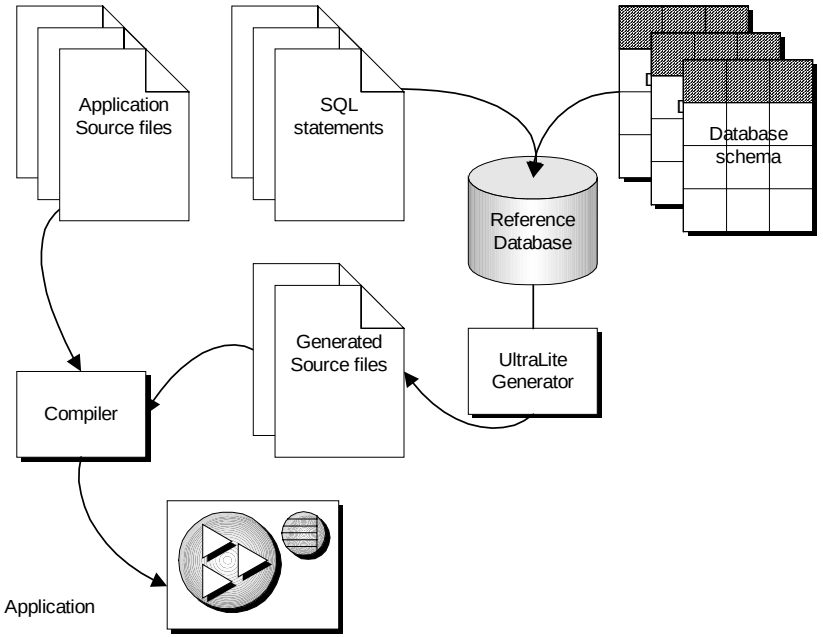
  - **The UltraLite generator** This application uses Java classes in the reference database to generate source code that implements the underlying query execution, data storage, and synchronization features of your application. The generator is required for all kinds of UltraLite development. The Java classes in the database are called the **UltraLite Analyzer**.

  - **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 UltraLite development process

The basic features of the development process are common to all development models. 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 development model 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. Adding synchronization to your application is a straightforward task.

For information on how to add synchronization to your application, and the kinds of synchronization available, see "Adding synchronization to your application" on page 94.

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 74.

- **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 74.

- **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 80.

- **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 76. For information on using publications for C++ API applications, see "Defining UltraLite tables" on page 123.

- **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 73.

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 74.

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" on page 3 of the book *ASA SQL User's Guide*.

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" on page 429 of the book *ASA SQL User's Guide*.

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. The database must be Java-enabled, which is the default setting.

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" on page 449 of the book *ASA SQL User's Guide*.

Designing your UltraLite database

The tables to be included in your UltraLite database are defined by the SQL statements you add to your reference database or, if you use publications and the C++ API development model, by the publications you add to your reference database.

The indexes to be included in your UltraLite database are also determined by the indexes defined in the reference database.

This section describes other aspects of UltraLite database design, including non-synchronizing tables, separate data sets for synchronization such as high-priority synchronization, and read-only tables.

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 76.

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 or ScoutSync 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 77.

2 Run the UltraLite generator, specifying the publications on the -v command-line option.

For more information, see "The UltraLite generator" on page 419.

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 "publication synchronization parameter" on page 386.

Creating publications for UltraLite databases

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 or ScoutSync 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" on page 314 of the book *ASA SQL Reference Manual*.

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 77, and "upload_only synchronization parameter" on page 396.

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 "download_only synchronization parameter" on page 383.

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.

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 SQL Remote publications to define data access methods. For information on using publications, see "Defining UltraLite tables" on page 123.

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. From Sybase Central, connect to your reference database.

   For instructions on using Sybase Central, see "Connect to the sample database" on page 53 of the book *ASA Getting Started*.

2. In the left pane, open the database container.
3. In the left pane, open the UltraLite Projects folder.
4. In the right pane, double-click Add UltraLite Project.

   The UltraLite Project Creation wizard appears.

5. Enter an UltraLite project name and click OK to create the project in the database.

   For information on UltraLite project naming rules, see "ul_add_project system procedure" on page 412.

❖ **To create an UltraLite project (SQL):**

♦ 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 412.

❖ **To create an UltraLite project (embedded SQL):**

♦ If you are using the embedded SQL development model, 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 "Preprocessing your embedded SQL files" on page 201.

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" on page 7 of the book *ASA SQL Reference Manual*.

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 development model, 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 SQL Remote publications to define data access methods. For information on using publications, see "Defining UltraLite tables" on page 123.

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 From Sybase Central, connect to your reference database.

☞ For instructions on using Sybase Central, see "Connect to the sample database" on page 53 of the book *ASA Getting Started*.

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 Double-click Add 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 83.

❖ **To add a SQL statement to an UltraLite project (SQL):**

♦ 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 411.

❖ **To add a SQL statement to an UltraLite project (embedded SQL):**

♦ If you are using the embedded SQL development model, specify the UltraLite project name on the SQL Preprocessor command line.

No statement name is used in embedded SQL development.

☞ For more information, see "Preprocessing your embedded SQL files" on page 201.

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 "SQL features and limitations of UltraLite applications" on page 437.

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" on page 9 of the book *ASA SQL Reference Manual*.

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" on page 209.

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.

Adding user authentication to your application

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

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

For a general description of UltraLite user authentication, see "User authentication for UltraLite databases" on page 442.

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

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

Enabling user authentication

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

❖ **To enable user authentication (embedded SQL):**

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

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

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

❖ **To enable user authentication (C++ API):**

1 Define the compiler directive UL_ENABLE_USER_AUTH when compiling **ulapi.cpp**.

2 Call **ULEnableUserAuthentication** before opening the database. For example:

```
ULData db;
…
ULEnableUserAuthentication( &sqlca );
db.open();
…
```

❖ **To enable user authentication (Java):**

♦ Call the **JdbcSupport.enableUserAuthentication** method before creating a new database object: For example:

```
JdbcSupport.enableUserAuthentication();
java.util.Properties p = new java.util.Properties();
p.put( "persist", "file" );
SampleDB db = new SampleDB( p );
```

☞ Once you have enabled user authentication, you must add user management code to your application. For more information, see "Managing user IDs and passwords" on page 86.

Managing user IDs and passwords

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

1 New users have to be added from an existing connection. As all UltraLite databases are created with a default user ID and password of **DBA** and **SQL**, respectively, you must first attempt to connect as this initial user and implement user management only upon successful connection.

2 You cannot change a user ID: you add a user and delete an existing user. A maximum of four user IDs are permitted for each UltraLite database.

3 To change the password for an existing user ID, call the same function as adding a user ID. This function is **ULGrantConnectTo** (embedded SQL), **ULConnection.GrantConnectTo** (C++ API), or **JdbcDatabase.grant** (Java).

Palm Computing Platform

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

Embedded SQL user authentication example

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

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

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

The code carries out the following tasks:

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

☞ For more information, see "ULGrantConnectTo function" on page 242, and "ULRevokeConnectFrom function" on page 248.

C++ API user authentication example

The following code fragment performs user management and authentication for a C++ API UltraLite application.

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

```
ULEnableUserAuthentication( &sqlca );
db.Open() ;
   if( conn.Open( &db,
                  UL_TEXT( "dba" ),
                  UL_TEXT( "sql" ) ) ){
      // prompt for new user ID and password
      printf("Enter new user ID and password\n" );
      scanf( "%s %s", uid, pwd );
      if( conn.GrantConnectTo( uid, pwd ) ){
         // new user added, remove dba
         conn.RevokeConnectFrom( UL_TEXT("dba") );
      }
      conn.Close();
   }
   // regular connection
   printf("Enter user ID and password\n" );
   scanf( "%s %s", uid, pwd );
   if( conn.Open( &db, uid, pwd )){
   …
```

The code carries out the following tasks:

1 Initiate database functionality by opening the database object.

2 Attempt to connect using the default user ID and password.

3 If the connection attempt is successful, add a new user.

4 If the new user is successfully added, delete the DBA user from the UltraLite database.

5 Disconnect. An updated user ID and password is now added to the database.

6 Connect using the updated user ID and password.

☞ For more information, see "GrantConnectTo method" on page 137, and "RevokeConnectFrom method" on page 141.

Java user authentication example

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

A complete sample can be found in the *Samples\UltraLite\javaauth* subdirectory of your SQL Anywhere directory. The code below is based on that in *Samples\UltraLite\javaauth\Sample.java*.

```
JdbcSupport.enableUserAuthentication();
// Create database environment
java.util.Properties p = new java.util.Properties();
p.put( "persist", "file" );
SampleDB db = new SampleDB( p );

// Get new user ID and password
try{
   conn = db.connect( "dba", "sql" );
   // Set user ID and password
   // a real application would prompt the user.
   uid = "50";
   pwd = "pwd50";

   db.grant( uid, pwd );
   db.revoke( "dba" );
   conn.close();
}
catch( SQLException e ){
   // dba connection failed - prompt for user ID and
password
   uid = "50";
   pwd = "pwd50";
}

// Connect
conn = db.connect( uid, pwd );
```

The code carries out the following tasks:

1 Opening the database object.

2 Attempt to connect using the default user ID and password.

3 If the connection attempt is successful, add a new user.

4 Delete the default user from the UltraLite database.

5 Disconnect. An updated user ID and password is now added to the database.

6 Connect using the updated user ID and password.

For more information, see "GrantConnectTo method" on page 137, and "RevokeConnectFrom method" on page 141.

Sharing MobiLink and UltraLite user IDs

Although UltraLite user IDs and MobiLink user authentication mechanisms are separate, you may wish to provide your end users with a single user ID and password that provides both MobiLink and UltraLite user authentication. To share user IDs and passwords, store them in variables and use the same variable in the UltraLite user authentication calls and the synchronization call.

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

❖ **To prompt for a new MobiLink/UltraLite password:**

1. Save the user ID and password in variables.
2. Synchronize.
3. If synchronization fails because the user was not authenticated, prompt the user for a new password.
4. Update the UltraLite user's password using the appropriate function or method:
   - **ULGrantConnectTo** (embedded SQL)
   - **Connection.GrantConnectTo** method (C++ API).
   - **JdbcDatabase.grant** method (Java)
5. Update the **synch_info** structure and synchronize again.

☞ For information on MobiLink user authentication, see "Authenticating MobiLink Users" on page 251 of the book *MobiLink Synchronization User's Guide*.

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. It does so using a set of Java classes that run inside your reference database, called the **UltraLite analyzer**.

The UltraLite analyzer generates code that implements data access and storage for your application. It is a single application that 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 analyzer 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 analyzer includes only the features your application uses.

Using the UltraLite generator

The UltraLite generator is a command-line application. It takes a set of command-line options to customize the behavior for each project.

❖ **To run the UltraLite generator:**

- Enter the following command at a command-prompt:

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

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

The UltraLite generator command-line customizes its behavior. The following command-line switches are used across development models:

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

  For information on Adaptive Server Anywhere connection strings, see "Connection parameters" on page 70 of the book *ASA Database Administration Guide*.

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

☞ For more information on UltraLite generator options, see "The UltraLite generator" on page 419.

If you are using embedded SQL, and if you need to run only a single source file through the SQL preprocessor, you can instruct the preprocessor to also run the UltraLite generation process as a shortcut.

☞ For more information, see "Preprocessing your embedded SQL files" on page 201.

Which databases contain the UltraLite analyzer?

The generator relies on a current version of the UltraLite analyzer classes being installed into the reference database. If you have upgraded your UltraLite software, you must also upgrade the reference database so that it contains the new analyzer classes.

Older databases, created with previous versions of Adaptive Server Anywhere, may not contain any version of the analyzer. You can upgrade these older databases using the Upgrade utility.

☞ For more information about upgrading an older Java-enabled database, see "Preparing a reference database" on page 72.

Error on starting the analyzer

Either *sqlpp* or *ulgen* can report the error message Unable to use Java in the database when these utilities are unable to run the analyzer. The UltraLite analyzer is a Java class that is added to your reference database when it is initialized. For the analyzer to run, the database must have been initialized with Java classes and the database engine must be able to start the Java support in Adaptive Server Anywhere.

The following situations may cause this error to happen:

- The database was not initialized with Java classes.
- The Adaptive Server Anywhere database server was not started with a cache of sufficient size. This should not generally be a problem as the database server can dynamically increase its cache size.
- SQL Anywhere Studio was moved to a new directory without uninstalling and reinstalling. In this case, there may be registry entries pointing to the old location.
- There may be mismatched DLLs or mismatched Java jar files. This can happen if you copy files from a maintenance release or emergency bug fix, but miss copying all the files.

Developing multi-threaded applications

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

You can also develop multi-threaded UltraLite Java applications.

Multi-threaded embedded SQL applications

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

For more information, see "db_init function" on page 231.

Multi-threaded C++ API applications

Each thread of a multi-threaded application must make its own objects, including the **ULData**, **ULConnection**, **ULTable**, **ULStatement** and **ULResultSet** objects.

For more information, see "Open method" on page 146.

Multi-threaded UltraLite Java applications

The UltraLite Java runtime library 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.

Connections cannot be shared among threads: each Java thread must obtain its own JDBC connection to the database and statements used by a Java thread for must be created with the thread's own connection. Any one thread can have multiple connections.

For more information, see "Using the UltraLite JdbcDatabase.connect method" on page 344.

Adding synchronization to your application

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

Overview

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

❖ **To add synchronization to your application:**

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

For information, see "Initializing the synchronization parameters" on page 94.

2 Assign the parameter values for your application.

For information, see "Synchronization stream parameters" on page 399.

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

For information, see "Invoking synchronization" on page 96.

You must ensure that there are no uncommitted changes when you synchronize. For more information, see "Commit all changes before synchronizing" on page 97.

Synchronization parameters

Synchronization specifics are controlled through a set of synchronization parameters. For information on these parameters, see "Synchronization stream parameters" on page 399.

Initializing the synchronization parameters

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

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

For a complete list of synchronization parameters, see "Synchronization parameters" on page 380.

❖ **To initialize the synchronization parameters (embedded SQL):**

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

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

❖ **To initialize the synchronization parameters (C++ API):**

- Call the **InitSynchInfo()** method on the **Connection** object. For example:

```
auto ul_synch_info synch_info;
conn.InitSynchInfo( &synch_info );
```

❖ **To initialize the synchronization parameters (Java):**

- Create a **UlSynchOptions** object. For example:

```
UlSynchOptions synch_options = new UlSynchOptions();
```

Once the structure or object is initialized, you must set the values to meet your particular requirements.

For information on the individual parameters, see "Synchronization stream parameters" on page 399.

Setting synchronization parameters: C/C++ examples

For Java examples, see "Initiating synchronization" on page 353.

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

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

The following code fragment initiates TCP/IP synchronization in a C++ API application. The MobiLink user name is `50`, with an empty password, the script version is `custdb`, and the MobiLink synchronization server is running on the same machine as the application (`localhost`), on the default port (2439):

```
auto ul_synch_info synch_info;
conn.InitSynchInfo( &synch_info );
synch_info.user_name = UL_TEXT( "50" );
synch_info.version = UL_TEXT( "custdb" );
synch_info.stream = ULSocketStream();
synch_info.stream_parms =
   UL_TEXT("host=localhost");
conn.Synchronize( &synch_info );
```

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

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

Invoking synchronization

The details of how to invoke synchronization depends on your target platform and programming language, and also on the particular synchronization stream.

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

- When using embedded SQL, call **ULInitSynchInfo** to initialize the synchronization parameters, and call **ULSynchronize** to synchronize.
  or
  When using the C++ API, use the **Connection.InitSynchInfo()** method to initialize the synchronization parameters, and **Connection.Synchronize()** method to synchronize. See "Synchronize method" on page 143.

or

When using Java, construct a new **ULSynchInfo** object to initialize the synchronization parameters, and use the **JdbcConnection.synchronize()** method to synchronize. See "Adding synchronization to your application" on page 352.

❖ **To invoke synchronization (HotSync or ScoutSync streams):**

- In embedded SQL, use **ULInitSynchInfo** to initialize the synchronization parameters, and call **ULPalmExit** and **ULPalmLaunch** functions to manage synchronization.

  or

  In the C++ API, use the **ULConnection.InitSynchInfo** to initialize the synchronization parameters, and call **ULData.PalmExit** and **ULData.PalmLaunch** functions to manage synchronization.

  For more information on the embedded SQL functions, see "ULPalmExit function" on page 244, and "ULPalmLaunch function" on page 245. For more information on the C++ API methods, see "PalmExit method" on page 147, and "PalmLaunch method" on page 148.

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

Commit all changes before synchronizing

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

For more information on download-only synchronizations, see "download_only synchronization parameter" on page 383.

Adding initial data to your application

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

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

For more synchronization development tips, see "Development tips" on page 85 of the book *MobiLink Synchronization User's Guide*.

Monitoring and canceling synchronization

This section describes how to monitor and cancel synchronization from C/C++ applications. For information on carrying out these tasks in Java, see "Monitoring and canceling synchronization" on page 356.

To monitor and cancel synchronization, you specify a synchronization observer callback function in the **ul_synch_info** structure. This structure is passed to the synchronization function (embedded SQL) or method (C++ API). The observer function is then called at various points during the synchronization, and supplied with information about the synchronization state.

Monitoring synchronization

To monitor synchronization from an UltraLite C/C++ application, you supply the name of a callback function in the **observer** member of your synchronization structure.

The overall process for monitoring synchronization is as follows:

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

Example

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

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

Writing a synchronization callback function

The callback function that you use to monitor synchronization takes a **ul_synch_status** structure as parameter. The **ul_synch_status** structure has the following members:

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

For a description of the synchronization process, see "The synchronization process" on page 24 of the book *MobiLink Synchronization User's Guide*.

- **tableCount** The total number of tables in the database. This number may be more than the number of tables being synchronized.
- **tableIndex** The current table which is being uploaded or downloaded. This number may skip values when not all tables are being synchronized.
- **info** A pointer to the **ul_synch_info** structure.
- **received.inserts** The number of inserted rows that have been downloaded so far.
- **received.updates** The number of updated rows that have been downloaded so far.
- **received.deletes** The number of deleted rows that have been downloaded so far.
- **received.bytes** The number of bytes that have been downloaded so far.
- **sent.inserts** The number of inserted rows that have been uploaded so far.
- **sent.updates** The number of updated rows that have been uploaded so far.
- **sent.deletes** The number of deleted rows that have been uploaded so far.
- **sent.bytes** The number of bytes that have been uploaded so far.
- **stop** Set this member to true to interrupt the synchronization

Example

The following code illustrates how a very simple observer function could be implemented:

```
extern void __stdcall ObserverFunc(
    p_ul_synch_status status )
{
    printf( "UL_SYNCH_STATE is %d: ",
            status->state );

    switch( status->state ) {
        case UL_SYNCH_STATE_STARTING:
        printf( "Starting\n");
        break;

    case UL_SYNCH_STATE_CONNECTING:
        printf( "Connecting\n"  );
        break;

    case UL_SYNCH_STATE_SENDING_HEADER:
        printf( "Sending Header\n" );
        break;
```

```
        case UL_SYNCH_STATE_SENDING_TABLE:
            printf( "Sending Table %d of %d\n",
                    status->tableIndex + 1,
                    status->tableCount );
            break;
    ...
```

This function produces the following output when synchronizing two tables:

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

CustDB example

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

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

Configuring development tools for 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,.

This section describes in general terms a strategy for incorporating UltraLite application development into a dependency-based build environment. The UltraLite plug-in for Metrowerks CodeWarrior automatically provides Palm Computing platform developers with the features described here. For other development tools, you must make the appropriate changes yourself.

For information on the UltraLite plugin for CodeWarrior, see "Developing UltraLite applications with Metrowerks CodeWarrior" on page 255.

For specific instructions on adding embedded SQL projects to a dependency-based development environment, see "Configuring development tools for embedded SQL development" on page 198.

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

1. Add a dummy file to your development project.

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

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

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

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

   ```
   "%asany8%\win32\ulgen.exe" –q –c "connection-string"
   $(InputName) $(InputName).c
   ```

where **asany8** is an environment variable that points to your SQL Anywhere installation directory, *connection-string* is a connection to your reference database, and *InputName* is the UltraLite project name, and should match the root of the text file name. The output is *$(InputName).c*.

3 Compile the dummy file to generate the UltraLite database code.

4 Add the generated UltraLite database file to your development project.

5 Add the UltraLite import libraries for your target platform to your include path.

The import libraries are held in platform-specific directories under the *SQL Anywhere 8\UltraLite* directory.

6 When you alter any SQL statements in the reference database, touch the dummy file, to update its timestamp and force the UltraLite generator to be run.

Deploying UltraLite applications

Once built, your application is contained in a single executable file (C/C++) or set of classes (Java).

Your UltraLite application automatically initializes its own database the first time it is invoked. At first, your database contains no data. You can add data explicitly using INSERT statements in your application, or you can import data from a consolidated database through synchronization. Explicit INSERT statements are especially useful when developing prototypes. You do not need to deploy a separate UltraLite database with your application.

When deploying a new version of an application, the default behavior is for UltraLite to create a new database, losing any data in the database before the new application was deployed. If you call **ULEnableGenericSchema** at the beginning of your application, the database is instead upgraded to the schema of the new application.

For more information, see "ULEnableGenericSchema function" on page 236.

If you linked a C/C++ UltraLite application using the UltraLite library, the custom database engine is an integral component of this executable. To deploy your application, copy the executable file to your target device.

Using the UltraLite runtime DLL

Some platforms, such as Windows CE, support dynamic link libraries. If your target is one of these platforms, you have the option to use the UltraLite runtime DLL.

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

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

For more information on specific platforms, see the following:

- "Deploying Palm applications" on page 291.
- "Deploying Windows CE applications" on page 299.
- "Developing Applications for VxWorks" on page 309
- "Deploying Java applications" on page 364

PART TWO

Developing UltraLite Applications in C/C++

This part describes the development of applications written in C or C++. It explains how to write and build applications using embedded SQL and using the UltraLite C++ API. It also provides tutorials to guide you through the development process.

CHAPTER 5

Tutorial: Build an Application Using the C++ API

About this chapter

This chapter provides a tutorial that guides you through the process of developing a UltraLite application using the C++ API. It describes how to build a very simple application, and how to add synchronization to your application.

Contents

Introduction to the UltraLite C++ API

You can use the UltraLite C++ API to develop UltraLite C/C++ programs using an API instead of embedded SQL. It provides an equivalent functionality to embedded SQL, but in the form of a C++ interface.

Base classes

The UltraLite C++ API starts with a set of base classes that represent the basic components of an UltraLite application. These are:

- **ULData** Represents an UltraLite database.
- **ULConnection** Represents a connection to an UltraLite database, and also handles synchronization.
- **ULCursor** Provides methods used by generated table or result set objects, for accessing and modifying the data.
- **ULTable** Provides methods used by generated table objects, but not by generated result set objects. This class inherits from **ULCursor**.
- **ULResultSet** Provides methods used by generated result set objects, but not by generated table objects. This class inherits from **ULCursor**, and is not documented separately as it contains only methods that are in **ULCursor**.
- **ULStatement** Represents a statement that does not return a result set, such as an INSERT or UPDATE statement. All methods of this class are generated.

Generated classes

For each application, the UltraLite generator writes out a set of classes that describe your particular UltraLite database.

- **Generated result set classes** Individual SQL statements that return result sets are represented by a class, with methods for traversing the result set, and for modifying the underlying data.
- **Generated table classes** Each table in the application is represented by a class, and methods on that table allow the rows of the table to be modified.

  For example, for a table named *Employee*, the UltraLite generator generates a class also named **Employee**.
- **Generated statement classes** Individual SQL statements that do not return result sets are represented by a simple class with an **Execute** method.

You use these classes in your application to access and modify data, and to synchronize with consolidated databases.

Overview

This tutorial describes how to construct a very simple application using the UltraLite C++ API. The application is a Windows console application, developed using Microsoft Visual C++, which queries data in the *ULProduct* table of the UltraLite 8.0 Sample database.

The tutorial takes you through configuration of Visual C++, in such a way that users of other development platforms should be able to identify the steps required. These steps are supplied so that you can start development of your own applications.

In the tutorial, you write and build an application that carries out the following tasks.

1. Connects to an UltraLite database, consisting of a single table. The table is a subset of the *ULProduct* table of the UltraLite Sample database.
2. Inserts rows into the table. Initial data is usually added to an UltraLite application by synchronizing with a consolidated database. Synchronization is added later in the chapter.
3. Writes the first row of the table to standard output.

In order to build the application, you must carry out the following steps:

1. Design the UltraLite database in an Adaptive Server Anywhere reference database.

   Here we use a single table from the UltraLite sample database (CustDB).
2. Run the UltraLite generator to generate the API for this UltraLite database.

   The generator writes out a C++ file and a header file.
3. Write source code that implements the logic of the application.

   Here, the source code is just *main.cpp*.
4. Compile, link, and run the application.

You then add synchronization to your application.

Lesson 1: Getting started

In this tutorial, you will be creating a set of files, including source files and executable files. You should make a directory to hold these files. In addition, you should make a copy of the UltraLite sample database so that you can work on it, and be sure you still have the original sample database for other projects.

Copies of the files used in this tutorial can be found in the *Samples\UltraLite\APITutorial* subdirectory of your SQL Anywhere directory.

❖ **To prepare a tutorial directory:**

- Create a directory to hold the files you will create. In the remainder of the tutorial, we assume that this directory is *c:\APITutorial*.

❖ **To copy the sample database:**

- Make a backup copy of the UltraLite 8.0 Sample database into the tutorial directory. The UltraLite 8.0 Sample database is the file *custdb.db*, in the *Samples\UltraLite\CustDB* subdirectory of your SQL Anywhere installation directory. In this tutorial, we use the original UltraLite 8.0 Sample database, and at the end of the tutorial you can copy the untouched version from the *APITutorial* directory back into place.

Lesson 2: Create an UltraLite database template

In this tutorial, you use the original copy of the UltraLite 8.0 Sample database (CustDB) as a reference database. The copy you placed in the *APITutorial* directory serves as a backup copy.

An **UltraLite database template** is a set of tables, and columns within tables, that are to be included in your UltraLite database. You create an UltraLite database template by creating a SQL Remote publication in the reference database. The SQL Remote publication is simply a convenient device for assembling tables and column-based subsets of tables: there is no direct connection to SQL Remote.

You can also define your UltraLite database by adding SQL statements to the reference database. SQL statements allow you to include joins and more advanced features in your UltraLite application. Here, we build an UltraLite database template by defining tables, as it is more simple.

The Java tutorial uses SQL statements to define the UltraLite database. For an example of how to add SQL statements to a database, see "Lesson 1: Add SQL statements to your reference database" on page 326.

❖ To create the UltraLite database template:

1. Start Sybase Central.
2. Connect to the UltraLite 8.0 Sample database.
   - Choose Tools➤Connect.
   - If a list of plugins is displayed, choose Adaptive Server Anywhere.
   - In the Connect dialog, choose the UltraLite 8.0 Sample ODBC data source.
   - Click OK to connect.
3. Create a SQL Remote publication that reflects the data you wish to include in your UltraLite database.
   - In Sybase Central, open the custdb database.
   - Open the SQL Remote folder. Open the Publications folder, and click Add Publication. The Publication Creation wizard appears.
   - Add the *ULProduct* table to the publication, including all columns in the publication.
   - Click Finish to create the publication.

You have now finished designing the UltraLite database template. Leave Sybase Central and the database server running for the next lesson.

Lesson 3: Run the UltraLite generator

The UltraLite generator writes a C++ file and a header file that define an interface to the UltraLite database, as specified in the UltraLite database template.

❖ **To generate the UltraLite interface code:**

1. From a command prompt, change directory to your *APITutorial* directory.
2. Run the UltraLite generator with the following arguments (all on one line):

```
ulgen -c "dsn=UltraLite 8.0 Sample" –t c++
-u ProductPub -f ProductPubAPI
```

The generator writes out the following files:

- **ProductPubAPI.hpp** This file contains prototypes for the generated API. *You should inspect this file to determine the API you can use in your application.*
- **ProductPubAPI.cpp** This file contains the interface source. You do not need to look at this file.
- **ProductPubAPI.h** This file contains internal definitions required by UltraLite. You do not need to look at this file.

Lesson 4: Write the application source code

Copy and paste the following source code into a file named *sample.cpp* in your *tutorial* directory. You can find this source code in *Samples\UltraLite\APITutorial\sample.cpp*, although you may have to edit the file to uncomment the inserts.

The code does not contain error checking or other features that you would require in a complete application. It is provided as a simplified application, for illustrative purposes only.

```
// (1) include headers
#include <stdio.h>
#include "ProductPubAPI.hpp"

void main() {
   // (2) declare variables
   long price;
   ULData db;
   ULConnection conn;
   ULProduct productTable;

   // (3) connect to the UltraLite database
   db.Open() ;
      conn.Open( &db, "dba", "sql" );
   productTable.Open( &conn );

   // (4) insert sample data
   productTable.SetProd_id( 1 );
   productTable.SetPrice( 400 );
   productTable.SetProd_name( "4x8 Drywall x100" );
   productTable.Insert();

   productTable.SetProd_id( 2 );
   productTable.SetPrice( 3000 );
   productTable.SetProd_name( "8’ 2x4 Studs x1000" );
   productTable.Insert();

   // (5) Write out the price of the items
   productTable.BeforeFirst();
   while( productTable.Next() ) {
      productTable.GetPrice( price );
      printf("Price: %d\n", price );
   }

   // (6) close the objects to finish
   productTable.Close();
   conn.Close();
   db.Close();
}
```

Explanation

The numbered comments in the code indicate the main tasks this routine carries out:

1 Include headers.

In addition to *stdio.h*, you need to include the generated header file *ProductPubAPI.hpp* to include the generated classes describing the Product table. This file in turn includes the UltraLite header file *ulapi.h*.

2 Declare variables.

The UltraLite database is declared as an instance of class **ULData**, and the connection to the database is an instance of class **ULConnection**. These classes are included from *ulapi.h*.

The table is declared as an instance of class **ULProduct**, a generated name derived from the name of the table in the reference database.

3 Connect to the database.

Opening each of the declared objects establishes access to the data. Opening the database requires no arguments; opening a connection requires a user ID and password, and also the name of the database. Opening the table requires the name of the connection.

4 Insert sample data.

In a production application, data is entered into the database by synchronizing. It is a useful practice to insert some sample data during the initial stages of development, and include synchronization at a later stage.

The method names in the **ULProduct** class are unique names that reflect the columns of the table in the reference database.

Synchronization is added to this routine in "Lesson 6: Add synchronization to your application" on page 118.

5 Write out the price of each item.

The price is retrieved and written out for each row in the table.

6 Close the objects.

Closing the objects used in the program frees the memory associated with them.

Lesson 5: Build and run your application

You can compile and link your application in the development tool of your choice. In this section, we describe how to compile and link using Visual C++; if you are using one of the other supported development tools, modify the instructions to fit your tool.

1 Start Microsoft Visual C++ from your desktop in the standard fashion.

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

Select Tools➤Options and click on the Directories tab. In the Show Directories For dropdown list, choose Include Files. Include the following directory, so that the header files can be accessed.

```
C:\Program Files\Sybase\SQL Anywhere 8\h
```

On the same tab, select Library Files under the Show Directories For dropdown menu. Include the following directory so that the UltraLite library files can be accessed.

```
C:\Program Files\Sybase\SQL Anywhere
8\ultralite\win32\386\lib
```

Click OK to submit the changes.

3 Create a project named **APITutorial** (it should be the same name as the directory you have used to hold your files).

- Select File➤New. The New dialog is displayed.
- On the Projects tab choose **Win32 Console Application**.
- Specify a project name of **APITutorial.**
- Specify the *APITutorial* directory as location.
- Select New Workspace and click OK.
- Choose to create An Empty Project and click Finish.
- On the Workspace window, click the FileView tab. The workspace tutorial consists of just the APITutorial project. Double-click APITutorial files to display the three folders: Source Files, Header Files, Resource Files.

4 Configure the project settings.

- Right click on APITutorial files and select Settings. The Project Settings dialog is displayed.
- From the Settings For dropdown menu, choose All Configurations.

- Click the Link tab. Add the following runtime library to the Object/Library Modules box.

  ```
  ulimp.lib
  ```

- Click on the C/C++ tab. From the Category dropdown menu, choose General. Add the following to the Preprocessor definitions list:

  ```
  __NT__,UL_USE_DLL
  ```

  Here, __NT__ has two underscores either side of NT.

- Click OK to finish.

5 Add *sample.cpp* and *ProductPubAPI.cpp* to the project.

- Right click on the Source Files folder and select Add Files to Folder. Locate your *sample.cpp* file and click OK. Open the Source Files folder to verify that it contains *sample.cpp*.
- Repeat to add the generated *ProductPubAPI.cpp* file to the project.

6 Add the file containing the base classes for the UltraLite API to the project.

- Right click on the Source Files folder and choose Add Files to Folder. Browse to *ulapi.cpp*, which is in the *src* subdirectory of your SQL Anywhere directory, and click OK.

7 Compile and link the application.

- Select Build➤Build APITutorial.exe to compile and link the executable. Depending on your settings, the *APITutorial.exe* executable may be created in a Debug directory within your *APITutorial* directory.

8 Ensure that the application can locate the UltraLite runtime library.

- The UltraLite runtime library is *ulrt8.dll*. In Visual C++, choose Tools➤Options, click the Directories tab. From the Show Directories For list choose **Executable files**. Add the *win32* subdirectory of your SQL Anywhere directory to the list. Click OK to complete.

9 Run the application.

- Select Build➤Execute APITutorial.exe.

A command prompt window appears and displays the prices of the products in the product table.

You have now built and run a simple UltraLite application. The next step is to add synchronization to your application.

Lesson 6: Add synchronization to your application

UltraLite applications exchange data with a consolidated database. In this lesson, you add synchronization to the simple application you created in the previous section. In addition, you change the output to verify that synchronization has taken place.

Adding synchronization actually simplifies the code. Your initial version of *main.cpp* has the following lines, that inserts some data into your UltraLite database.

```
productTable.SetProd_id( 1 );
productTable.SetPrice( 400 );
productTable.SetProd_name( "4x8 Drywall x100" );
productTable.Insert();

productTable.SetProd_id( 2 );
productTable.SetPrice( 3000 );
productTable.SetProd_name( "8’ 2x4 Studs x1000" );
productTable.Insert();
```

This code is included to provide an initial set of data for your application. In a production application, you would usually not insert an initial copy of your data from source code, but would carry out a synchronization.

❖ **To add synchronization to your application:**

1 Add a synchronization information structure to your code.

- Add the following line immediately after the line that says `// (2) declare variables`.

  ```
  auto ul_synch_info synch_info;
  ```

  This structure holds the parameters that control the synchronization.

2 Replace the explicit inserts with a synchronization call.

- Delete the **productTable** methods listed above.
- Add the following lines in their place:

  ```
  conn.InitSynchInfo( &synch_info );
  synch_info.user_name = UL_TEXT( "50" );
  synch_info.version = UL_TEXT( "custdb" );
  synch_info.stream = ULSocketStream();
  synch_info.stream_parms =
     UL_TEXT("host=localhost");
  conn.Synchronize( &synch_info );
  ```

The value of `50` is the MobiLink user name.

The string `custdb` instructs MobiLink to use the default script version for synchronization.

`ULSocketStream()` instructs the application to synchronize over TCP/IP, and `host=localhost` specifies the host name of the MobiLink server, which in this case is the current machine.

3 Compile and link your application.

- Select Build➤Build APITutorial.exe to compile and link the executable. Depending on your settings, the *APITutorial.exe* executable may be created in a Debug directory within your *APITutorial* directory.

4 Start the MobiLink server running against the sample database.

From a command prompt in your *APITutorial* directory, enter the following command:

```
start dbmlsrv8 -c "dsn=UltraLite 8.0 Sample"
```

5 Run your application.

From the Build menu, choose Execute APITutorial.exe.

The application connects, synchronizes to receive data, and writes out information to the command prompt window. The output is as follows:

```
The ULData object is open
Price: 400
Price: 3000
Price: 40
Price: 75
Price: 100
Price: 400
Price: 3000
Price: 75
Price: 40
Price: 100
```

In this lesson, you have added synchronization to a simple UltraLite application.

Restore the sample database

Now that you have completed the tutorial, you should restore the sample database so that it can be used again. You created a copy of the UltraLite 8.0 Sample database in "Lesson 1: Getting started" on page 110. You can now replace the version of *custdb.db* that you just changed with the copy.

❖ **To restore the sample database:**

1. Copy the *custdb.db* file from your tutorial directory to the *UltraLite\Samples\CustDB* subdirectory of your SQL Anywhere directory.
2. In the same directory, delete the transaction log file *custdb.log*.

Your sample database is now restored to its original state.

CHAPTER 6

Developing C++ API Applications

About this chapter

This chapter describes how to develop applications using the UltraLite C++ API. This interface represents predefined queries or tables in your UltraLite database as objects, and provides methods that enable you to manipulate them from your application without using SQL.

Contents

Introduction

This chapter provides notes for developers who are writing and building UltraLite applications using the C++ API.

What's in this chapter?

The chapter includes the following information:

- Information about how to define the data access features to be used in your application.

  See "Defining features for your application" on page 123.

- Information on generating C++ API classes from your reference database.

  See "Generating UltraLite C++ classes" on page 127.

- Notes on the C++ API classes that are generated.

  See "Working with the C++ API classes" on page 125.

- Notes on compiling and linking UltraLite C++ API applications.

  See "Compiling and linking your application" on page 128.

Before you begin

The development process for the C++ API is similar to that for other UltraLite development models. This chapter assumes a familiarity with that process.

For more information, see "Developing UltraLite Applications" on page 67.

Defining features for your application

The SQL statements to be included in the UltraLite application, and the structure of the UltraLite database itself, are defined by adding the SQL statements to the reference database for your application.

Defining projects

When you run the UltraLite generator, it writes out class definitions for all the SQL statements in a given **project**. A project is a name defined in the reference database, which groups the SQL statements for an application. You can store SQL statements for multiple applications in a single reference database by defining multiple projects.

For information on creating projects, see "Creating an UltraLite project" on page 80.

You can use the **ul_delete_project** stored procedure to remove a project definition.

Adding statements to a project

For information on adding SQL statements to an UltraLite project, see "Adding SQL statements to an UltraLite project" on page 81.

For information on using placeholders, and other aspects of writing SQL statements for UltraLite, see "Writing UltraLite SQL statements" on page 83.

Defining UltraLite tables

If you do not intend to carry out joins, and if you have strong constraints on your application executable size, you can define tables instead of queries for your UltraLite application.

You define a subset of a database for use in a C++ API application by creating a publication in the reference database. A publication defines the set of tables, and columns in those tables, that you want to include in your UltraLite application. The use of a SQL Remote publication is purely a convenience for UltraLite, and does not imply any connection with SQL Remote or MobiLink software.

SQL Remote publications allow you to qualify which rows any SQL Remote user receives using subqueries and parameters. You cannot use these devices when creating publications for use with UltraLite: only the set of tables and columns within those tables is used for defining the UltraLite classes.

Tables or queries?

Table definitions and query definitions provide alternative ways of defining the data that is to be included in your UltraLite database, and the range of operations you can carry out on that data.

Using SQL statements and projects provides a more general approach to defining applications, and are most likely to be used in larger enterprise applications. Table definitions may be useful as a convenient device in the following cases:

- Your application needs to access data only one table at a time. You cannot define joins using table definitions.
- You are severely constrained for memory use. The code generated for table definitions is smaller than that for queries, because of their simpler structure.

Defining database features for C++ API applications

C++ API applications use some functions that are not part of the class hierarchy. These functions control aspects of the database storage and access. They are as follows:

- "ULEnableFileDB function" on page 235.
- "ULEnablePalmRecordDB function" on page 237.
- "ULEnableStrongEncryption function" on page 238.
- "ULEnableUserAuthentication function" on page 238.

Other aspects of database storage are configured using the UL_STORE_PARMS macro. For more information, see "UL_STORE_PARMS macro" on page 428.

Working with the C++ API classes

This section contains notes about the classes that make up the C++ API.

Working with the ULData and ULConnection objects

The **ULData** object makes the data in the database object available to your application. You need to call **ULData::Open()** before you can connect to the UltraLite database or carry out any operations on the data.

The **ULData::Open()** method can be called with parameters that define the storage and access parameters for the database (file name, cache size, reserved size).

Once the **ULData** object is opened, you can open a connection on the database. You do that using the **ULConnection::Open()** method, supplying a reference to the **ULData** object and a set of connection parameters to establish the connection. You can use multiple connections on a single database. Once the connection is established, you can open the generated **ULStatement**, **ULResultSet** or **ULTable** objects that define the tables or statements used in your application, and use these objects to manipulate the data.

The **ULConnection** object defines the general characteristics of how you interact with the data.

Synchronization is carried out using the **ULConnection** object. The **Synchronize** method carries out synchronization of the data with a MobiLink server.

Palm Computing Platform developers

If you are developing an application for the Palm Computing Platform, there are some extra considerations for how to use these objects. In particular, the **PalmLaunch** and **PalmExit** methods are called when launching and leaving the application. The **ULData::Close()** method is not called on the Palm Computing Platform.

For more information, see "Developing Applications for the Palm Computing Platform" on page 253.

Using table and query classes

Each table or query is represented by a class. The API for accessing and modifying the rows in the table or query is based on a SQL **cursor**: a pointer to a position in the table or query.

The cursor can have the following positions:

- **Before the first row** This position has value 0. This is the position of the cursor when the table or query is opened.
- **On a row** If a table or query has *n* rows, positions 1 to *n* for the cursor correspond to the rows.
- **After the last row** This position has value (*n*+ 1)

You can move through the rows of the object using methods of the object, including **Next()** and **Previous()**.

Palm Computing Platform developers

If you are developing an application for the Palm Computing Platform, there are some extra considerations for how to use these objects.

For more information, see "Launching and closing UltraLite applications" on page 261.

Row ordering

The order of the rows in the object is determined when it is opened. By default, tables are ordered by primary key. The UltraLite generator adds an enumeration for the object definition, with a member for each index on the table in the reference database (the primary key is named **Primary**), and by specifying a member of this enumeration, you can control the ordering of the rows in the object.

If you update a row so that it no longer belongs in the current position the current row of the cursor moves to that row.

For example, consider a single-column object with the values A, B, C, and E.

- If a cursor is sitting on row B (position 2) and modifies the value to D, then the row is moved to sit between C and E (becoming position 3) and the current row of the cursor changes to position 3.

If you insert a row, the current position does not move to that row.

Building your UltraLite C++ application

This section covers the following subjects:

- "Generating UltraLite C++ classes" on page 127.
- "Compiling and linking your application" on page 128.

Some small sample applications are provided that include makefiles for compilation. These applications can be found in subdirectories of the *Samples\UltraLite* directory.

Generating UltraLite C++ classes

The generator generates table classes from publications in the database, and query classes from any SQL statements added with the *ul_add_statement* stored procedure, writing the output to the following files:

- *filename.hpp* This file contains the prototypes for the generated interface. You should inspect this file to determine the API you can use in your application.
- *filename.cpp* This file contains the interface source. You do not need to look at this file.
- *filename.h* This file contains internal definitions required by UltraLite. You do not need to look at this file.

Here, *filename* is the name supplied on the **ulgen** command line.

Whether you use queries in a project, publications, or a mix to define the classes in your application, you must generate all the code in a single run of the UltraLite generator.

❖ **To generate UltraLite code for a publication:**

- Run the UltraLite generator specifying the publication name with the `-u` command-line switch. For example:

```
ulgen -c "uid=dba;pwd=sql" –t c++ -u pubName -f
filename
```

❖ **To generate UltraLite code for a UltraLite project:**

- Run the UltraLite generator, specifying the project name with the `–j` command-line switch. For example:

```
ulgen -c "uid=dba;pwd=sql" –t c++ -j projectname -f
filename
```

❖ **To generate UltraLite code for both a project and a publication:**

♦ Run the UltraLite generator, specifying the project name and the publication name. For example:

```
ulgen -c "uid=dba;pwd=sql" –t c++ -j projectname –u
pubname -f filename
```

☞ For more information on the UltraLite generator, see "The UltraLite generator" on page 419.

Compiling and linking your application

When you compile your UltraLite application, you must ensure that the compiler can locate all the required files.

♦ **Generated source files** You must included the generated files describing the API in your project. This includes the generated *.cpp* file, *.h* file, and *.hpp* file.

♦ **UltraLite header files** You must configure your compiler so that it can locate the UltraLite header files.

These header files are installed into the *h* directory under your Adaptive Server Anywhere installation directory.

♦ **UltraLite c file** You must configure your linker so that it can locate the UltraLite API file *ulapi.cpp*.

This file is installed into the *src* subdirectory of your Adaptive Server Anywhere installation directory.

♦ **Library or import library** You must configure your compiler so that it can locate the UltraLite runtime library for your target platform or, in the case that you are using the UltraLite runtime DLL, the UltraLite imports library.

These files are installed under the *UltraLite* subdirectory of your Adaptive Server Anywhere installation directory. Each target platform has a separated directory, and if there are different processors for a platform, each has its own subdirectory.

☞ For a sample application that includes compilation options, see the files in *Samples\UltraLite\apitutorial*.

CHAPTER 7

C++ API Reference

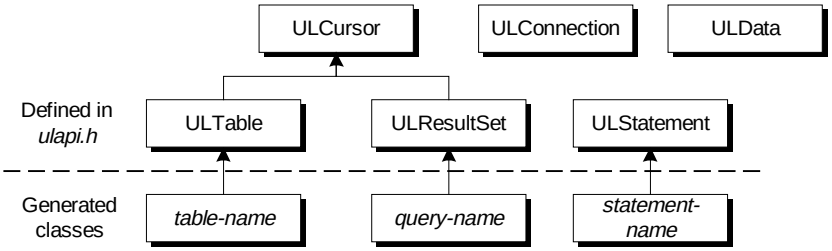
About this chapter

This chapter describes the UltraLite C++ API.

Contents

C++ API class hierarchy

The classes in the C++ API are displayed in the following diagram:



The classes are described in the following header files:

- **generated-name.hpp** The interface generated for a particular set of statements or tables is defined in the generated *.hpp* file.
- **ulapi.h** The base classes are defined in *ulapi.h*, in the *h* subdirectory of your SQL Anywhere installation directory.
- **ulglobal.h** You may want to look at *ulglobal.h*, in the *h* subdirectory of your SQL Anywhere installation directory, for some of the data types and other definitions used in *ulapi.h*.

Functions available from the C++ API

C++ API applications use some functions that are not part of the class hierarchy. These functions are as follows:

- "ULEnableFileDB function" on page 235.
- "ULEnableGenericSchema function" on page 236.
- "ULEnablePalmRecordDB function" on page 237.
- "ULEnableStrongEncryption function" on page 238.
- "ULEnableUserAuthentication function" on page 238.

C++ API language elements

The UltraLite API methods and variables are described in terms of a set of UltraLite data types. These data types are described in this section.

UltraLite data types

- **an_SQL_code** A data type for holding SQL error codes.
- **ul_char** A data type representing a character. If the operating system uses Unicode, **ul_char** uses two bytes per character. For single-byte character sets, **ul_char** uses a single byte per character.
- **ul_binary** A data type representing one byte of binary information.
- **ul_column_num** A data type for holding a number indicating a column of a table or query. The first column in the table or query is number one.
- **ul_fetch_offset** A data type for holding a relative number in a ULCursor object.
- **ul_length** A data type for holding the length of a data type.
- **DECL_DATETIME** A type for holding date and time information in a SQLDATETIME structure, which is defined as follows:

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

  DECL_DATETIME is also used in embedded SQL programming. Other embedded SQL data types with named DECL_*type* are not needed in C++ API programming.

- **UL_NULL** A constant representing a SQL NULL.

ULConnection class

Object Represents a database connection.

Description A ULConnection object represents an UltraLite database connection. It provides methods to open and close a connection, to check whether a connection is open, to synchronize a database on the current connection, and more.

For embedded SQL users, opening a **ULConnection** object is equivalent to the EXEC SQL CONNECT statement.

Close method

Prototype bool **Close ( )**

Description Disconnects your application from the database, and frees resources associated with the ULConnection object. Once you have closed the ULConnection object, your application is no longer connected to the UltraLite database.

Closing a connection rolls back any outstanding changes.

You should not close a connection object in a Palm Computing Platform application. Instead, use the **Reopen** method when the application is reactivated. For more information, see "Reopen method" on page 140.

Returns **true** (1) if successful.

false (0) if unsuccessful.

Example The following example closes a ULConnection object:

```
conn.Close();
```

See also "Open method" on page 139

Commit method

Prototype bool **Commit( )**

Description Commits outstanding changes to the database.

Returns **true** (1) if successful.

false (0) if unsuccessful.

Example The following code inserts a value to the database, and commits the change.

```
productTable.Open( &conn );
productTable.SetProd_id( 2 );
productTable.SetPrice( 3000 );
productTable.SetProd_name( "8’ 2x4 Studs x1000" );
productTable.Insert();
conn.Commit();
```

See also "Rollback method" on page 141

CountUploadRows method

Prototype

ul_u_long **CountUploadRows(**
ul_publication_mask *mask*,
ul_u_long *threshold***)**

Description

Returns the number of rows that need to be uploaded when the next synchronization takes place.

You can use this function to determine if a synchronization is needed.

Parameters

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

```
UL_PUB_PUB1 | UL_PUB_PUB2
```

For more information on publication masks, see "publication synchronization parameter" on page 386.

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

Returns

The number of rows to be uploaded.

GetCA method

Prototype

SQLCA ***GetCA( )**

Description

Retrieves the SQLCA associated with the current connection.

This function is useful if you are combining embedded SQL and the C++ API in a single application.

Returns

A pointer to the SQLCA.

Example

```
ULConnection conn;
conn.Open();
conn.GetCA();
```

See also "The SQL Communication Area (SQLCA)" on page 188 of the book *ASA Programming Guide*

GetLastIdentity method

Prototype ul_u_big **GetLastIdentity( )**

Description Returns the most recent identity value used. This function is equivalent to the following SQL statement:

```
SELECT @@identity
```

The function is particularly useful in the context of global autoincrement columns.

Returns The last identity value.

See also "Determining the most recently assigned value" on page 61
"Global autoincrement default column values" on page 58

GetLastDownloadTime method

Prototype bool **GetLastDownloadTime(**
ul_publication_mask *mask*,
DECL_DATETIME **value* **)**

Description Provides the last time a specified publication was downloaded.

Parameters

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

```
UL_PUB_PUB1 | UL_PUB_PUB2
```

For more information on publication masks, see "publication synchronization parameter" on page 386.

value A pointer to the DECL_DATETIME structure to be populated.

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

Returns

- **true** Indicates that *value* is successfully populated by the last download time of the publication specified by *publication-mask*.

- **false** Indicates that *publication-mask* specifies more than one publication or that the publication is undefined. If the return value is false, the contents of *value* are not meaningful.

GetSQLCode method

Prototype an_SQL_code **GetSQLCode( )**

Description Provides error checking capabilities by checking the SQLCODE value for the success or failure of a database operation. The SQLCODE is the standard Adaptive Server Anywhere code.

SQLCODE is reset by any subsequent UltraLite database operation, including those on other connections.

Returns The SQLCODE value as an integer.

Example The following code writes out a SQLCODE. If the synchronization call fails, a value of -85 is returned.

```
conn.Synchronize( &synch_info );
sqlcode = conn.GetSQLCode();
printf("sqlcode: %d\n", sqlcode );
```

See also "Database Error Messages" on page 1 of the book *ASA Errors Manual*

GetSynchResult method

Prototype bool **GetSynchResult(** ul_synch_result * *synch-result* **);**

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

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

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

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

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

where the individual members have the following meanings:

- **sql_code** The SQL code from the last synchronization. For a list of SQL codes, see "Error messages indexed by Adaptive Server Anywhere SQLCODE" on page 2 of the book *ASA Errors Manual*.
- **stream_error** The communication stream error code from the last synchronization. For a listing, see "MobiLink Communication Error Messages" on page 631 of the book *MobiLink Synchronization User's Guide*.
- **upload_ok** Set to **true** if the upload was successful; **false** otherwise.
- **ignored_rows** Set to **true** if uploaded rows were ignored; **false** otherwise.
- **auth_status** The synchronization authentication status. For more information, see "auth_status synchronization parameter" on page 381.
- **auth_value** The value used by the MobiLink synchronization server to determine the **auth_status** result. For more information, see "auth_value synchronization parameter" on page 382.
- **timestamp** The time and date of the last synchronization.
- **status** The status information used by the observer function. For more information, see "observer synchronization parameter" on page 384.

Returns The method returns a boolean value.

true Success.

false Failure.

See also "PalmLaunch method" on page 148

GlobalAutoincUsage method

Prototype ul_u_short **GlobalAutoincUsage()**

Description Returns the percentage of available global autoincrement values that have been used.

If the percentage approaches 100, your application should set a new value for the global database ID, using the SetDatabaseID.

Returns The percent usage of the available global autoincrement values.

See also "Global autoincrement default column values" on page 58
"SetDatabaseID method" on page 142

GrantConnectTo method

Prototype bool **GrantConnectTo(** *userid*, *password* **)**

Parameters

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

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

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

See also "User authentication for UltraLite databases" on page 442
"Adding user authentication to your application" on page 85
"RevokeConnectFrom method" on page 141

InitSynchInfo method

Prototype an_SQL_code **InitSynchInfo(** ul_synch_info * *synch_info* **)**

Description Initializes the *synch_info* structure used for synchronization.

Returns None

Example The following code illustrates where the **InitSynchInfo** method is used in the sequence of calls that synchronize data in a UltraLite application.

```
auto ul_synch_info synch_info;
conn.InitSynchInfo( &synch_info );
conn.Synchronize( &synch_info );
```

See also "Synchronize method" on page 143

IsOpen method

Prototype bool **IsOpen ( )**

Description Checks whether the ULConnection object is currently open.

Returns **true** (1) if the ULConnection object is open.

false (0) if the ULConnection object is not open.

Example The following example checks that an attempt to Open a connection succeeded:

```
ULConnection conn;
conn.Open();
if( conn.IsOpen() ){
   printf( "Connected to the database.\n" );
}
```

See also "Open method" on page 139

LastCodeOK method

Prototype bool **LastCodeOK ( )**

Description Checks the most recent SQLCODE and returns **true** if the code represents a warning or success. The function returns **false** if the most recent SQLCODE represents an error.

This method provides a convenient way of checking for the success or potential failure of operations. You can use GetSQLCode to obtain the numerical value.

SQLCODE is reset by any subsequent UltraLite database operation, including those on other connections.

Returns **true** (1) if the previous SQLCode was zero or a warning.

false (0) if the previous SQLCode was an error.

Example The following example checks that an attempt to Open a connection succeeded:

```
ULConnection conn;
conn.Open();
if( conn.LastCodeOK() ){
   printf( "Connected to the database.\n" );
};
```

See also "GetSQLCode method" on page 135

LastFetchOK method

Prototype

bool **LastFetchOK( )**

Description

Provides a convenient way of checking that the most recent fetch of a row succeeded (**true**) or failed (**false**).

The value is reset by any subsequent UltraLite database operation, including those on other connections.

Returns

true (1) if successful.

false (0) if unsuccessful.

Example

The following example moves to the last row in a table, fetches a value from the row, and checks for the success of the fetch:

```
tb.Open( &conn );
tb.Last();
tb.GetID( iVal );
if( tb.LastFetchOK() ){
   ... operations on success...
}
```

See also

"AfterLast method" on page 152
"First method" on page 154

Open method

Prototype

bool **Open (** ULData* *db*,
ul_char* *userid*,
ul_char* *password*,
ul_char* *name* = **SQLNULL)**

Description

Open a connection to a database. The **ULData** object must be open for this call to succeed.

Parameters

db A pointer to the **ULData** object on which the connection is made. This argument is usually the address of the **ULData** object opened prior to making the connection.

userid The user ID argument is a placeholder reserved for possible future use. It is ignored.

For more information on user IDs and UltraLite, see "User authentication for UltraLite databases" on page 442.

password The password parameter is a placeholder reserved for possible future use. It is ignored.

name An optional name for the connection. This is needed only if you have multiple connections from a single application to the same database.

Returns

true (1) if successful.

false (0) if unsuccessful.

Example

The following example opens a connection to the UltraLite database.

```
ULData db;
ULConnection conn;

db.Open();
conn.Open( &db, "dummy", "dummy" );
```

See also

"Close method" on page 132

Reopen method

Prototype

bool **Reopen ( )**

bool **Reopen(** ULData **db***,** ul_char * *name* **= SQLNULL)**

Description

This method is available for the Palm Computing Platform only. The **ULData** object must be reopened for this call to succeed.

When developing Palm applications, you should never close the connection object. Instead, you should call **Reopen** when the user switches to the UltraLite application. The method prepares the data in use by the database object for use by the application.

db A pointer to the **ULData** object on which the connection is made. This argument is usually the address of the **ULData** object opened prior to reopening the connection.

name An optional name for the connection. This is needed only if you have multiple connections from a single application to the same database.

Returns

true (1) if successful.

false (0) if unsuccessful.

Example

The following example reopens a database object, and then a connection object:

```
db.Reopen();
conn.Reopen( &db );
```

See also

"Open method" on page 139

ResetLastDownloadTime method

Prototype bool **ResetLastDownloadTime(** ul_publication_mask *publication-mask* **)**

Description This method can be used to repopulate values and return an application to a known clean state. It resets the last download time so that the application resynchronizes previously downloaded data.

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

```
UL_PUB_PUB1 | UL_PUB_PUB2
```

For more information on publication masks, see "publication synchronization parameter" on page 386.

Example The following example resets the download time for all tables in the database:

```
db.Reopen();
conn.ResetLastDownloadTime( UL_SYNC_ALL );
```

See also "GetLastDownloadTime method" on page 134
"Timestamp-based synchronization" on page 86 of the book *MobiLink Synchronization User's Guide*

RevokeConnectFrom method

Prototype bool **RevokeConnectFrom(** ul_char * *userid* **)**

Description Revoke access from an UltraLite database for a user ID.

Parameters **userid** Character array holding the user ID to be excluded from database access. The maximum length is 16 characters.

See also "User authentication for UltraLite databases" on page 442
"Adding user authentication to your application" on page 85
"GrantConnectTo method" on page 137

Rollback method

Prototype bool **Rollback( )**

Description Rolls back outstanding changes to the database.

Returns **true** (1) if successful.

false (0) if unsuccessful.

Example The following code inserts a value to the database, but then rolls back the change.

```
productTable.Open( &conn );
productTable.SetProd_id( 2 );
productTable.SetPrice( 3000 );
productTable.SetProd_name( "8’ 2x4 Studs x1000" );
productTable.Insert();
conn.Rollback();
```

See also "Commit method" on page 132

SetDatabaseID method

Prototype bool **SetDatabaseID(** ul_u_long *value* **)**

Description Sets the database ID value to be used for global autoincrement columns

Parameters **value** The value to use for generating global autoincrement values.

Returns **true** (1) if successful.

false (0) if unsuccessful.

See also
"Global autoincrement default column values" on page 58
"GLOBAL_DATABASE_ID option" on page 569 of the book *ASA Database Administration Guide*
"GlobalAutoincUsage method" on page 136

StartSynchronizationDelete method

Prototype bool **StartSynchronizationDelete( )**

Description Once this function is called, all delete operations are again synchronized.

Returns **true** (1) if successful.

false (0) if unsuccessful.

See also
"START SYNCHRONIZATION DELETE statement [MobiLink]" on page 556 of the book *ASA SQL Reference Manual*
"StopSynchronizationDelete method" on page 142

StopSynchronizationDelete method

Prototype bool **StopSynchronizationDelete( )**

Description Prevents delete operations from being synchronized. This is useful for deleting old information from an UltraLite database to save space, while not deleting this information on the consolidated database.

Returns **true** (1) if successful.

false (0) if unsuccessful.

See also "START SYNCHRONIZATION DELETE statement [MobiLink]" on page 556 of the book *ASA SQL Reference Manual*
"StartSynchronizationDelete method" on page 142

Synchronize method

Prototype bool **Synchronize (** ul_synch_info * *synch_info* **)**

Description Synchronizes an UltraLite database.

For a detailed description of the members of the *synch_info* structure, see "Synchronization parameters" on page 380.

Returns **true** (1) if successful.

false (0) if unsuccessful.

Example The following code fragment illustrates how information is provided to the Synchronize method.

```
auto ul_synch_info synch_info;
conn.InitSynchInfo( &synch_info );
synch_info.user_name = UL_TEXT( "50" );
synch_info.version = UL_TEXT( "custdb" );
synch_info.stream = ULSocketStream();
synch_info.stream_parms =
   UL_TEXT("host=localhost");
conn.Synchronize( &synch_info );
```

See also "Synchronization parameters" on page 380

ULData class

Object

Represents an UltraLite database.

Prototype

```
ULData db;
    db.Open();
```

Description

The ULData class represents an UltraLite database to your application. It provides methods to open and close a database, and to check whether a database is open.

You must open a database before connecting to it or carrying out any other operation, and you must close the database after you have finished all operations on the database, and before your application terminates.

For multi-threaded applications, each thread must create its own **ULData**. Neither the **ULData** object nor the other objects inherited from it (**ULConnection** and other classes) can be shared across threads.

For embedded SQL users, opening a **ULData** object is equivalent to calling **db_init**.

For its position in the API hierarchy, see "C++ API class hierarchy" on page 130.

Example

The following example declares a ULData object and opens it:

```
ULData db;
db.Open();
```

Close method

Prototype

bool **Close ( )**

Description

Frees resources associated with a ULData object, before you terminate your application. Once you have closed the ULData object, you cannot execute any other operations on that database using the C++ API without reopening.

> **Palm Computing Platform**
> Do not call **ULData.Close()** on the Palm Computing Platform. On the Palm Computing Platform, the database must be kept open when you leave the application. Use **ULData.PalmExit** to save the state of the application between sessions instead of calling **ULData.Close**. Use the **Reopen** method when the application is reactivated. For more information, see "Reopen method" on page 150.

Returns

true (1) if successful.

false (0) if unsuccessful.

Example The following example closes a ULData object:

```
db.Close();
```

See also "Open method" on page 146

Drop method

Prototype bool **Drop (** SQLCA * *sqlca***,** ul_char * *store-parms* **)**

Description Delete the UltraLite database file.

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

Do not call this function while a database connection is open. Call this function only after closing the database or before opening the database (C++ API).

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

Parameters

sqlca A pointer to the SQLCA.

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

For more information, see "UL_STORE_PARMS macro" on page 428.

IsOpen method

Prototype bool **IsOpen ( )**

Description Checks whether the ULData object is currently open.

Returns **true** (1) if the ULData object is open.

false (0) if the ULData object is not open.

Example The following example declares a ULData object, opens it, and checks that the Open method succeeded:

```
ULData db;
db.Open();
if( db.IsOpen() ){
   printf( "The ULData object is open\n" );
}
```

See also "Open method" on page 146

Open method

Prototype

bool **Open ( )**

bool **Open(** SQLCA* *ca* **)**

bool **Open(** ul_char* *parms* **)**

bool **Open(** SQLCA* *ca* , ul_char* *parms* **)**

Description

Prepares your application to work with a database. You must open the ULData object before carrying out any other operations on the database using the C++ API. Exceptions to this rule are as follows:

- On the Palm Computing Platform, the **ULData.PalmLaunch** method is called before **ULData.Open**. The resources that this library requires for your program are allocated and initialized on this call.

  On the Palm Computing Platform, call **ULData.Open** whenever **ULData.PalmLaunch** returns LAUNCH_SUCCESS_FIRST. For more information, see "PalmLaunch method" on page 148.

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

  For special purposes, you can specify persistent storage parameters when opening a database to configure caching, encryption, and the database file name. For information on these parameters, see "Configuring and managing database storage" on page 45.

For multi-threaded applications, each thread must open its own **ULData** object. Neither the **ULData** object nor the other objects inherited from it (**ULConnection** and other classes) can be shared across threads.

Parameters

Open( ) This prototype can be used by most UltraLite applications. Any persistent storage parameters defined in the UL_STORE_PARMS macro are employed when opening the database.

Open(SQLCA* ca) Use this prototype if you are using embedded SQL as well as the C++ API in your application, and if you have a SQLCA in use, to access the same data using the C++ API.

Open(ul_char* parms) Persistent storage parameters can be specified using the UL_STORE_PARMS macro. This prototype provides an alternative way of specifying persistent storage parameters. The string is a semicolon-separated list of assignments, of the form *parameter=value*.

Open(SQLCA *ca, ul_char* parms) A call specifying both the SQLCA and persistent storage parameters.

For more information on persistent storage parameters, see "UL_STORE_PARMS macro" on page 428.

Returns

true (1) if successful.

false (0) if unsuccessful.

Example

The following example declares a ULData object and opens it:

```
ULData db;
db.Open();
```

See also

"Close method" on page 144
"Configuring and managing database storage" on page 45
"Developing multi-threaded applications" on page 93
"UL_STORE_PARMS macro" on page 428

PalmExit method

Prototype

bool **PalmExit(** SQLCA ******ca* **)**

bool **PalmExit(** ul_synch_info * *synch_info* **)**

Description

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

For applications using HotSync or Scout Sync synchronization, the method also writes an upload stream. When the user uses HotSync or Scout Sync to synchronize data between their Palm device and a PC, the upload stream is read by the MobiLink HotSync conduit or the MobiLink Scout conduit respectively.

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

Parameters

sqlca A pointer to the SQLCA. You do not need to supply this argument unless you are using embedded SQL as well as the C++ API in your application and have used a non-default SQLCA.

synch_info A synchronization structure.

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

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

☞ For information on the members of the *synch_info* structure, see "Synchronization parameters" on page 380.

Returns

true (1) if successful.

false (0) if unsuccessful

PalmLaunch method

Prototype

UL_PALM_LAUNCH_RET **PalmLaunch( )**;

UL_PALM_LAUNCH_RET **PalmLaunch(** ul_synch_info * *synch_info* **)**;

UL_PALM_LAUNCH_RET **PalmLaunch(** SQLCA* *ca* **)**;

UL_PALM_LAUNCH_RET **PalmLaunch(** SQLCA* *ca* ,
ul_synch_info * *synch_info* **)**;

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

Description

This function restores the application state when the application is activated. For applications using HotSync or Scout Sync synchronization, it carries out the additional task of processing the download stream prepared by the MobiLink HotSync conduit or MobiLink Scout conduit.

If you are using TCP/IP or HTTP synchronization, supply a null value for the stream parameter in the **ul_synch_info** synchronization structure. This information is supplied instead in the synchronization structure called by the **ULConnection.Synchronize** method.

Parameters

ca A pointer to the SQLCA. You do not need to supply this argument unless you are using embedded SQL as well as the C++ API in your application and have used a non-default SQLCA.

synch_info A synchronization structure. For information on the members of this structure, see "Synchronization parameters" on page 380.

If you are using TCP/IP or HTTP synchronization, supply a null value for the **stream** parameter.

Returns

A member of the **UL_PALM_LAUNCH_RET** enumeration. The return values have the following meanings:

- **LAUNCH_SUCCESS_FIRST** This value is returned the first time the application is successfully launched and at any subsequent time the internal state of the UltraLite database needs to be re-established. In general, the state of the database needs to be re-established only after severe failures.

  You should open a **ULData** object when LAUNCH_SUCCESS_FIRST is returned.

- **LAUNCH_SUCCESS** This value is returned when an application is successfully launched, after the Palm user has been using other applications.

- **LAUNCH_FAIL** This value is returned when the launch fails.

Examples

A typical C++ API example is

```
ULData db;
ULEnablePalmRecordDB( & sqlca );
switch( db.PalmLaunch( &synch_info ) ){
case LAUNCH_SUCCESS_FIRST:
   if( !db.Open() ){
      // initialization failed: add error handling here
      break;
   }
   // fall through
case LAUNCH_SUCCESS:
   db.Reopen();
   // do work here
   break;
case LAUNCH_FAIL:
   // error
   break;
}
```

Reopen method

Prototype

bool **Reopen ( )**

bool **Reopen(** SQLCA* *ca* **)**

Description

This method is available for the Palm Computing Platform only.

When developing Palm applications, you should never close the database object. Instead, you should call **Reopen** when the user switches to the UltraLite application. The method prepares the data in use by the database object for use by the application.

Parameters

Open( ) No arguments are needed if you are not using embedded SQL as well as the C++ API in your application.

Open(SQLCA* ca) If you are also using embedded SQL in your application, and you have a non-default SQLCA in use, you can use this method to access the same data using the C++ API.

Returns

true (1) if successful.

false (0) if unsuccessful.

Example

The following example reopens a database object and a connection object:

```
db.Reopen();
conn.Reopen( &db );
```

See also

"Open method" on page 146

ULCursor class

The **ULCursor** class contains methods needed by both generated table objects and generated result set objects.

For its position in the API hierarchy, see "C++ API class hierarchy" on page 130.

Data types enumeration

This enumeration lists the available UltraLite data types, as constants. It contains the following members:

Enumeration value	Description
BAD_INDEX	An inappropriate argument was provided
S_LONG	Signed 4-byte integer
S_SHORT	Signed 2-byte integer
LONG	4-byte integer
SHORT	2-byte integer
TINY	1-byte integer
BIT	Bit
TIMESTAMP_STRUCT	Timestamp information as a struct.
DATE	Data and time information as a string
TIME	Time information as a string
S_BIG	Signed 8-byte integer
BIG	8-byte integer
DOUBLE	Double precision number
REAL	Real number
BINARY	Binary data, with a specified length
TCHAR	Character data, with a specified length
NUMERIC	Exact numerical data, with a specified precision and scale
MAX_INDEX	Reserved

The GetColumnType method returns a value from this enumeration.

See also "GetColumnType method" on page 156

SQL data types enumeration

This enumeration lists the available UltraLite SQL data types, as constants. It contains the following members:

```
enum {
        SQL_BAD_INDEX,
        SQL_S_LONG,
        SQL_U_LONG,
        SQL_LONG = SQL_U_LONG,
        SQL_S_SHORT,
        SQL_U_SHORT,
        SQL_SHORT = SQL_U_SHORT,
        SQL_S_BIG,
        SQL_U_BIG,
        SQL_BIG = SQL_U_BIG,
        SQL_TINY,
        SQL_BIT,
        SQL_TIMESTAMP,
        SQL_DATE,
        SQL_TIME,
        SQL_DOUBLE,
        SQL_REAL,
        SQL_NUMERIC,
        SQL_BINARY,
        SQL_CHAR,
        SQL_LONGVARCHAR,
        SQL_LONGBINARY,
        SQL_MAX_INDEX
    };
```

The GetColumnSQLType method returns a value from this enumeration.

See also "GetColumnSQLType method" on page 156

AfterLast method

Prototype bool **AfterLast( )**

Description Changes the cursor position to be after the last row in the current table or result set.

Returns **true** (1) if successful.

false (0) if unsuccessful.

Example The following example makes the current row the last row of the table **tb**:

```
tb.AfterLast();
tb.Previous();
```

See also "BeforeFirst method" on page 153
"Last method" on page 158

BeforeFirst method

Prototype bool **BeforeFirst( )**

Description Changes the cursor position to be before the first row in the current table or result set.

Returns **true** (1) if successful.

false (0) if unsuccessful.

Example The following example makes the current row the first row of the table **tb**:

```
tb.BeforeFirst();
tb.Next();
```

See also "AfterLast method" on page 152
"First method" on page 154

Close method

Prototype bool **Close( )**

Description Frees resources associated with the generated object in your application. This method must be called after all processing involving the table is complete, and before the ULConnection and ULData objects are closed.

Any uncommitted operations are rolled back when the **Close()** method is called.

Returns **true** (1) if successful.

false (0) if unsuccessful.

Example The following example closes a generated object for a table named ULProduct:

```
tb.Close();
```

See also "Open method" on page 177

Delete method

Prototype bool **Delete( )**

Description Deletes the current row from the current table or result set.

Returns **true** (1) if successful.

false (0) if unsuccessful. For example, if you attempt to use the method on a SQL statement that represents more than one table.

Example The following example deletes the last row from a table **tb**:

```
tb.Open( &conn );
tb.Last();
tb.Delete();
```

See also "Insert method" on page 157
"Update method" on page 162

First method

Prototype bool **First( )**

Description Moves the cursor to the first row of the table or result set.

Returns **true** (1) if successful.

false (0) if unsuccessful. For example, the method fails if there are no rows.

Example The following example deletes the first row from a table **tb**:

```
tb.Open( &conn );
tb.First();
tb.Delete();
```

See also "BeforeFirst method" on page 153
"Last method" on page 158

Get method

Prototype bool **Get(** ul_column_num *colnum*,
value-declaration,
bool* *isNull* = **UL_NULL)**

value-declaration:
ul_char * *ptr*, ul_length *length*
| p_ul_binary *name* , ul_length *length*
| DECL_DATETIME &*date-value*
| { DECL_BIGINT | DECL_UNSIGNED_BIGINT } &*bigint-value*
| [unsigned] long &*integer-value*
| unsigned char &*char-value*
| double & *double-value*
| float & *float-value*
| [unsigned] short &*short-value*

Description Gets a value from the specified column.

colnum A 2-byte integer. The first column is column 1.

value declaration The arguments required to specify the value depend on the data type. Character and binary data must be mapped into buffers, with the buffer name and length specified in the call. For other data types, a pointer to a variable of the proper type is needed. For character data, the length parameter specifies the length of the C array *including* the space used for the terminator.

isNULL If a value in a column is NULL, **isNull** is set to **true**. In this case, the **value** argument is not meaningful.

Returns **true** (1) if successful.

false (0) if unsuccessful.

See also "Get generated method" on page 175
"Set method" on page 161

GetColumnCount method

Prototype int **GetColumnCount( )**

Description Returns the number of columns in the current table or result set.

Returns Integer number of columns.

Example The following example opens a table object named `tb` and places the number of columns in the variable `numCol`:

```
tb.Open( &conn );
numCol = tb.GetColumnCount();
```

GetColumnSize method

Prototype ul_length **GetColumnSize(** ul_column_num *column-index* **)**

Description Returns the number of bytes needed to hold the information in the specified column.

Parameters **column-index** The number of the column. The first column in the table has a value of one.

Returns The number of bytes.

Example The following example gets the number of bytes needed to hold the third column in the table **tb**:

```
tb.Open( &conn );
colSize = tb.GetColumnSize( 3 );
```

See also "GetColumnType method" on page 156

GetColumnType method

Prototype int **GetColumnType(** ul_column_num *column-index* **)**

Description Returns the data type needed to hold the information in the specified column.

Parameters **column-index** The number of the column. The first column in the table or result set has a value of one.

Returns The column type is a member of the UltraLite data types enumeration. For more information, see "Data types enumeration" on page 151:

Example The following example gets the column type for the third column in the table **tb**:

```
tb.Open( &conn );
colType = tb.GetColumnType( 3 );
```

See also "Data types enumeration" on page 151
"Get generated method" on page 175
"GetColumnSQLType method" on page 156

GetColumnSQLType method

Prototype int **GetColumnSQLType(** ul_column_num *column-index* **)**

Description Returns the SQL data type of the specified column.

Parameters **column-index** The number of the column. The first column in the table or result set has a value of one.

Returns The column type is a member of the UltraLite data types enumeration. For more information, see "Data types enumeration" on page 151:

Example The following example gets the column type for the third column in the table **tb**:

```
tb.Open( &conn );
colType = tb.GetColumnType( 3 );
```

See also
"Data types enumeration" on page 151
"Get generated method" on page 175
"GetColumnType method" on page 156

GetSQLCode method

This is a convenience method that calls the **ULConnection::GetSQLCode** method.

For more information see "GetSQLCode method" on page 135.

Insert method

Prototype bool **Insert( )**

Description Inserts a row in the table with values specified in previous **Set** methods.

Returns **true** (1) if successful.

false (0) if unsuccessful.

Example The following example inserts a new row into the table based at the current position:

```
productTable.SetProd_id( 2 );
productTable.SetPrice( 3000 );
productTable.SetProd_name( "8’ 2x4 Studs x1000" );
productTable.Insert();
```

When inserting a row, you must supply a value for each column in the table.

For information on cursor positioning after inserts, and the position of the inserted row, see "Using table and query classes" on page 125.

See also
"Delete method" on page 154
"Update method" on page 162

IsOpen method

Prototype bool **IsOpen ( )**

Description Checks whether the ULCursor object is currently open.

Returns **true** (1) if the ULCursor object is open.

false (0) if the ULCursorobject is not open.

See also "Open method" on page 159

Last method

Prototype bool **Last( )**

Description Move the cursor to the last row in the table or result set.

Returns **true** (1) if successful.

false (0) if unsuccessful.

Example The following example moves to a position after the last row in a table:

```
tb.Open( &conn );
tb.Last();
tb.Next();
```

See also "AfterLast method" on page 152
"First method" on page 154

LastCodeOK method

This is a convenience method that calls the **ULConnection::LastCodeOK** method.

For more information see "LastCodeOK method" on page 138.

LastFetchOK method

This is a convenience method that calls the **ULConnection::LastFetchOK** method.

For more information see "LastFetchOK method" on page 158.

Next method

Prototype bool **Next( )**

Description Moves the cursor position to the next row in the table or result set.

Returns **true** (1) if successful.

false (0) if unsuccessful.

Example The following example moves the cursor position to the first row in the table:

```
tb.Open( &conn );
tb.BeforeFirst();
tb.Next();
```

See also "Previous method" on page 159
"Relative method" on page 160

Open method

Prototype bool **Open(** ULConnection * *conn* **)**

Description Opens a cursor on the specified connection. If the object is a result set with parameters, you must set the parameters before opening the result set.

When using **Open** from the **ULTable** subclass of **ULCursor**, do not open two connections on a **ULTable** objct at one time.

Returns **true** (1) if successful.

false (0) if unsuccessful.

Example The following example opens a result set object (which extends the cursor class) and moves the cursor position to the first row:

```
rs.Open( &conn );
rs.BeforeFirst();
rs.Next();
```

See also "Close method" on page 153
"Open method" on page 172

Previous method

Prototype bool **Previous( )**

Description Moves the cursor position to the previous row in the table or result set.

Returns **true** (1) if successful.

false (0) if unsuccessful.

Example

The following example moves to the last row in a table:

```
tb.Open( &conn );
tb.AfterLast();
tb.Previous();
```

See also

"Next method" on page 159
"Relative method" on page 160

Relative method

Prototype

bool **Relative(** ul_fetch_offset *offset* **)**

Description

Moves the cursor position relative to the current position. If the row does not exist, the method returns false, and the cursor is left at **AfterLast()** if *offset* is positive, and **BeforeFirst()** if *offset* is negative.

offset The number of rows to move. Negative values correspond to moving backwards.

Returns

true (1) if the row exists.

false (0) if the row does not exist.

See also

"Next method" on page 159
"Previous method" on page 159

Reopen method

Prototype

bool **Reopen(** ULConnection **conn* **)**

Description

This method is available for the Palm Computing Platform only. The **ULData** and **ULCOnnection** objects must already be reopened for this call to succeed.

When developing Palm applications, you should never close result set objects if you wish to maintain the cursor position. Instead, you should call **Reopen** when the user switches back to the UltraLite application.

Although the **ULTable** object inherits from the **ULCursor** class, you should not use **Reopen** on table objects. Instead, you should close them on exiting the Palm application and Open them on re-entering. The cursor position is not maintained in **ULTable** objects.

Parameters

conn A pointer to the **ULConnection** object on which the cursor is defined.

Returns **true** (1) if successful.

false (0) if unsuccessful.

Example The following example reopens a database object, and then a connection object, and then a result set object:

```
db.Reopen();
conn.Reopen( &db );
rs.Reopen( &conn );
```

See also "Open method" on page 139

Set method

Prototype bool **Set(** ul_column_num *colnum*, *value* **)**

value:
p_ul_binary *buffer-name*, ul_length *buffer-length*
| ul_char * *buffer-name*, ul_length *buffer-length* = 0
| DECL_DATETIME *date-value*
| DECL_UNSIGNED_BIGINT *bigint-value*
| unsigned char *char-value*
| double *double-value*
| float *float-value*
| [unsigned] long *long-value*
| [unsigned] short *short-value*

Description Sets a value in the specified column, for the current row.

colnum A 2-byte integer. The first column is column 1.

value For character and binary data you must supply a buffer name and length. For other data types, a value of the proper type is needed. The function fails if the data type is incorrect for the column.

Returns **true** (1) if successful.

false (0) if unsuccessful.

See also "Get method" on page 154

SetColumnNull method

Prototype int **SetColumNull(** ul_column_num *column-index* **)**

Description Sets a column to the SQL NULL. The data is not actually changed until you execute an Insert or Update, and that change is not permanent until it is committed.

Parameters **column-index** The number of the column. The first column in the table has a value of one.

Returns **true** (1) if successful.

false (0) if unsuccessful.

See also "SetNull<Column> generated method" on page 178

Update method

Prototype bool **Update( )**

Description Updates a row in the table with values specified in previous **Set** methods.

Returns **true** (1) if successful.

false (0) if unsuccessful.

Example The following example sets a new price on the current row of the **productTable** object, and then updates the row in the UltraLite database:

```
productTable.SetPrice( 400 );
productTable.Update();
```

See also "Delete method" on page 154
"Insert method" on page 157

ULResultSet class

The **ULResultSet** class extends the **ULCursor** class, and provides methods needed by all generated result sets.

For more information, see "ULCursor class" on page 151, and "Generated result set class" on page 171.

For its position in the API hierarchy, see "C++ API class hierarchy" on page 130.

SetParameter method

Prototype

virtual bool **SetParameter(** int *argnum*, *value-reference* **)**

value-reference:
[unsigned] long & *value*
| p_ul_binary *value*
| unsigned char & *value*
| ul_char * *value*
| double & *value*
| float & *value*
| [unsigned] short & *value*
| DECL_DATETIME *value*
| DECL_BIGINT *value*
| DECL_UNSIGNED_BIGINT *value*

Description

The following query defines a result set with a parameter:

```
SELECT id
FROM mytable
WHERE id < ?
```

The result set object defined in the C++ API that corresponds to this query has a parameter. You must set the value of the parameter before opening the generated result set object.

Parameters

argnum An identifier for the argument to be set. The first argument is 1, the second 2, and so on.

value-reference A reference to the parameter value. The data type listing above provides the possibilities. As the parameter are passed as pointers, they must remain valid until used. Do not free them until they are used.

Returns

true (1) if successful.

false (0) if unsuccessful. If you supply a parameter of the wrong data type, the method fails.

See also "Open method" on page 172

ULTable class

The **ULTable** class extends the **ULCursor** class, and provides methods needed by all generated table objects.

You cannot have multiple connections to a **ULTable** object at one time.

For its position in the API hierarchy, see "C++ API class hierarchy" on page 130.

DeleteAllRows method

Prototype ul_ret_void **DeleteAllRows( )**

Description The function deletes all rows in the table.

In some applications, it can be useful to delete all rows from tables before downloading a new set of data into the table. Rows can be deleted from the UltraLite database without being deleted from the consolidated database using the **ULConnection::StartSynchronizationDelete** method.

See also
"StartSynchronizationDelete method" on page 142
"StopSynchronizationDelete method" on page 142

Find method

Equivalent to the FindNext method.

See "FindNext method" on page 167.

FindFirst method

Prototype bool **FindFirst(** ul_column_num *ncols* **)**

Description Move forwards through the table from the beginning, looking for a row that exactly matches a value or set of values in the current index.

The current index is that used to specify the sort order of the table, It is specified when your application calls the generated table Open method. The default index is the primary key. For more information, see "Open method" on page 177.

To specify the value to search for, set the column value for each column in the index. The cursor is left on the first row that exactly matches the index value. On failure the cursor position is **AfterLast()**.

Parameters

ncols For composite indexes, the number of columns to use in the lookup. For example, if there is a three column index, and you want to lookup a value that matches based on the first column only, you should **Set** the value for the first column, and then supply an *ncols* value of 1.

Returns

true (1) if successful.

false (0) if unsuccessful.

See also

"FindLast method" on page 166
"FindNext method" on page 167
"FindPrevious method" on page 167
"LookupBackward method" on page 168
"LookupForward method" on page 169

FindLast method

Prototype

bool **FindLast(** ul_column_num *ncols* **)**

Description

Move backwards through the table from the end, looking for a row that matches a value or set of values in the current index.

The current index is that used to specify the sort order of the table, It is specified when your application calls the generated table Open method. The default index is the primary key. For more information, see "Open method" on page 177.

To specify the value to search for, set the column value for each column in the index. The cursor is left on the first row found that exactly matches the index value. On failure the cursor position is **BeforeFirst()**.

Parameters

ncols For composite indexes, the number of columns to use in the lookup. For example, if there is a three column index, and you want to lookup a value that matches based on the first column only, you should **Set** the value for the first column, and then supply an *ncols* value of 1.

Returns

true (1) if successful.

false (0) if unsuccessful.

See also

"FindFirst method" on page 165
"FindNext method" on page 167
"FindPrevious method" on page 167
"LookupBackward method" on page 168
"LookupForward method" on page 169

FindNext method

Prototype bool **FindNext(** ul_column_num *ncols* **)**

Description Move forwards through the table from the current position, looking for a row that exactly matches a value or set of values in the current index.

The current index is that used to specify the sort order of the table, It is specified when your application calls the generated table Open method. The default index is the primary key. For more information, see "Open method" on page 177.

To specify the value to search for, set the column value for each column in the index. The cursor is left on the first row found that exactly matches the index value. On failure, the cursor position is **AfterLast()**.

Parameters

ncols For composite indexes, the number of columns to use in the lookup. For example, if there is a three column index, and you want to lookup a value that matches based on the first column only, you should **Set** the value for the first column, and then supply an *ncols* value of 1.

Returns **true** (1) if successful.

false (0) if unsuccessful.

See also

"FindFirst method" on page 165
"FindLast method" on page 166
"FindPrevious method" on page 167
"LookupBackward method" on page 168
"LookupForward method" on page 169

FindPrevious method

Prototype bool **FindPrevious(** ul_column_num *ncols* **)**

Description Move backwards through the table from the current position, looking for a row that exactly matches a value or set of values in the current index.

The current index is that used to specify the sort order of the table, It is specified when your application calls the generated table Open method. The default index is the primary key. For more information, see "Open method" on page 177.

To specify the value to search for, set the column value for each column in the index. The cursor is left on the first row found that exactly matches the index value. On failure the cursor position is **BeforeFirst()**.

Parameters

ncols For composite indexes, the number of columns to use in the lookup. For example, if there is a three column index, and you want to lookup a value that matches based on the first column only, you should **Set** the value for the first column, and then supply an *ncols* value of 1.

Returns

true (1) if successful.

false (0) if unsuccessful.

See also

"FindFirst method" on page 165
"FindLast method" on page 166
"FindNext method" on page 167
"LookupBackward method" on page 168
"LookupForward method" on page 169

Lookup method

Equivalent to the LookupForward method.

See "LookupForward method" on page 169

GetRowCount method

Prototype

ul_ul_long **GetRowCount( )**

Description

The function returns the number of rows in the table.

One use for the function is to decide when to delete old rows to save space. Old rows can be deleted from the UltraLite database without being deleted from the consolidated database using the **ULConnection::StartSynchronizationDelete** method.

Returns

The number of rows in the table.

See also

"StartSynchronizationDelete method" on page 142
"StopSynchronizationDelete method" on page 142

LookupBackward method

Prototype

bool **LookupBackward(** ul_column_num *ncols* **)**

Description

Move backwards through the table starting from the end, looking for the first row that matches or is less than a value or set of values in the current index.

The current index is that used to specify the sort order of the table, It is specified when your application calls the generated table Open method. The default index is the primary key. For more information, see "Open method" on page 177.

To specify the value to search for, set the column value for each column in the index. The cursor is left on the first row that matches or is less than the index value. On failure (that is, if no row is less than the value being looked for), the cursor position is **BeforeFirst()**.

Parameters

ncols For composite indexes, the number of columns to use in the lookup. For example, if there is a three column index, and you want to lookup a value that matches based on the first column only, you should **Set** the value for the first column, and then supply an *ncols* value of 1.

Returns

true (1) if successful.

false (0) if unsuccessful.

See also

"FindFirst method" on page 165
"FindLast method" on page 166
"FindNext method" on page 167
"FindPrevious method" on page 167
"LookupForward method" on page 169

LookupForward method

Prototype

bool **LookupForward(** ul_column_num *ncols* **)**

Description

Move forward through the table starting from the beginning, looking for the first row that matches or is greater than a value or set of values in the current index.

The current index is that used to specify the sort order of the table, It is specified when your application calls the generated table Open method. The default index is the primary key. For more information, see "Open method" on page 177.

To specify the value to search for, set the column value for each column in the index. The cursor is left on the first row that matches or is greater than the index value. On failure (that is, if no rows are greater than the value being looked for), the cursor position is **AfterLast()**.

Parameters

ncols For composite indexes, the number of columns to use in the lookup. For example, if there is a three column index, and you want to lookup a value that matches based on the first column only, you should **Set** the value for the first column, and then supply an *ncols* value of 1.

Returns

true (1) if successful.

false (0) if unsuccessful.

See also

"FindFirst method" on page 165
"FindLast method" on page 166
"FindNext method" on page 167
"FindPrevious method" on page 167
"LookupBackward method" on page 168

Generated result set class

Object

The generated result set class represents a query result set to your application. The name of the class is generated by the UltraLite generator, based on the name of the statement supplied when it was added to the database.

Prototype

To create a generated result set object, you use the generated name in the declaration

result-set rs;
rs.**Open()**;

result-set: generated name

Description

The UltraLite generator defines a class for each named statement in an UltraLite project that returns a result set. This class inherits methods from **ULCursor**.

For its position in the API hierarchy, see "C++ API class hierarchy" on page 130.

See also

"ULCursor class" on page 151
"ul_add_statement system procedure" on page 411

Get<Column> generated method

Prototype

bool **Get***column-name***(** type* *variable*, [ul_length* *length*,]
bool* *isNull* = **UL_NULL)**

Description

Retrieves a value from *column-name*. The type specification depends on the column data type.

Parameters

column-name The name of the column.

variable A variable of the proper data type for the column. This data type can be retrieved using **GetColumnType**.

length For variable length data. For character data, the length parameter specifies the length of the C array *including* the space used for the terminator.

isNull If the value is NULL, this argument is **true**.

Returns

true (1) if successful.

false (0) if unsuccessful.

See also

"Set<Column> generated method" on page 172

Open method

Prototype

bool **Open(** ULConnection * *conn*,
datatype *value*, … **)**

Description

The UltraLite generator defines a class for each named statement in an UltraLite project that returns a result set. This class inherits methods from **ULCursor**.

You must supply a value for each placeholder in the result set.

Parameters

conn The connection on which the result set is to be opened.

value The value for the placeholder in the result set.

Example

The following query contains a single placeholder:

```
select prod_id, price, prod_name
from "DBA".ulproduct
where price < ?
```

The generator writes out the following methods for the object (in addition to some others):

```
bool Open( ULConnection* conn,
    long Price   );
bool Open( ULConnection* conn );
bool SetParameter( int index, long &value );
```

See also

"SetParameter method" on page 163

Set<Column> generated method

Prototype

bool **Set***column-name***( )**

Description

Sets the value of the cursor at the current position. The data in the row is not actually changed until you execute an Insert or Update, and that change is not permanent until it is committed.

Parameters

column-name A generated name derived from the name of the column in the reference database.

Returns

true (1) if successful.

false (0) if unsuccessful.

See also

"Get<Column> generated method" on page 171
"SetNull<Column> generated method" on page 173

SetNull<Column> generated method

Prototype bool **SetNull***column-name***( )**

Description Sets a column to the SQL NULL. The data is not actually changed until you execute an Insert or Update, and that change is not permanent until it is committed.

Parameters **column-name** A generated name derived from the name of the column in the reference database.

Returns **true** (1) if successful.

false (0) if unsuccessful.

See also "Set<Column> generated method" on page 172

Generated statement class

For each SQL statement that does not return a result set, including inserts, updates, and deletes, the UltraLite generator defines a generated statement class. The name of the class is the name provided in the *ul_add_statement* stored procedure call that added the statement to the reference database.

The generated statement class inherits from the **ULStatement** class, which has no methods of its own.

For its position in the API hierarchy, see "C++ API class hierarchy" on page 130.

Execute method

Prototype

bool **Execute(** ULConnection* *conn*,
[*datatype column-name*,...] **)**

Description

Executes a named statement that does not return a result set. Any change made is not permanent until it is committed.

When a statement is defined using *ul_add_statement*, you supply placeholders for the values, and supply them at run time. The generated prototype has a data type and name for each value.

Parameters

conn The connection on which the statement is to be executed.

datatype A member of the UltraLite data type enumeration.

column-name The name of the column.

Returns

true (1) if successful.

false (0) if unsuccessful.

See also

"ul_add_statement system procedure" on page 411

Generated table class

Object

The generated table class represents a database table to your application. The name of the class is generated by the UltraLite generator, based on the name of the table in the database.

Prototype

Tablename tb;
    tb.**Open()**;

Tablename:
    generated name

Description

The UltraLite generator defines a class for each table in a named SQL Remote publication. The generated table class inherits from **ULTable** and **ULCursor**. The class has a name based on the table or statement name, so that for a table named *Product,* the generator defines a class named **Product**.

For its position in the API hierarchy, see "C++ API class hierarchy" on page 130.

See also

"ULCursor class" on page 151
"ULTable class" on page 165

Get generated method

Prototype

bool **Get (**
    ul_column_num *column-index***,**
    *value-declaration***,**
    bool* *is-null* = **UL_NULL);**

value-declaration:
    ul_char * *buffer-name***,** ul_length *buffer-length*
  | p_ul_binary *buffer-name***,** ul_length *buffer-length*
  | DECL_DATETIME & *date-value*
  | { DECL_BIGINT | DECL_UNSIGNED_BIGINT } & *bigint-value*
  | unsigned char & *char-value*
  | double & *double-value*
  | float & *float-value*
  | [unsigned] long & *integer-value*
  | [unsigned] short & *short-value*

Description

Gets a value of from a column, specified by index.

Parameters

column-index The number of the column. The first column in the table has a value of one.

value declaration The arguments required to specify the value depend on the data type. Character and binary data must be mapped into buffers, with the buffer name and length specified in the call. For other data types, a pointer to a variable of the proper type is needed. For character data, the length parameter specifies the length of the C array *including* the space used for the terminator.

isNULL If a value in a column is NULL, **isNull** is set to **true**. In this case, the **value** argument is not meaningful.

Returns

true (1) if successful.

false (0) if unsuccessful.

Example

The following example is part of a **switch** statement that gets values from rows based on their data type:

```
switch( tb.GetColumnType( colIndex ) ) {
case tb.S_LONG :
   ret = tb.Get( colIndex, longval );
   printf( "Long column: %d\n", longval );
   break;
...
```

See also

"Data types enumeration" on page 151
"Get method" on page 154
"Get<Column> generated method" on page 176
"GetColumnSize method" on page 155

Get<Column> generated method

Prototype

bool **Get***column-name***(** type* *variable*, [ul_length* *length*,]
bool* *isNull* = **UL_NULL)**

Description

Retrieves a value from *column-name*. The type specification depends on the column data type.

Parameters

column-name The name of the column.

variable A variable of the proper data type for the column. This data type can be retrieved using **GetColumnType**.

length For variable length data types. For character data, the length parameter specifies the length of the C array *including* the space used for the terminator.

isNull If the value is NULL, this argument is **true**.

Returns

true (1) if successful.

false (0) if unsuccessful.

See also "Get generated method" on page 175

GetSize<Column> generated method

Prototype ul_length **GetSize***column-name***( )**

Description Returns the storage area needed to hold a value from the specified column.

Parameters **column-name** A generated name derived from the name of the column in the reference database.

Returns **true** (1) if successful.

false (0) if unsuccessful.

See also "GetColumnType method" on page 156

Open method

Prototype bool **Open(** ULConnection* *conn* **)**

bool **Open(** ULConnection* *conn*, ul_index_num *index* **)**

Description Prepares your application to work with the data in a generated table object.

Parameters **conn** The address of a ULConnection object. The connection must be open.

index An optional index number, used to order the rows in the table. The index is one of the members of the generated index enumeration. By default, the table is ordered by primary key value.

For more information, see "Index enumeration" on page 178.

When the table is opened, the cursor is positioned before the first row

Returns **true** (1) if successful.

false (0) if unsuccessful.

Example The following example declares a generated object for a table named ULProduct, and opens it:

```
ULData db;
ULConnection conn;
ULProduct tb;
db.Open();
conn.Open( &db, "DBA", "SQL" );
tb.Open( &conn );
```

See also "Close method" on page 153
"Index enumeration" on page 178

Set<Column> generated method

Prototype bool **Set***column-name***( )**

Description Sets the value of the cursor at the current position. The data in the row is not actually changed until you execute an Insert or Update, and that change is not permanent until it is committed.

Parameters **column-name** A generated name derived from the name of the column in the reference database.

Returns **true** (1) if successful.

false (0) if unsuccessful.

See also "SetColumnNull method" on page 161

SetNull<Column> generated method

Prototype bool **SetNull***column-name***( )**

Description Sets a column to the SQL NULL. The data is not actually changed until you execute an Insert or Update, and that change is not permanent until it is committed.

Parameters **column-name** A generated name derived from the name of the column in the reference database.

Returns **true** (1) if successful.

false (0) if unsuccessful.

See also "SetColumnNull method" on page 161

Index enumeration

Prototype **enum**{ *index-name*,... }

Description

Each member of the enumeration is an index name in the table being generated. You can use the index name to specify an ordering for the table when it is opened, and thereby control the behavior of the cursor movement methods.

Parameters

index-name The name of an index in the table. The primary key has the name **Primary**, and other indexes have their name in the database.

See also

"Open method" on page 177

CHAPTER 8

Tutorial: Build an Application Using Embedded SQL

About this chapter

This chapter provides a tutorial that guides you through the process of developing an embedded SQL UltraLite application. The first section includes a sample embedded SQL source file and discusses the key elements in the sample source file. The second section provides instructions for building an UltraLite application using this sample source file.

For an overview of the development process and background information on the UltraLite database, see "Designing UltraLite Applications" on page 41.

For information on developing embedded SQL UltraLite Applications, see "Developing Embedded SQL Applications" on page 193.

For a description of embedded SQL, see "The Embedded SQL Interface" on page 205.

Contents

Introduction

In this tutorial, you will create an embedded SQL source file and use this source file to build a simple UltraLite application. The next section "Writing source files in embedded SQL" on page 183 provides a sample embedded SQL program. Copy this program into a new file and save it as a *.sqc* source file. Then, follow the step by step instructions in "Building the sample embedded SQL UltraLite application" on page 187 to build the UltraLite application. The UltraLite application can be executed in the command prompt on your PC.

This tutorial assumes that you have UltraLite installed on a machine with Microsoft Visual C++ 6.0 installed. If you use a different C/C++ development tool, you will have to translate the Visual C++ instructions into their equivalent for your development tool.

The source files for this tutorial can be found in the *Samples\UltraLite\ESQLTutorial* subdirectory of your SQL Anywhere directory.

❖ **To prepare for the tutorial**

- Create a directory to hold the files you will create: *c:\esqltutorial.*

Writing source files in embedded SQL

The following sample program establishes a connection with the UltraLite *CustDB* sample database and executes a select query. Copy the following code into a new file and save it as *sample.sqc* in your *c:\esqltutorial* directory, or retype the material into a file.

You can also find this file as *Samples\UltraLite\ESQLTutorial\sample.sqc*.

```
#include <stdio.h>

EXEC SQL INCLUDE SQLCA;

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

   /* Before working with data*/
   db_init(&sqlca);

   /* Connect to database */
   EXEC SQL CONNECT "DBA" IDENTIFIED BY "SQL";

   /* Fill table with data first */
   EXEC SQL INSERT INTO ULProduct(
         prod_id, price, prod_name)
      VALUES (1, 400, '4x8 Drywall x100');
   EXEC SQL INSERT INTO ULProduct (
         prod_id, price, prod_name)
      VALUES (2, 3000, '8''2x4 Studs x1000');
   EXEC SQL COMMIT;

   /* Fetch row from database */

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

   /* Error handling.  If the row does not exist,
      or if an error occurs, -1 is returned */

   if((SQLCODE==SQLE_NOTFOUND)||(SQLCODE<0)) {
      return(-1);
   }

   /* Print query results */
   printf("Product id: %ld Price: %ld Product name: %s",
      pid, cost, pname);
```

```
    /* Preparing to exit:
    rollback any outstanding changes and disconnect */
    EXEC SQL ROLLBACK;
    EXEC SQL DISCONNECT;
    db_fini(&sqlca);
    return(0);
}
```

Tip
You can configure Visual C++ to provide syntax highlighting for *.sqc* files, by adding **;sqc** to the list of file extensions in the following registry location:

```
HKEY_CURRENT_USER\Software\Microsoft\DevStudio\6.0\
Text Editor\Tabs\Language Settings\C/C++\FileExtensions
```

Explanation of the sample program

Although too simple to be useful, this example contains elements that must be present in every embedded SQL source file used for database access. The following describes the key elements in the sample program. Use these steps as a guide when creating your own embedded SQL UltraLite application.

1 Include the appropriate header files.

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

2 Define the **SQL communications area**, sqlca.

Use the following command:

```
EXEC SQL INCLUDE SQLCA;
```

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

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

3 Define host variables by creating a declaration section.

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

```
EXEC SQL BEGIN DECLARE SECTION;
    long pid=1;
```

```
    long cost;
    char pname[31];
EXEC SQL END DECLARE SECTION;
```

For information on host variables, see "Using host variables" on page 209.

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

Call this function as follows:

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

5 Connect to the database using the CONNECT statement.

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

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

6 Insert data into database tables.

When an application is first started, its database tables are empty. Only when you choose to synchronize the remote database with the consolidated database will the tables be filled with values so that you may execute select, update or delete commands. Rather than using synchronization, however, you may also directly insert data into the tables. Directly inserting data is a useful technique during the early stages of UltraLite development.

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

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

For information on adding synchronization to an UltraLite application, see "Adding synchronization to your application" on page 190.

7 Execute your SQL query.

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

8 Perform error handling.

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

9 Disconnect from the database.

You should rollback or commit any outstanding changes before disconnecting.

To disconnect, use the DISCONNECT statement as follows:

```
EXEC SQL DISCONNECT;
```

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

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

Building the sample embedded SQL UltraLite application

After you have created a source file *sample.sqc* using the sample code in the previous section, you are ready to build your UltraLite application. Follow these steps in Microsoft Visual C++ 6.0 to create the sample embedded SQL UltraLite application.

❖ **To build the sample embedded SQL UltraLite application:**

1. Start the Adaptive Server Anywhere personal database server.

   By starting the database server, both the SQL preprocessor and the UltraLite analyzer will have access to your reference database. The sample application uses the CustDB sample database *custdb.db* as a reference database and as consolidated database. Start the database server at the command line from the *Samples\UltraLite\CusDB* directory containing *custdb.db* as follows:

   ```
   dbeng8.exe custdb.db
   ```

   Alternatively, you can start the database server by selecting Start➤Programs➤SQL Anywhere 8➤UltraLite➤Personal Server Sample for UltraLite.

2. Start Microsoft Visual C++ from your desktop in the standard fashion.

3. Configure Visual C++ to search the appropriate directories for Embedded SQL header files and UltraLite library files.

   Select Tools➤Options and click on the Directories tab. Choose Include Files under the Show Directories For drop down menu. Include the following directory, so that the embedded SQL header files can be accessed.

   ```
   C:\Program Files\Sybase\SQL Anywhere 8\h
   ```

   If you have installed SQL Anywhere to a directory other than the default, substitute your installation directory above. On the same tab, select Library Files under the Show Directories For drop down menu. Include the following directory so that the UltraLite library files can be accessed.

   ```
   C:\Program
   Files\Sybase\SQL Anywhere 8\UltraLite\win32\386\lib
   ```

   Click OK to submit the changes.

4. Create a new workspace **tutorial**:

- Select File➤New.
- Click the Workspaces tab.
- Choose Blank Workspace. Specify a workspace name **tutorial** and specify *C:\esqltutorial\tutorial* as the location to save this workspace. Click OK. Workspace **tutorial** will be added to the Workspace window.

5 Create a new project **ultutorial** and add it to the **tutorial** workspace:

- Select File➤New.
- Click the Project tab.
- Choose Win32 Console Application. Specify a project name **ultutorial** and select Add To Current Workspace. Click OK. Choose to create An Empty Project and click Finish. The project will be saved in the **tutorial** folder.
- Click the FileView tab on the Workspace window. The workspace tutorial now consists of one project. Project **ultutorial** is listed under the workspace **tutorial** and has three folders: Source Files, Header Files, Resource Files.

6 Add the *sample.sqc* embedded SQL source file to the project:

- Right click on the Source Files folder and select Add Files to Folder. Locate your *sample.sqc* file and click OK. Open the Source Files folder to verify that it contains *sample.sqc*.

7 Configure the *sample.sqc* source file settings to invoke the SQL preprocessor to preprocess the source file:

- Right click on the *sample.sqc* file in the Workspace window and select Settings. A Project Settings dialog appears.
- From the Settings For drop down menu, choose All Configurations.
- In the Custom Build tab, enter the following statement in the Commands box. Ensure that the statement is entered all on one line.

  ```
  "%asany8%\win32\sqlpp.exe" -q -o WINNT –c
  "dsn=Ultralite 8.0 Sample" $(InputPath) sample.cpp
  ```

- Specify *sample.cpp* in the Outputs box.
- Click OK to submit the changes. This statement runs the SQL preprocessor *sqlpp* on the *sample.sqc* file, and writes the processed output in a file named *sample.cpp*. The SQL preprocessor translates SQL statements in the source file into C/C++.

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

8 Preprocess the *sample.sqc* file:

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

9 Add the *sample.cpp* file to the project:

- Right click on the Source Files folder in the Workspace window and select Add Files to Folder. Locate your *sample.cpp* file (in *c:\sql\tutorial\tutorial\ultutorial*) and click OK. The *sample.cpp* file appears inside the Source Files folder.

10 Configure the project settings:

- Right click on the **ultutorial** project and select Settings. The Project Settings dialog appears.
- Select All Configurations under the Settings For drop down menu.
- Click the Link tab and add the following runtime library to the Object/Library Modules box.

  ```
  ulimp.lib
  ```

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

  ```
  __NT__,UL_USE_DLL
  ```

- Click OK to close the dialog.

11 Build the executable:

- Select Build➤Build ultutorial.exe. The **ultutorial** executable will be created. Depending on your settings, the executable may be created in a Debug directory within your tutorial directory.

12 Run the application:

- Select Build➤Execute ultutorial.exe. A screen will appear and display the first row of the product table.

Adding synchronization to your application

Once you have tested that your program is functioning properly, you can remove the lines of code that manually insert data into the *ULProduct* table. Replace these statements with a call to the **ULSynchronize** function to synchronize the remote database with the consolidated database. This process will fill the tables with data and you can subsequently execute a select query.

Synchronization via TCP/IP

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

```
auto ul_synch_info synch_info;
ULInitSynchInfo( &synch_info );
synch_info.user_name = UL_TEXT("50");
synch_info.version = UL_TEXT("custdb");
synch_info.stream = ULSocketStream();
synch_info.stream_parms =
   UL_TEXT("host=localhost;port=2439");
ULSynchronize( &sqlca, &synch_info );
```

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

For more information on the ULSynchronize function, see "ULSynchronize function" on page 250.

Running the sample application with synchronization

After you have made changes to the *sample.sqc* file, you must preprocess the altered *sample.sqc* file and rebuild *ultutorial.exe*.

This section assumes that you have completed the tutorial in the previous section and therefore have the appropriate project settings for the **ultutorial** project. If, however, you have not run the tutorial, you should refer to the guidelines in the previous section for including the appropriate project settings.

❖ **To rebuild the sample application:**

1. Ensure that the Adaptive Server Anywhere database server is still running.
2. Preprocess your new *sample.sqc* file.

   In the Workspace window, right click on *sample.sqc* and select Settings. Ensure that the Commands box contains the appropriate command for running the SQL preprocessor. Choose Build➤Compile *sample.sqc* to recompile the altered file. A new *sample.cpp* file will be generated.
3. Build the executable.

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

❖ **To run the sample application:**

1. Start the MobiLink synchronization server:
   - Select Start➤Programs➤Sybase SQL Anywhere 8➤MobiLink➤Synchronization Server Sample, or
   - At a command prompt execute the following command (on a single line):

     ```
     dbmlsrv8 –c "DSN=UltraLite 8.0 Sample"
     –o ulsync.mls –vcr –x tcpip
     ```
2. Run the application:
   - Select Build➤Execute ultutorial.exe to run the sample application.

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

CHAPTER 9

Developing Embedded SQL Applications

About this chapter

Embedded SQL is a method of mixing SQL statements for database access with either C or C++ source code.

This chapter describes the development process for embedded SQL UltraLite applications. It explains how to write applications using embedded SQL and provides instructions on building and deploying an embedded SQL UltraLite application.

Contents

Before you begin

This chapter assumes an elementary familiarity with the UltraLite development process. For an overview, see "Developing UltraLite Applications" on page 67

For a description of embedded SQL, see "The Embedded SQL Interface" on page 205.

For detailed information about the SQL preprocessor, see "The SQL preprocessor" on page 415.

Building embedded SQL applications

You can use a simpler build procedure if all your embedded SQL source code is contained in one file.

If you have only one embedded SQL source file *and* specify no project name when you run the SQL preprocessor, then the SQL preprocessor automatically runs the UltraLite generator. The supplementary code is generated and appended to the generated C/C++ source file.

If you specify a project name, or use more than one embedded SQL source file, you must generate the UltraLite code using the UltraLite generator.

Single-file build process

If you have a single file containing embedded SQL code, you can use the SQL preprocessor to run the UltraLite generator when it completes processing of your file. The generator appends extra C/C++ code to the single generated source file.

This single-file build process cannot be used if you wish to use any of the following features:

- transport-layer security.
- publications for synchronization.

In these circumstances you must use the general build process. For instructions, see "Build process for UltraLite embedded SQL applications " on page 195.

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

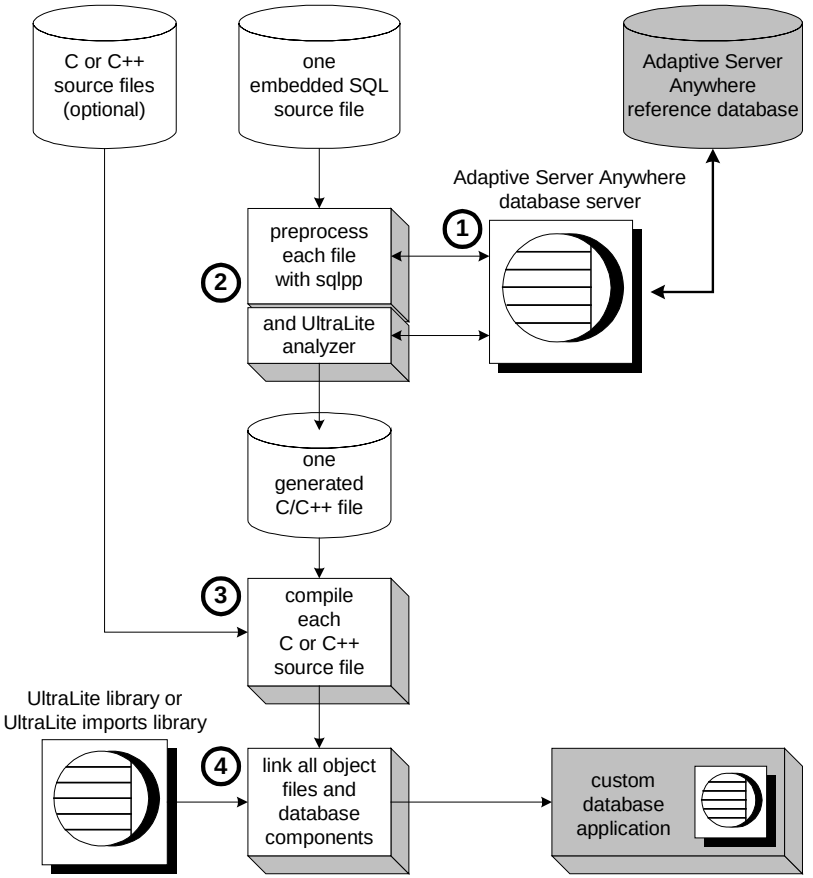
1. Start the Adaptive Server Anywhere personal database server, specifying your reference database and a cache size of at least 10 Mb.
2. Preprocess the embedded SQL source file using the SQL preprocessor, supplied with SQL Anywhere. Do not specify a project. The SQL preprocessor runs the UltraLite generator automatically and appends additional code to the generated C/C++ source file. This step relies on your reference database and on the database server.

   For more information, see "Preprocessing your embedded SQL files" on page 201.
3. Compile the C or C++ source file for the target platform of your choice. Include

- the C file generated by the SQL preprocessor,
- any additional C/C++ source files that comprise your application.

4 Link *all* these object files, together with the UltraLite runtime library.

The following diagram depicts the procedure for building your own UltraLite database application. In addition to your source files, you need to create a reference database. As explained below, this database plays dual roles, acting as an instance of the schema used in your application and storing information that the UltraLite analyzer needs to implement your database.



Build process for UltraLite embedded SQL applications

The build process for embedded SQL UltraLite applications has two steps:

1. Preprocess each embedded SQL file (*.sqc*) to produce *.cpp* files.

   You must supply a project name on the preprocessor command line and use the same project name each time you preprocess an embedded SQL source file.

   For information about projects, see "Creating an UltraLite project" on page 80.

2. Run the UltraLite generator to generate the database code.

Sample code

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

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

❖ **To build an UltraLite embedded SQL application:**

1. Start the Adaptive Server Anywhere personal database server, specifying your reference database.
2. Preprocess *each* embedded SQL source file using the SQL preprocessor, specifying the project name. This step relies on your reference database and on the database server.

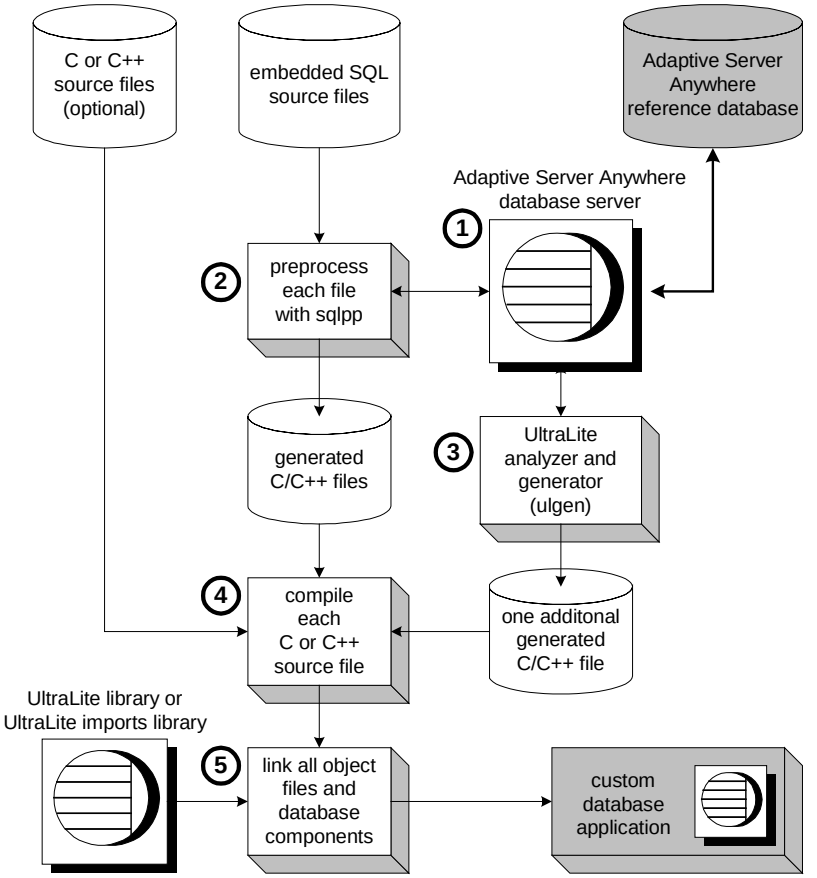
   For more information, see "Preprocessing your embedded SQL files" on page 201.

3. Run the UltraLite generator. The generator uses the analyzer, inside your reference database, to analyze information collected while pre-processing your embedded SQL files. The analyzer prepares extra code and the generator writes out a new C source file. This step also relies on your reference database and database server.

   For more information, see "Generating the UltraLite data access code" on page 91.

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

The following diagram depicts the procedure for building your own UltraLite database application. In addition to your source files, you need to create a reference database. As explained below, this database plays dual roles, acting as an instance of the schema used in your application and storing information that the UltraLite analyzer needs to implement your database.



Each section remaining in this chapter describes one step in writing or building your application.

For more information about the SQL preprocessor, see "Preprocessing your embedded SQL files" on page 201.

For more information on the generator, see "Generating the UltraLite data access code" on page 91.

Configuring development tools for embedded SQL development

The SQL preprocessor and the UltraLite generator are key features of UltraLite application development. Most development tools use a dependency-based model to assist in compilation, and this section describes how to incorporate UltraLite features into such a model.

For a general overview of the techniques needed, see "Configuring development tools for UltraLite development" on page 102.

The UltraLite plug-in for Metrowerks CodeWarrior automatically provides Palm Computing platform developers with the techniques described here. For information on this plug-in, see "Developing UltraLite applications with Metrowerks CodeWarrior" on page 255.

This section describes how to add UltraLite code generation and the SQL preprocessor into a dependency-based development environment. The specific instructions provided are for Visual C++.

For a tutorial describing details for a very simple project, see "Tutorial: Build an Application Using Embedded SQL" on page 181.

SQL preprocessing

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

❖ **To add embedded SQL preprocessing into a dependence-based development tool:**

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

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

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

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

     ```
     "%asany8%\win32\sqlpp.exe" –q –o WINNT
     –c connection-string –p project-name
     $(InputPath) $(InputName).c
     ```

     where *asany8* is an environment variable that points to your SQL Anywhere installation directory, *connection-string* provides the connection to the reference database, and *project-name* is the name of your UltraLite project.

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

☞ For a full description of the SQL preprocessor command-line, see "The SQL preprocessor" on page 415.

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

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

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

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

- Under General or Preprocessor, add UL_USE_DLL to the Preprocessor definitions.
- Under Preprocessor, add *$(asany8)\h* and any other include folders you require to your include path, as a comma-separated list.

UltraLite code generation

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

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

1 Add a dummy file to your development project.

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

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

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

```
"%asany8%\win32\ulgen.exe" –q –c "connection-string"
$(InputName) $(InputName).c
```

where *asany8* is an environment variable that points to your SQL Anywhere installation directory, *connection-string* is a connection to your reference database, and *InputName* is the UltraLite project name, and should match the root of the text file name. The output is *$(InputName).c*.

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

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

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

4 Compile your dummy file to generate the *.c* file that implements the UltraLite database.

5 Add the generated UltraLite database file to your project and change its C/C++ settings.

6 Add the UltraLite imports library to your object/libraries modules list.

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

```
$(asany8)\ultralite\win32\386\lib\ulimp.lib
```

For other targets, choose the appropriate import library.

Preprocessing your embedded SQL files

The SQL preprocessor (*sqlpp.exe*) carries out two functions in an UltraLite development project:

- It preprocesses the embedded SQL files, producing C files to be compiled into your application.
- It adds the SQL statements to the reference database, for use by the UltraLite generator. You must use the –c switch to specify connection parameters that give *sqlpp* access to the reference database.

  For information on adding SQL statements to a reference database, see "Adding SQL statements to an UltraLite project" on page 81.

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

Preprocessing projects with a single embedded SQL source file

If all your embedded SQL code is contained in *one* source file, you may use the following technique to preprocess this code using the SQL preprocessor and generate the supplementary code in a single step. In this case, the preprocessor automatically runs the UltraLite generator, which writes additional code that describes your database schema and implements the SQL used in your application.

❖ **To preprocess an embedded SQL source file (one file only):**

- Specify *no* project name when you run *sqlpp*, the SQL preprocessor. When invoked without a project name, the preprocessor automatically runs the UltraLite generator and appends the additional C/C++ code, which implements your application database. A single C/C++ source file is generated.
- Do not run the UltraLite generator explicitly.

Example

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

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

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

If the analyzer cannot be invoked, an error results. For more information, see "Error on starting the analyzer" on page 92.

For a list of the parameters to *sqlpp*, see "The SQL preprocessor" on page 415.

Preprocessing projects with more than one embedded SQL file

If you have more than one embedded SQL source file, you must run the UltraLite generator separately. You must also run the UltraLite generator separately if you use transport-layer security or if you use synchronization publications.

You must specify an UltraLite project on the *sqlpp* command line so that the SQL statements in the files are grouped together in the same project in the reference database.

For information on projects, see "Creating an UltraLite project" on page 80.

❖ **To preprocess an embedded SQL source file (more than one file):**

- Preprocess each embedded SQL source file.

  Use the `-p` switch to specify a project name when you run the SQL preprocessor. Use the same project name when you preprocess each embedded SQL file in your project.

- Run the UltraLite generator to create the supplementary C/C++ code.

  For more information on the generator, see "Generating the UltraLite data access code" on page 91.

By default, the UltraLite generator, *ulgen*, writes to the file **project-name**.*c*. Choose a project name not already assigned to another C or embedded SQL source file, so that an existing file will not be overwritten.

Example

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

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

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

For detailed information about the SQL preprocessor, see "The SQL preprocessor" on page 415.

CHAPTER 10

The Embedded SQL Interface

About this chapter

UltraLite applications can interact with their database through a static embedded SQL programming interface, which is a subset of the embedded SQL interface available in Adaptive Server Anywhere. This chapter introduces the UltraLite embedded SQL interface and describes its language and functions.

For information about the full Adaptive Server Anywhere embedded SQL interface, see "Embedded SQL Programming" on page 163 of the book *ASA Programming Guide*.

Contents

Introduction

Embedded SQL lets you insert standard SQL statements into either C or C++ code. A **SQL preprocessor** translates these SQL statements into C and C++ compatible source code. You compile each preprocessed file as you would an ordinary C or C++ source file.

The preprocessor generates source code that makes calls to library functions. These functions are defined in a library or imports library. You include one of these libraries when you link your program.

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

```
#include <stdio.h>
EXEC SQL INCLUDE SQLCA;

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

Although too simple to be useful, this example demonstrates the following aspects common to all embedded SQL applications:

- Each SQL statement is prefixed with the keywords EXEC SQL.
- Each SQL statement ends with a semicolon.
- Some embedded SQL statements are not found in standard SQL. The INCLUDE SQLCA statement is one example.
- Embedded SQL provides library functions to perform some specific tasks. The functions **db_init** and **db_fini** are examples.

Before working with data

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

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

```
EXEC SQL INCLUDE SQLCA;
```

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

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

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

For more information, see "db_init function" on page 231.

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

Preparing to exit

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

1 Commit or rollback any outstanding changes.

2 Disconnect from the database.

3 End your SQL work with a call to a library function named *db_fini*.

If you leave changes to the database uncommitted when you exit, any uncommitted operations are automatically rolled back.

Error handling

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

Structure of embedded SQL programs

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

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

```
EXEC SQL INCLUDE SQLCA;
```

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

For information about UltraLite user authentication in embedded SQL applications, see "Managing user IDs and passwords" on page 86, and "User authentication for UltraLite databases" on page 442.

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

Using host variables

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

Declaring host variables

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

Whenever you use a host variable in a SQL statement, you must prefix the variable name with a colon (:) so that the SQL preprocessor can distinguish it from other identifiers allowed in the statement.

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

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

Example

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

```
/* Declare fields for personal data. */
EXEC SQL BEGIN DECLARE SECTION;
   long employee_number = 0;
   char employee_name[50];
   char employee_initials[8];
   char employee_phone[15];
EXEC SQL END DECLARE SECTION;

/* Fill variables with appropriate values. */

/* Insert a row in the database. */
EXEC SQL INSERT INTO Employee
   VALUES (:employee_number, :employee_name,
      :employee_initials, :employee_phone );
```

Data types in embedded SQL

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

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

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

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

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

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

- 16-bit signed integer.

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

- 32-bit signed integer.

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

- 4-byte floating point number.

  ```
  float f;
  ```

- 8-byte floating point number.

  ```
  double d;
  ```

- Packed decimal number.

  ```
  DECL_DECIMAL(p,s)
  ```

  ```
  typedef struct TYPE_DECIMAL {
     char array[1];
  } TYPE_DECIMAL;
  ```

- NULL-terminated blank-padded character string.

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

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

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

- NULL terminated UNICODE or wide character string.

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

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

- NULL terminated system-dependent character string.

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

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

- Fixed-length blank padded character string.

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

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

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

  ```
  DECL_VARCHAR(n) a; /* n >= 1 */

  typedef struct VARCHAR {
     unsigned short int len;
     TCHAR array[1];
  } VARCHAR;
  ```

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

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

```
DECL_BINARY(n) a; /* n >= 1 */

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

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

```
DECL_DATETIME a;

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

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

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

For more information, see the DATE_FORMAT, TIME_FORMAT, TIMESTAMP_FORMAT, and DATE_ORDER database options in "Database Options" on page 535 of the book *ASA Database Administration Guide*. While these options cannot be set during execution of an UltraLite program, their values are identical to the settings in the reference database used to generate the program.

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

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

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

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

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

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

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

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

DATE and TIME database types

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

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

Host variable usage

Host variables can be used in the following circumstances:

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

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

The scope of host variables

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

The preprocessor assumes all host variables are global

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

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

Examples

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

```
// Example demonstrating poor coding
EXEC SQL BEGIN DECLARE SECTION;
   long emp_id;
EXEC SQL END DECLARE SECTION;

long getManagerID( void )
{
   EXEC SQL BEGIN DECLARE SECTION;
      long manager_id = 0;
   EXEC SQL END DECLARE SECTION;

   EXEC SQL  SELECT manager_id
             INTO :manager_id
             FROM employee
             WHERE emp_number = :emp_id;
```

```
    return( manager_number );
}

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

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

```
// Rewritten example
#if 0
    // Declarations for the SQL preprocessor
    EXEC SQL BEGIN DECLARE SECTION;
        long emp_id;
        long manager_id;
    EXEC SQL END DECLARE SECTION;
#endif

long getManagerID( long emp_id )
{
    long manager_id = 0;

    EXEC SQL  SELECT manager_id
              INTO :manager_id
              FROM employee
              WHERE emp_number = :emp_id;

    return( manager_number );
}

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

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

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

Using expressions as host variables

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

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

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

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

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

```
EXEC SQL INCLUDE SQLCA;

#include <sqlerr.h>
#include <stdio.h>

typedef struct my_struct {
   long     host_field;
} my_struct;

#if 0
   // Because it ignores #if preprocessing directives,
   // SQLPP reads the following declaration.

   EXEC SQL BEGIN DECLARE SECTION;
      long    host_value;
   EXEC SQL END DECLARE SECTION;
#endif

// Make C/C++ recognize the 'host_value' identifier
// as a macro that expands to a struct field.
#define host_value my_s.host_field
```

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

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

```
void main( void )
{
   my_struct    my_s;

   db_init( &sqlca );
   EXEC SQL CONNECT "DBA" IDENTIFIED BY "SQL";

   EXEC SQL DECLARE my_table_cursor CURSOR FOR
      SELECT int_col FROM my_table order by int_col;

   EXEC SQL OPEN my_table_cursor;

   for( ; ; ) {
      // :host_value references my_s.host_field

      EXEC SQL FETCH NEXT AllRows INTO :host_value;

      if( SQLCODE == SQLE_NOTFOUND ) {
         break;
      }

      printf( "%ld\n", my_s.host_field );
   }

   EXEC SQL CLOSE my_table_cursor;
   EXEC SQL DISCONNECT;
   db_fini( &sqlca );
}
```

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

- pointer indirections

  ```
  *ptr

  p_struct->ptr

  (*pp_struct)->ptr
  ```

- array references

  ```
  my_array[ i ]
  ```

- arbitrarily complex lvalues

Using host variables in C++

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

```
typedef   short  a_bool;
#define   TRUE   ((a_bool)(1==1))
#define   FALSE  ((a_bool)(0==1))
```

```
public class {
    long  host_member;

    my_class();     // Constructor

    ~my_class();    // Destructor

    a_bool FetchNextRow( void );
        // Fetch the next row into host_member
} my_class;
```

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

```
EXEC SQL INCLUDE SQLCA;
#include "my_class.hpp"

#if 0
    // Because it ignores #if preprocessing directives,
    // SQLPP reads the following declaration.

    EXEC SQL BEGIN DECLARE SECTION;
        long  this_host_member;
    EXEC SQL END DECLARE SECTION;
#endif

// Macro used by the C++ compiler only.
#define this_host_member this->host_member

my_class::my_class()
{
    EXEC SQL DECLARE my_table_cursor CURSOR FOR
        SELECT int_col FROM my_table order by int_col;

    EXEC SQL OPEN my_table_cursor;
}

my_class::~my_class()
{
    EXEC SQL CLOSE my_table_cursor;
}

a_bool my_class::FetchNextRow( void )
{
    // :this_host_member references this->host_member
    EXEC SQL FETCH NEXT AllRows INTO :this_host_member;
    return( SQLCODE != SQLE_NOTFOUND );
}
```

```
void main( void )
{
   db_init( &sqlca );
   EXEC SQL CONNECT "DBA" IDENTIFIED BY "SQL";
   {
      my_class mc; // Created after connecting.
      while( mc.FetchNextRow() ) {
         printf( "%ld\n", mc.host_member );
      }
   }

   EXEC SQL DISCONNECT;
   db_fini( &sqlca );
}
```

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

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

Indicator variables

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

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

Example

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

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

Indicator variable values

The following table provides a summary of indicator variable usage.

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

Using indicator variables to handle NULL

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

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

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

Using indicator variables when inserting NULL

An INSERT statement can include an indicator variable as follows:

```
EXEC SQL BEGIN DECLARE SECTION;
short int employee_number;
char employee_name[50];
char employee_initials[6];
char employee_phone[15];
short int ind_phone;
EXEC SQL END DECLARE SECTION;

/* set values of empnum, empname,
   initials, and homephone */

if( /* phone number is known */ ) {
   ind_phone =  0;
} else {
   ind_phone = -1; /* NULL */
}

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

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

Using indicator variables when fetching NULL

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

☞ Errors and warnings are returned in the SQLCA structure, as described in "The SQL Communication Area" on page 228.

Fetching data

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

1 The SELECT statement returns at most one row.

2 The SELECT statement may return multiple rows.

Fetching one row

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

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

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

☍ Errors and warnings are returned in the SQLCA structure, as described in "The SQL Communication Area" on page 228.

Example

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

```
EXEC SQL BEGIN DECLARE SECTION;
   long int    emp_id;
   char        name[41];
   char        sex;
   char        birthdate[15];
   short int   ind_birthdate;
EXEC SQL END DECLARE SECTION;

int find_employee( long employee )
{
   emp_id = employee;
```

```
    EXEC SQL  SELECT emp_fname || ' ' || emp_lname,
          sex, birth_date
       INTO :name, :sex, birthdate:ind_birthdate
       FROM "DBA".employee
       WHERE emp_id = :emp_id;

    if( SQLCODE == SQLE_NOTFOUND ) {
       return( 0 ); /* employee not found */
    } else if( SQLCODE < 0 ) {
       return( -1 ); /* error */
    } else {
       return( 1 ); /* found */
    }
}
```

Fetching multiple rows

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

For an introduction to cursors, see "Working with cursors" on page 19 of the book *ASA Programming Guide*.

❖ To manage a cursor in embedded SQL

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

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

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

The following is a simple example of cursor usage:

```
void print_employees( void )
{
    int status;
```

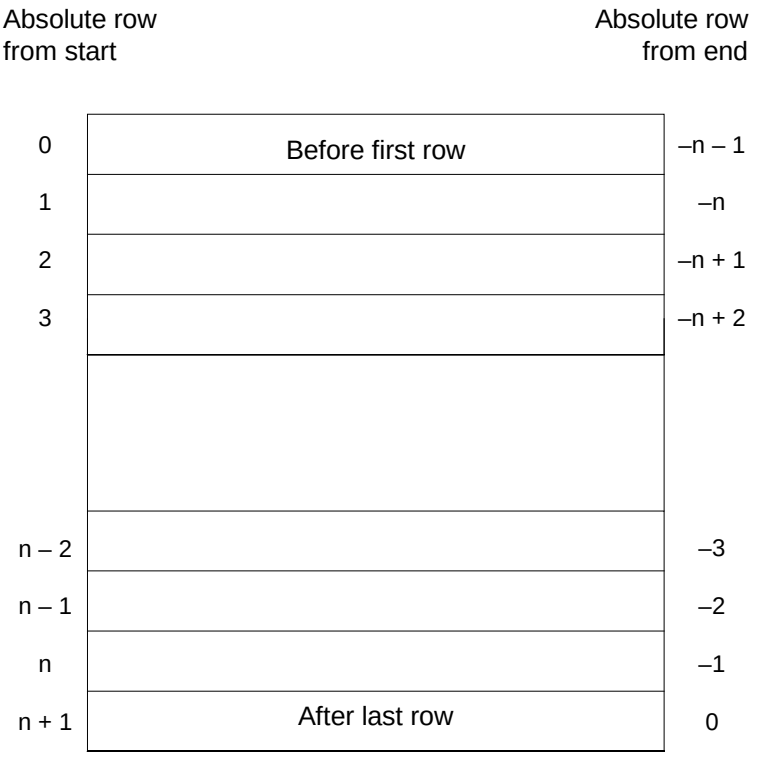
```
EXEC SQL BEGIN DECLARE SECTION;
char name[50];
char sex;
char birthdate[15];
short int ind_birthdate;
EXEC SQL END DECLARE SECTION;

/* 1. Declare the cursor. */
EXEC SQL DECLARE C1 CURSOR FOR
   SELECT emp_fname || ' ' || emp_lname,
             sex, birth_date
   FROM "DBA".employee
   ORDER BY emp_fname, emp_lname;

/* 2. Open the cursor. */
EXEC SQL OPEN C1;

/* 3. Fetch each row from the cursor. */
for( ;; ) {
   EXEC SQL FETCH C1 INTO :name, :sex,
         :birthdate:ind_birthdate;

   if( SQLCODE == SQLE_NOTFOUND ) {
      break; /* no more rows */
   } else if( SQLCODE < 0 ) {
      break; /* the FETCH caused an error */
   }

   if( ind_birthdate < 0 ) {
      strcpy( birthdate, "UNKNOWN" );
   }

   printf( "Name: %s Sex: %c Birthdate:
            %s\n",name, sex, birthdate );
}

/* 4. Close the cursor. */
EXEC SQL CLOSE C1;
}
```

For details of the FETCH statement, see "FETCH statement [ESQL] [SP]" on page 424 of the book *ASA SQL Reference Manual*.

Cursor positioning

A cursor is positioned in one of three places:

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



Order of the rows in a cursor

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

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

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

Repositioning a cursor

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

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

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

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

Cursor positioning after updates

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

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

For more information about temporary tables, see "Use of work tables in query processing" on page 160 of the book *ASA SQL User's Guide*.

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

Optimizing query operation

Although some aspects of UltraLite applications are optimized automatically, you can improve the performance of your 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.

The SQL Communication Area

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

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

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

SQLCA provides error codes

You reference the SQLCA to test for a particular error code. The **sqlcode** field contains an error code when a database request causes an error (see below). Some C macros are defined for referencing the **sqlcode** field and some other fields.

SQLCA fields

The fields in the SQLCA have the following meanings:

- **sqlcaid** An 8-byte character field that contains the string **SQLCA** as an identification of the SQLCA structure. This field helps in debugging when you are looking at memory contents.
- **sqlcabc** A long integer that contains the length of the SQLCA structure (136 bytes).
- **sqlcode** A long integer that specifies the error code when the database detects an error on a request. Definitions for the error codes can be found in the header file *sqlerr.h*. The error code is 0 (zero) for a successful operation, positive for a warning and negative for an error.

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

  For a list of error codes, see "Database Error Messages" on page 1 of the book *ASA Errors Manual*.
- **sqlerrml** The length of the information in the **sqlerrmc** field.

  UltraLite applications do not use this field.

- **sqlerrmc** May contain one or more character strings to be inserted into an error message. Some error messages contain a placeholder string (*%1*) which is replaced with the text in this field.

  UltraLite applications do not use this field.
- **sqlerrp** Reserved.
- **sqlerrd** A utility array of long integers.
- **sqlwarn** Reserved.

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

  UltraLite applications do not use this field.

SQLCA management for multi-threaded or reentrant code

UltraLite applications should not be created where multiple threads access the database simultaneously, as multi-threading is not supported.

Using multiple SQLCAs

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

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

   For more information, see "db_init function" on page 231.
2. The embedded SQL statement SET SQLCA is used to tell the SQL preprocessor to use a different SQLCA for database requests. Usually, a statement such as the following:

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

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

   For information about creating SQLCAs, see "SET SQLCA statement [ESQL]" on page 545 of the book *ASA SQL Reference Manual*.

Connection management with multiple SQLCAs

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

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

For more information, see "SET CONNECTION statement [Interactive SQL] [ESQL]" on page 536 of the book *ASA SQL Reference Manual*.

Library function reference

The SQL preprocessor generates calls to functions in the runtime library or DLL. In addition to the calls generated by the SQL preprocessor, several routines are provided for the user to make database operations easier to perform. Prototypes for these functions are included by the EXEC SQL INCLUDE SQLCA command.

db_fini function

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

Description Frees resources used by the UltraLite runtime library.

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

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

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

See also "db_init function" on page 231

db_init function

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

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

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

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

On the Palm Computing Platform, call **db_init** whenever **ULPalmLaunch** returns LAUNCH_SUCCESS_FIRST. For more information, see "ULPalmLaunch function" on page 245.

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

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

In most cases, this function should be called only once (passing the address of the global **sqlca** variable defined in the *sqlca.h* header file). If you have multiple execution paths in your application, you can use more than one **db_init** call, as long as each one has a separate **sqlca** pointer. This separate SQLCA pointer can be a user-defined one, or could be a global SQLCA that has been freed using **db_fini**.

In multi-threaded applications, each thread must call **db_init** to obtain a separate SQLCA. Subsequent connections and transactions that use this SQLCA must be carried out on a single thread.

See also

"db_fini function" on page 231
"ULPalmLaunch function" on page 245
"Developing multi-threaded applications" on page 93

ULActiveSyncStream function

Prototype

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

Description

Defines an ActiveSync stream suitable for synchronization.

The ActiveSync stream is available only on Windows CE devices.

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

See also

"ULIsSynchronizeMessage function" on page 243
"ULSynchronize function" on page 250
"Synchronize method" on page 143
"ActiveSync parameters" on page 399

ULChangeEncryptionKey function

Prototype ul_bool **ULChangeEncryptionKey(** SQLCA **sqlca*, ul_char **new_key* **);**

Description Changes the encryption key for an UltraLite database.

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

See also "Changing the encryption key for a database" on page 49

ULClearEncryptionKey function

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

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

This function clears the encryption key.

Parameters

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

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

See also
"ULRetrieveEncryptionKey function" on page 247
"ULSaveEncryptionKey function" on page 248
"Using the encryption key on the Palm Computing Platform" on page 50

ULConduitStream function (deprecated)

Prototype ul_stream_defn **ULConduitStream(** void **);**

Description Defines a stream under the Palm Computing Platform suitable for HotSync synchronization.

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

See also

"ULPalmDBStream function (deprecated)" on page 243
"ULPalmExit function" on page 244
"ULPalmLaunch function" on page 245
"HotSync and ScoutSync parameters" on page 401
"Synchronize method" on page 143

ULCountUploadRows function

Prototype

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

Description

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

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

Parameters

sqlca A pointer to the SQLCA.

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

```
UL_PUB_PUB1 | UL_PUB_PUB2
```

For more information on publication masks, see "publication synchronization parameter" on page 386.

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

Example

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

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

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

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

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

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

ULDropDatabase function

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

Description Delete the UltraLite database file.

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

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

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

Parameters

sqlca A pointer to the SQLCA.

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

For more information, see "UL_STORE_PARMS macro" on page 428.

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

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

ULEnableFileDB function

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

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

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

Parameters

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

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

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

See also "ULEnablePalmRecordDB function" on page 237

ULEnableGenericSchema function

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

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

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

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

The schema upgrading process uses matching names in the old and new schema. It 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.

In general, adding constraints to tables that have data in them or carrying out unpredictable column conversions may result in lost rows.

Parameters

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

See also

"Deploying UltraLite applications" on page 104

ULEnablePalmRecordDB function

Prototype

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

Description

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

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

Parameters

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

Examples

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

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

See also "ULEnableFileDB function" on page 235

ULEnableStrongEncryption function

Prototype void **ULEnableStrongEncryption(** SQLCA * *sqlca* **)**

Description Strongly encrypt an UltraLite database.

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

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

See also "Encrypting UltraLite databases" on page 45
"Changing the encryption key for a database" on page 49

ULEnableUserAuthentication function

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

Description Enable user authentication in the UltraLite application.

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

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

See also "User authentication for UltraLite databases" on page 442
"Adding user authentication to your application" on page 85

ULGetLastDownloadTime function

Prototype

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

Description

Obtains the last time a specified publication was downloaded.

Parameters

sqlca A pointer to the SQLCA.

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

```
UL_PUB_PUB1 | UL_PUB_PUB2
```

For more information on publication masks, see "publication synchronization parameter" on page 386.

value A pointer to the DECL_DATETIME structure to be populated.

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

Returns

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

Examples

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

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

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

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

See also

"publication synchronization parameter" on page 386
"UL_SYNC_ALL macro" on page 431
"UL_SYNC_ALL_PUBS macro" on page 431

ULGetSynchResult function

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

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

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

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

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

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

where the individual members have the following meanings:

- **sql_code** The SQL code from the last synchronization. For a list of SQL codes, see "Error messages indexed by Adaptive Server Anywhere SQLCODE" on page 2 of the book *ASA Errors Manual*.
- **stream_error** The communication stream error code from the last synchronization. For a listing, see "MobiLink Communication Error Messages" on page 631 of the book *MobiLink Synchronization User's Guide*.
- **upload_ok** Set to **true** if the upload was successful; **false** otherwise.
- **ignored_rows** Set to **true** if uploaded rows were ignored; **false** otherwise.

- **auth_status** The synchronization authentication status. For more information, see "auth_status synchronization parameter" on page 381.
- **auth_value** The value used by the MobiLink synchronization server to determine the **auth_status** result. For more information, see "auth_value synchronization parameter" on page 382.
- **timestamp** The time and date of the last synchronization.
- **status** The status information used by the observer function. For more information, see "observer synchronization parameter" on page 384.

Returns The function returns a Boolean value.

true Success.

false Failure.

Examples The following code checks for success of the previous synchronization.

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

See also "ULPalmLaunch function" on page 245

ULGlobalAutoincUsage function

Prototype short **ULGlobalAutoincUsage(** SQLCA * *sqlca* **);**

Description Obtains the percent of the default values used in all the columns having global autoincrement defaults. If the database contains more than one column with this default, this value is calculated for all columns and the maximum is returned. For example, a return value of 99 indicates that very few default values remain for at least one of the columns.

Returns The function returns a value of type short in the range 0–100.

See also "ULSetDatabaseID function" on page 248

ULGrantConnectTo function

Prototype

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

Description

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

Parameters

sqlca A pointer to the SQLCA.

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

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

See also

"User authentication for UltraLite databases" on page 442
"Adding user authentication to your application" on page 85
"ULRevokeConnectFrom function" on page 248

ULHTTPSStream function

Prototype

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

Description

Defines an UltraLite HTTPS stream suitable for synchronization via HTTP.

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

See also

"ULSynchronize function" on page 250
"Synchronize method" on page 143
"stream synchronization parameter" on page 389
"HTTPS stream parameters" on page 406

ULHTTPStream function

Prototype

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

Description

Defines an UltraLite HTTP stream suitable for synchronization via HTTP.

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

See also

"ULSynchronize function" on page 250
"Synchronize method" on page 143
"stream synchronization parameter" on page 389
"HTTP stream parameters" on page 403

ULIsSynchronizeMessage function

Prototype

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

Description

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

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

Example

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

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

  switch( uMsg ) {

  // code to handle other windows messages

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

See also

"Adding ActiveSync synchronization to your application" on page 305

ULPalmDBStream function (deprecated)

Prototype

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

Description

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

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

See also

"ULPalmExit function" on page 244
"ULPalmLaunch function" on page 245
"HotSync and ScoutSync parameters" on page 401
"Synchronize method" on page 143

ULPalmExit function

Prototype

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

Description

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

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

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

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

Parameters

sqlca A pointer to the SQLCA.

synch_info A synchronization structure.

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

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

For information on the members of the *synch_info* structure, see "Synchronization stream parameters" on page 399.

Returns The function returns a Boolean value.

true Success.

false Failure.

ULPalmLaunch function

Prototype

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

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

Parameters

sqlca A pointer to the SQLCA.

synch_info A synchronization structure. For information on the members of this structure, see "Synchronization parameters" on page 380.

If you are using TCP/IP or HTTP synchronization, supply UL_NULL as *synch_info*.

Description This function restores application state for UltraLite applications on the Palm Computing Platform. This function is required by all UltraLite Palm applications.

Your application must call **ULEnablePalmDB** or **ULEnableFileDB** before calling **ULPalmLaunch**.

All UltraLite Palm applications need to use this function to handle the launch code in your application's **PilotMain**.

This function restores the application state when the application is activated. For applications using HotSync or Scout Sync synchronization, it carries out the additional task of processing the download stream prepared by the MobiLink HotSync conduit or MobiLink Scout conduit.

If you are using TCP/IP or HTTP synchronization, supply a null value for the stream parameter in the **ul_synch_info** synchronization structure. This information is supplied instead in the call to **ULSynchronize**.

Returns A member of the **UL_PALM_LAUNCH_RET** enumeration. The return values have the following meanings:

- **LAUNCH_SUCCESS_FIRST** This value is returned the first time the application is successfully launched and at any subsequent time the internal state of the UltraLite database needs to be re-established. In general, the state of the database needs to be re-established only after severe failures.

  In embedded SQL applications you should call **db_init** immediately after this return code is detected; in C++ API applications, you should open a database object.
- **LAUNCH_SUCCESS** This value is returned when an application is successfully launched, after the Palm user has been using other applications.
- **LAUNCH_FAIL** This value is returned when the launch fails.

Examples

A typical embedded SQL example is

```
ULEnablePalmRecordDB( &sqlca );
switch( ULPalmLaunch( &sqlca, &synch_info ) ){
case LAUNCH_SUCCESS_FIRST:
   if( !db_init( &sqlca ) ){
      // db_init failed: add error handling here
      break;
   }
   // fall through
case LAUNCH_SUCCESS:
   // do work here
   break;
case LAUNCH_FAIL:
   // error
   break;
}
```

See also

"Launching an UltraLite Palm application" on page 261
"ULEnableFileDB function" on page 235
"ULEnablePalmRecordDB function" on page 237

ULResetLastDownloadTime function

Prototype

void **ULResetLastDownloadTime(**
SQLCA * *sqlca*,
ul_publication_mask *publication-mask* **);**

Description

This function can be used to repopulate values and return an application to a known clean state. It resets the last download time so that the application resynchronizes previously downloaded data.

Parameters

sqlca A pointer to the SQLCA.

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

```
UL_PUB_PUB1 | UL_PUB_PUB2
```

For more information on publication masks, see "publication synchronization parameter" on page 386.

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

```
ULResetLastDownloadTime( &sqlca, UL_SYNC_ALL );
```

See also
"ULGetLastDownloadTime function" on page 239
"Timestamp-based synchronization" on page 86 of the book *MobiLink Synchronization User's Guide*

ULRetrieveEncryptionKey function

Prototype

ul_bool **ULRetrieveEncryptionKey(**
ul_char * *key*,
ul_u_short *len*,
ul_u_long * *creator*,
ul_u_long * *feature-num* **);**

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

This function retrieves the encryption key from memory.

Parameters

key A pointer to a buffer in which to hold the retrieved encryption key.

len The length of the buffer that holds the encryption key with a terminating null character.

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

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

Returns

- **true** if the operation is successful.
- **false** if the operation is unsuccessful. This occurs if the feature was not found or if the supplied buffer length is insufficient to hold the key plus a terminating null character.

See also
"ULClearEncryptionKey function" on page 233
"ULSaveEncryptionKey function" on page 248
"Using the encryption key on the Palm Computing Platform" on page 50

ULRevokeConnectFrom function

Prototype void **ULRevokeConnectFrom(** SQLCA * *sqlca***,** ul_char * *userid* **);**

Description Revoke access from an UltraLite database for a user ID.

Parameters

sqlca A pointer to the SQLCA.

userid Character array holding the user ID to be excluded from database access. The maximum length is 16 characters.

See also

"User authentication for UltraLite databases" on page 442
"Adding user authentication to your application" on page 85
"ULGrantConnectTo function" on page 242

ULSaveEncryptionKey function

Prototype

ul_bool **ULSaveEncryptionKey(**
ul_char * *key*,
ul_u_long * *creator***,**
ul_u_long * *feature-num* **);**

Description On the Palm Computing Platform the encryption key is saved in dynamic memory as a Palm **feature**. Features are indexed by creator and a feature number. They are not backed up and are cleared on any reset of the device.

This function saves the encryption key in Palm dynamic memory.

Parameters

key A pointer to the encryption key.

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

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

Returns

- **true** if the operation is successful.
- **false** if the operation is unsuccessful.

See also

"ULClearEncryptionKey function" on page 233
"ULRetrieveEncryptionKey function" on page 247
"Using the encryption key on the Palm Computing Platform" on page 50

ULSetDatabaseID function

Prototype void **ULSetDatabaseID(** SQLCA * *sqlca***,** ul_u_long *id* **);**

Description Sets the database identification number.

Parameters

sqlca A pointer to the SQLCA.

id A positive integer that uniquely identifies a particular database in a replication or synchronization setup.

See also "ULGlobalAutoincUsage function" on page 241

ULSocketStream function

Prototype

ul_stream_defn **ULSocketStream(** void **);**

Description Defines an UltraLite socket stream suitable for synchronization via TCP/IP.

See also

"ULSynchronize function" on page 250
"Synchronize method" on page 143

ULStoreDefragFini function

Prototype

ul_ret_void **ULStoreDefragFini(**
SQLCA * *sqlca*,
p_ul_store_defrag_info *dfi* **);**

Description This function disposes of the defragmentation information block returned by **ULStoreDefragInit**.

Parameters

sqlca A pointer to the SQLCA.

dfi A defragmentation information block.

See also

"Defragmenting UltraLite databases" on page 51
"ULStoreDefragInit function" on page 249

ULStoreDefragInit function

Prototype

p_ul_store_defrag_info **ULStoreDefragInit(** SQLCA * *sqlca* **);**

Description This function initializes and returns a defragmentation information block to maintain the defragmentation state of the database.

Parameters

sqlca A pointer to the SQLCA.

Returns If successful, returns a defragmentation information block **p_ul_store_defrag_info**. If unsuccessful, for example if there is not enough memory, returns **UL_NULL**.

See also "Defragmenting UltraLite databases" on page 51
"ULStoreDefragFini function" on page 249

ULStoreDefragStep function

Prototype

```
ul_bool ULStoreDefragStep(
    SQLCA * sqlca
    p_ul_store_defrag_info dfi );
```

Description This function defragments a piece of the database.

Parameters

sqlca A pointer to the SQLCA.

dfi A defragmentation information block.

Returns If the entire store has been defragmented, returns **ul_true**.

If the entire store is not defragmented, returns **ul_false**.

If an error occurs, SQLCODE is set.

See also "Defragmenting UltraLite databases" on page 51
"ULStoreDefragFini function" on page 249
"ULStoreDefragInit function" on page 249

ULSynchronize function

Prototype

```
void ULSynchronize(
    SQLCA * sqlca,
    ul_synch_info * synch_info );
```

Description Initiates synchronization in an UltraLite application.

For TCP/IP or HTTP synchronization, the **ULSynchronize** function initiates synchronization. Errors during synchronization that are not handled by the **handle_error** script are reported as SQL errors. Your application should test the SQLCODE return value of this function.

Parameters

sqlca A pointer to the SQLCA.

synch_info A synchronization structure. For information on the members of this structure, see "Synchronization parameters" on page 380.

See also

"MobiLink Synchronization Server Options" on page 379 of the book *MobiLink Synchronization User's Guide*

"START SYNCHRONIZATION DELETE statement" on page 583 of the book *MobiLink Synchronization User's Guide*

CHAPTER 11

Developing Applications for the Palm Computing Platform

About this chapter

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

Contents

Introduction

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

Development environments

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

- Metrowerks CodeWarrior, version 6, 7, or 8.

  See "Developing UltraLite applications with Metrowerks CodeWarrior" on page 255.

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

- GCC PRC Tools. This set of tools is based on the GNU compiler.

  See "Developing UltraLite applications with GCC PRC-Tools" on page 259.

For more information on target platforms, see "Supported platforms for C/C++ applications" on page 6.

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

Target platforms

For a list of supported target operating systems, see "Supported platforms for C/C++ applications" on page 6.

Palm-specific notes

The information in this chapter concerning Palm development supplements the general information on UltraLite development provided "Developing UltraLite Applications" on page 67.

Developing UltraLite applications with Metrowerks CodeWarrior

Metrowerks CodeWarrior versions 6 and 7 are supported host platforms for Palm Computing Platform UltraLite development.

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

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

Installing the UltraLite plug-in for CodeWarrior

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

❖ **To install the UltraLite plug-in for CodeWarrior:**

1. Ensure that you are running CodeWarrior version 6 or CodeWarrior version 7. You can obtain patches for CodeWarrior from the Metrowerks Web site.
2. From a command prompt, change to the *UltraLite\palm\68k\cwplugin* subdirectory of your SQL Anywhere directory.
3. Run *install.bat* to copy the appropriate files into your CodeWarrior installation directory: The *install.bat* file takes two arguments:
   - Your CodeWarrior directory
   - Your CodeWarrior version. Version 6 is the default.

   For example, the following command (which should be entered on one line) installs the plug-in for CodeWarrior 7 in the default CodeWarrior installation directory.

   ```
   install "c:\Program Files\Metrowerks\CodeWarrior for
   Palm OS Platform 7.0" r7
   ```

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

Uninstalling the CodeWarrior plug-in

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

Creating UltraLite projects in CodeWarrior

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

❖ **To create an UltraLite project in CodeWarrior:**

1 Start CodeWarrior.

2 Create a new project.

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

On the Projects dialog, choose one of the available choices, and choose a name and location for the project. Click OK.

3 Choose an UltraLite stationery.

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

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

This stationery is standard C stationery for embedded SQL, and standard C++ stationery for the C++ API, and contains almost-empty source files.

4 Configure the target settings for your project.

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

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

When you build your project by pressing F7, the following preprocessing steps are carried out:

- For embedded SQL applications, *sqlpp* and *ulgen* utilities are invoked automatically to convert any *.sqc* files into *.c* or *.cpp* files and to generate the database code.
- For C++ API applications, *ulgen* is invoked to generate the UltraLite API files and the database code.

Also, the paths to required UltraLite files, such as headers and runtime library, are automatically added to the search paths.

Converting an existing CodeWarrior project to an UltraLite application

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

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

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

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

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

Using the UltraLite plug-in for CodeWarrior

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

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

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

For an overview of the tasks the plug-in carries out, see "Configuring development tools for UltraLite development" on page 102.

Using prefix files

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

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

Building the CustDB sample application from CodeWarrior

CustDB is a simple sales-status application.

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

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

- **cwcommon** Files common to both CodeWarrior 6 and CodeWarrior 7.
- **cw6** Files for CodeWarrior 6.
- **cw7** Files for CodeWarrior 7.
- **cw8** Files for CodeWarrior 8.

The instructions in this section describe how to build the CustDB application using CodeWarrior 7. The process is very similar for CodeWarrior 6.

❖ **To build the CustDB sample application using CodeWarrior:**

1. Start the CodeWarrior IDE.
2. Open the CustDB project file:
   - Choose File➤Open.
   - Open the project file *Samples\UltraLite\custdb\cw8\custdb.mcp* under your SQL Anywhere directory.
3. To build the target application (*custdb.prc*), choose Project➤Make.

You can use the UltraLite plug-in to customize settings for your own application. For more information, see "Developing UltraLite applications with Metrowerks CodeWarrior" on page 255.

Developing UltraLite applications with GCC PRC-Tools

You can use the GNU GCC PRC-Tools suite for the Palm Computing Platform to develop UltraLite applications. This section assumes that you are familiar with the installation and use of the GCC PRC-Tools, and provides UltraLite-specific tips and information.

☞ For information on the GCC PRC-Tools suite, see http://www.palmos.com/dev/tech/tools/gcc/.

The UltraLite runtime library for PRC-Tools is located in subdirectories of your SQL Anywhere directory:

```
UltraLite\Palm\68k\lib\prctools20\libulrt.a
```

No transport-layer security using GCC tools
The Certicom transport-layer security is not available when using GCC tools.

Compiler issues

Compile UltraLite applications using the `-DM68000 -mnoshort` switches. The first option declares a symbol required to properly set the SQL_OS macro. The second option forces **int** data types to be four bytes (two is default).

Building the CustDB sample application with PRC Tools

CustDB is a simple sales-status application. It is located in the UltraLite *Samples* directory of your installation. Generic files are located in the *CustDB* directory. Files specific to PRC Tools for the Palm Computing Platform are located in the *prctools20* subdirectory of *CustDB*.

You must have the full set of PRC Tools installed before building the CustDB sample application, including the following:

- Cygnus **cygwin** tools. The directory containing *cygwin1.dll* must be in your path.
- GCC Tool chain for the Palm OS (PRC Tools 2.0). The *multigen.exe* utility must be in your path.
- Palm SDK 3.5.
- PilRC resource compiler tools. The *pilrc.exe* utility must be in your path.

❖ **To build the sample application:**

1. Open a command prompt window.
2. Change directory to the *Samples\UltraLite\CustDB\prctools20* subdirectory of your SQL Anywhere directory.
3. Entering the following command:

   ```
   build
   ```

   The *build.bat* file contains instructions to build the UltraLite sample application.

Once you have built the sample application, you can use the Palm Desktop software deploy the *custdb.prc* executable to your target device.

Launching and closing UltraLite applications

Palm OS applications are single threaded. To maintain the illusion that an application is running in the background after you close it, the application must save its internal state when the user switches to another application. When the application is **launched** again, it must restore its internal state.

This section describes how to handle launching and closing of an UltraLite Palm application.

Two Palm-specific UltraLite functions save and restore internal state information, and must be used by all UltraLite applications for the Palm Computing Platform. These functions also handle synchronization if you are using the HotSync or ScoutSync synchronization streams, but not if you are using TCP/IP or HTTP streams.

Launching an UltraLite Palm application

Whenever your UltraLite application is launched, your code must call the function to restore state.

For embedded SQL development, this function is **ULPalmLaunch**. For C++ API development, this function is the **ULData.PalmLaunch()** method.

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

For more information, see "ULPalmLaunch function" on page 245, and "PalmLaunch method" on page 148.

Closing an UltraLite Palm application

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

For embedded SQL development, this function is **ULPalmExit**. For C++ API development, this function is the **ULData.PalmExit()** method.

C++ API

For C++ API developers, the following considerations also apply:

- Do *not* close any **ULData** or **ULConnection** objects.

- When the user returns to the application, call **Reopen**, first on the **ULData** and then on the **ULConnection** object.
- For cursor objects, including instances of generated result set classes, you can do either of the following:
  - Ensure that the object is closed when the user switches away from the application, and call **Open** when you next need the object. If you choose this option, the current position is not restored.
  - Do not close the object when the user switches away, and call **Reopen** when you next need to access the object. The current position is then maintained, but the application takes more memory in the Palm when the user is using other applications.
- For table objects, including instances of generated table classes, you cannot save a position. You must close table objects before a user moves away from the application, and **Open** them when the user needs them again. Do not use Reopen on table objects.

Embedded SQL

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

☞ For more information, see "ULPalmExit function" on page 244, and "PalmExit method" on page 147.

Building multi-segment applications

Application code for the Palm Computing Platform must be divided into **segments**. For CodeWarrior, these segments are at most 64 kb in size. For PRC Tools, they are at most 32 kb. This section describes how to manage the assignment of code into segments.

UltraLite applications include the following types of code:

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

Building multi-segment applications is a general feature of application development for the Palm Computing Platform, whether or not you are using UltraLite. Some familiarity with building multi-segment applications using your development tool is assumed. User-defined code is no different to other standard Palm applications. For a reminder about assigning user-defined code to segments, see "Assigning user-defined code to segments" on page 266.

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

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

  For more information, see "Enabling multi-segment code generation" on page 264.

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

  For more information, see "Explicitly assigning segments" on page 265.

Enabling multi-segment code generation

This section describes how to instruct the UltraLite Analyzer to generate multi-segment code using its default scheme. If you wish to customize the assignment of code to segments by explicitly assigning functions to segments, you can do so. For more information, see "Explicitly assigning segments" on page 265.

You enable generated code segments by defining macros. Macro definition is different for the CodeWarrior and PRC Tools development tools, so the procedure for enabling multi-segment code generation also differs.

❖ **To enable multi-segment code generation (CodeWarrior):**

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

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

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

   ☞ For more information, see "UL_ENABLE_SEGMENTS macro" on page 428.

❖ **To enable multi-segment code generation (PRC Tools):**

1. Instruct the gnu compiler to compile segmented code.

   Define the following two macros on the compiler command line:

   - UL_ENABLE_SEGMENTS
   - UL_ENABLE_GNU_SEGMENTS

   ☞ For an example, see the file *Samples\UltraLite\CustDB\PRCTools20\ build.bat* relative to your SQL Anywhere directory.

   ☞ For more information, see "UL_ENABLE_SEGMENTS macro" on page 428, and ."UL_ENABLE_GNU_SEGMENTS macro" on page 428.

2. Construct a segment definition file for the GNU link tools *multilink* and *build-prc*.

   Run the *dbulseg* command-line utility against each source file, and supply a name for the definition file. For example, the following command line:

```
dbulseg gensource.c project.def AppName CreatorID
```

   creates a *project.def* definition file with the following content:

```
application{ "AppName" CreatorID }
multiple code{ ULRT1 … ULRT17 ULG512 ULG513 … }
```

where the ULG segment names are obtained from the generated source file *gensource.c*.

For more information on the UltraLite segment utility, see "The UltraLite segment utility" on page 425.

Notes

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

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

PRC Tools compiler issues

When a function defined at the bottom of a source file makes an inter-segment call to a function defined at the top of the same source file, and there is more than 32 kb of code in between, the PRC Tools compiler may generate **jsr** instructions unacceptable to the assembler. Normally, the offset of the **jsr** instruction is replaced during the relocation stage of the linker, but in this case, the error prevents the compilation from going any further. To avoid this issue, instruct the assembler to ignore any signed overflow errors by using the `-Wa,-J` compiler switches.

Explicitly assigning segments

This section describes how to explicitly assign the generated code for SQL statements to segments. You must first enable multi-segment code generation as described in "Enabling multi-segment code generation" on page 264.

The mechanism for assigning the code is different for the embedded SQL and C++ API development models.

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

❖ **To explicitly assign generated statement code to segments (embedded SQL):**

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

❖ **To explicitly assign generated statement code to segments (C++ API):**

- Do one of the following:
  - Call the **ul_set_codesegment** procedure for each SQL statement, providing the name of the segment to which the statement should be assigned.

    For example, the following statement assigns the statement **mystmt**, in the project **myproject**, to the segment **MYSEG1**.

    ```
    call ul_set_codesegment(
         'myproject', 'mystmt', 'MYSEG1' )
    ```

    For more information, see "ul_set_codesegment system procedure" on page 413.
  - From Sybase Central, open the UltraLite Project folder. Right click the statement and choose Properties from the popup menu. Enter a code segment name in the box.

Assigning user-defined code to segments

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

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

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

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

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

❖ **To assign user-defined code to segments (PRC Tools):**

- Add the following declaration to each function:

  ```
  __attribute__( section( "segment-name" ) )
  ```

The first segment

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

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

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

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

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

Palm synchronization overview

UltraLite applications running on the Palm Computing Platform can synchronize using the following streams:

- **TCP/IP** Through the cradle or through a modem.
- **HTTP** Through the cradle or through a modem.
- **HotSync** The Palm Computing Platform built-in synchronization method.
- **ScoutSync** The synchronization method from Aether Systems. ScoutSync cannot be used with UltraLite databases stored on Palm OS 4 expansion cards.

  ScoutSync support is deprecated. Version 8.0.x will continue to support ScoutSync up to version 3.6, but the next major release of SQL Anywhere Studio will not support ScoutSync.

  For more information regarding ScoutSync, see http://www.aethersystems.com.

Choosing a synchronization method

Each synchronization method has its advantages and disadvantages.

- **Multiple applications** If you have more than one UltraLite application installed on a Palm device, they all synchronize when you invoke HotSync or ScoutSync. To synchronize multiple applications through a TCP/IP or HTTP connection, you must activate and synchronize each application in turn.
- **Universal Serial Bus support** HotSync synchronization has automatic support for USB.
- **Publications** Synchronization using HotSync or ScoutSync cannot include WHERE clauses.

  For more information, see "Designing sets of data to synchronize separately" on page 76.

Understanding HotSync and ScoutSync synchronization

UltraLite applications on Palm devices can synchronize over a TCP/IP or HTTP stream, in much the same manner as UltraLite applications on other platforms. They can also synchronize using the Palm-specific HotSync or ScoutSync synchronization streams, which operate in a different manner. This section describes the architecture of the HotSync and ScoutSync synchronization.

The sequence of events that occur during HotSync and ScoutSync synchronization is as follows:

1. When your UltraLite application is closed, it saves the state of your UltraLite application. The state information is stored in the Palm database, separately from the UltraLite database.

   For more information, see "Closing an UltraLite Palm application" on page 261.

2. When you synchronize your Palm device, HotSync or ScoutSync calls the MobiLink conduit to synchronize with the MobiLink synchronization server. The MobiLink conduit reads the pages from the UltraLite database and sends the upload to the MobiLink synchronization server.

3. The MobiLink synchronization server integrates updates into the consolidated database and sends a download stream to the conduit.

4. The conduit integrates the download stream into the UltraLite database on the Palm device.

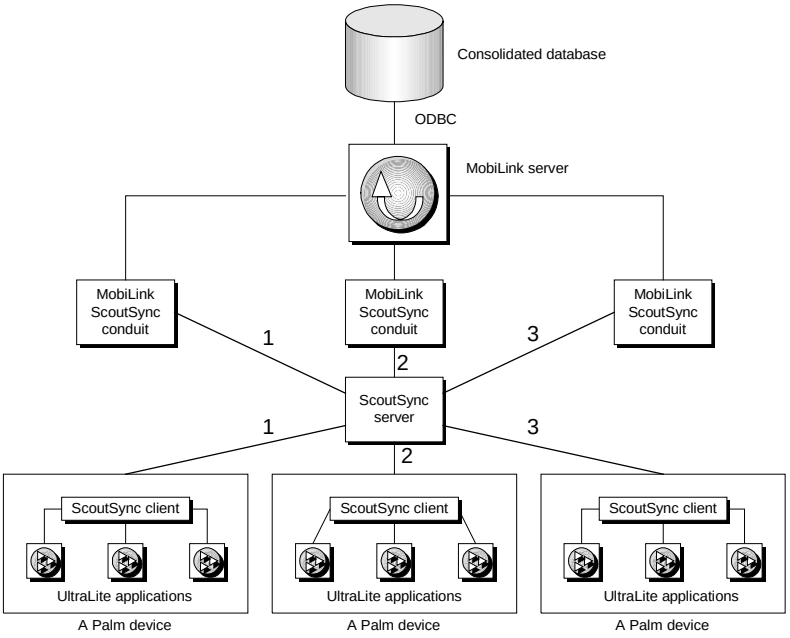
5. When your application is launched, it loads the previously saved state of your UltraLite application.

   For more information, see "Launching an UltraLite Palm application" on page 261.

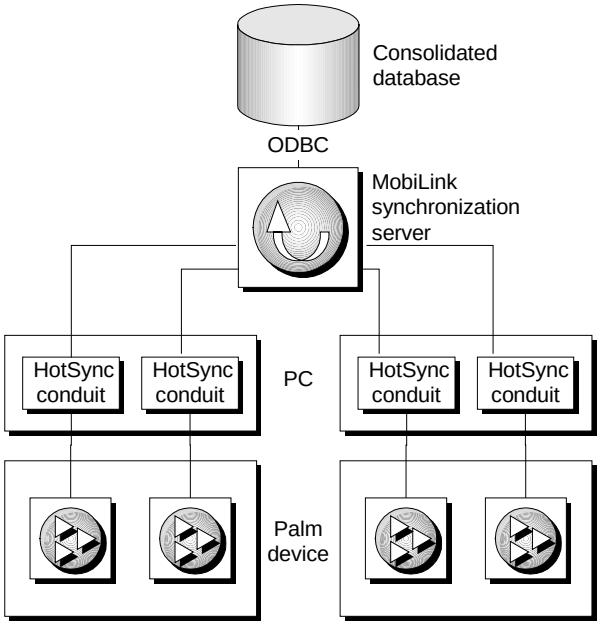
HotSync and ScoutSync architecture

The application code for HotSync and ScoutSync synchronization is identical. The synchronization architecture is different, however.

The following diagram depicts the ScoutSync architecture. A separate instance of the conduit is instantiated by the ScoutSync server for each Palm device.



The following diagram depicts the HotSync architecture. In this case, a separate HotSync conduit is required for each application (as opposed to each device for ScoutSync). You can have multiple HotSync conduits on a single PC.



Adding HotSync or ScoutSync synchronization to Palm applications

This section describes what you need to include in your UltraLite application code to synchronize using HotSync or ScoutSync. From the UltraLite application side, the procedure is very similar for these two synchronization streams.

See also

For an overview of HotSync and ScoutSync, see "Understanding HotSync and ScoutSync synchronization" on page 269.

For information on configuring HotSync and ScoutSync, see "Configuring HotSync synchronization" on page 274, and "Configuring ScoutSync synchronization" on page 279.

Synchronization functions

If you use HotSync or ScoutSync, then you synchronize by calling **ULPalmLaunch** (embedded SQL) or **ULData.PalmLaunch** (C++ API) when your application is launched, and **ULPalmExit** (embedded SQL) or **ULData.PalmExit** (C++ API) when your application is closed. You must supply a **ul_synch_info** structure holding the synchronization parameters to **ULPalmExit** or **ULData.PalmExit**. The **stream** parameter for the **ul_synch_info** structure is ignored, and can be UL_NULL.

Do not use **ULSynchronize** or **ULConnection.Synchronize** for HotSync or ScoutSync synchronization.

For more information, see "Launching and closing UltraLite applications" on page 261, and "Synchronization parameters" on page 380.

If there are uncommitted transactions when you close your Palm application, and if you synchronize, the conduit reports that synchronization fails because of uncommitted changes in the database.

Specifying the stream parameters

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

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

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

  For a list of available values, see "Synchronization stream parameters" on page 399.

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

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

  For more information, see "HotSync configuration overview" on page 274.

Adding HotSync or ScoutSync synchronization to your application

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

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

   This function is called when the user switches away from the UltraLite application. You must ensure that all outstanding operations are committed before calling the **ULPalmExit** or **ULData.PalmExit** function.

   The **ul_synch_info.stream** parameter is ignored, and so does not need to be set.
3. Call the **ULPalmLaunch** or **ULData.PalmLaunch** function

❖ To add HotSync or ScoutSync synchronization code to your application:

- In the source code for your UltraLite application call **ULPalmExit()** or **ULData.PalmExit** with parameters such as the following:

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

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

  For more information, see "Adding HotSync or ScoutSync synchronization to Palm applications" on page 272.

Configuring HotSync synchronization

This section describes how to set up your MobiLink HotSync conduit which is required for HotSync synchronization of UltraLite applications.

For an overview of HotSync synchronization, see "HotSync and ScoutSync architecture" on page 269.

HotSync configuration overview

During HotSync synchronization, the HotSync Manager starts the MobiLink HotSync conduit, *dbhsync8.dll*, which sends the upload stream to a MobiLink synchronization server, and receives the download stream from the MobiLink synchronization server.

The MobiLink HotSync conduit synchronizes with the MobiLink synchronization server using one of TCP/IP, HTTP, or HTTPS streams.

Depending on the demands of your installation, you may deploy only the MobiLink HotSync conduit onto the desktop machines of your users.

For information on HotSync architecture, see "HotSync and ScoutSync architecture" on page 269.

❖ **To install and configure the MobiLink HotSync conduit:**

1. Place the MobiLink conduit files on the user's machine.

   For instructions, see "HotSync conduit files" on page 275.

2. Add the MobiLink conduit to the HotSync Manager. The HotSync manager is then able to use the MobiLink conduit.

   For instructions, see "Adding the MobiLink conduit into HotSync manager" on page 275.

3. If you did not include a **stream_parms** parameter in your UltraLite **ul_synch_info** structure, enter these parameters from the HotSync manager.

   For instructions, see "Configuring conduit synchronization" on page 277.

   For information on including **stream_parms** parameter in your UltraLite synchronization call, see "Adding HotSync or ScoutSync synchronization to Palm applications" on page 272.

4 If you are using an encrypted database, enter the encryption key in the conduit configuration dialog. If you do not enter this key, you will have to enter it on every synchronization.

For instructions, see "Configuring conduit synchronization" on page 277.

HotSync conduit files

The HotSync conduit consists of the following files:

- **dbhsync8.dll** The DLL that is called by the HotSync manager.
- **dblgen8.dll** The language resource library. For languages other than English, the file has the letters en replaced by a two-letter abbreviation for the language, such as *dblgde8.dll* or *dblgja8.dll*.
- **Stream dll** You need a DLL for the communication between the conduit and the MobiLink synchronization server. A separate DLL for each stream is provided:
  - For TCP/IP, use *dbsock8.dll*.
  - For HTTP, use *dbsock8.dll* and *dbhttp8.dll*.
  - If you use encryption for this communication, you also need to supply the encryption DLL *dbtls8.dll*.

These files should be in the same directory, in your system path. When you install SQL Anywhere Studio, they are installed into the *win32* subdirectory of your installation directory, which is already in the system path.

Adding the MobiLink conduit into HotSync manager

UltraLite includes a command-line **conduit installation utility** named *dbcond8.exe* to make a set of registry entries for the HotSync manager to be able to use the MobiLink conduit. This utility requires the following files:

- *dbcond8.exe*
- *condmgr.dll*

❖ **To deploy the conduit installation utility:**

1 Choose a top-level deployment directory.

For example, you may choose a directory named *c:\deploy*.

2 Add a registry entry with the deployment directory as its value.

The registry entry must be as follows:

```
HKEY_CURRENT_USER\Software\Sybase\Adaptive Server
Anywhere\version string\Location
```

where *version string* is **8.0** for this version of the software. If the entry is not found in *HKEY_CURRENT_USER*, the software looks in *HKEY_LOCAL_MACHINE*.

3 Add the *dbcond8.exe* file.

The *dbcond8.exe* file must go in the *win32* subdirectory of the deployment directory.

4 Add the *condmgr.dll* file.

The *condmgr.dll* file must go in the *win32\condmgr* subdirectory of the deployment directory.

The SQL Anywhere Studio installation creates the required registry entries and places files in the appropriate locations.

❖ **To add the MobiLink HotSync conduit into HotSync manager:**

1 Ensure the HotSync conduit files and the files for the conduit installation utility are in place.

2 Run the conduit installation utility, providing the creator ID of the Palm application, and a name that HotSync will use to identify the conduit. For example, the following command installs a conduit for the application with creator ID **Syb2**, named **CustDB**. These are the settings for the CustDB sample application:

```
dbcond8 "Syb2" –n CustDB
```

For full syntax of the conduit installation utility, including the options to use when uninstalling a conduit, see "The HotSync conduit installation utility" on page 414.

Checking that conduit installation is correct

You can check that a conduit is installed by right-clicking HotSync Manager in the system tray and choosing Custom from the popup menu. A list of conduits is displayed for each user. Check that your conduit is listed.

❖ **To check that the HotSync conduit is properly installed**

1 Set the environment variable UL_DEBUG_CONDUIT to any value.

2 Shut down and restart the HotSync manager.

3 If the MobiLink conduit is properly installed, two dialog boxes appear. If no dialog appears, the conduit is not properly installed.

4 Unset the environment variable.

5 Shut down and restart the HotSync manager.

MobiLink must be started before using HotSync
Before using HotSync, the MobiLink synchronization server must be started and be ready to accept connections from the MobiLink HotSync conduit.

Configuring conduit synchronization

The conduit needs to communicate with a MobiLink synchronization server to pass upload and download streams between the UltraLite application and the consolidated database. You can provide the information needed by the conduit to locate the MobiLink synchronization server in a **stream_parms** member of the UltraLite **ul_synch_info** structure supplied to **PalmExit**. If you did not specify a non-null **stream_parms** value, you can enter the required parameters from the HotSync manager.

In addition, if you are using a strongly encrypted UltraLite database, you can save the encryption key so that you do not have to enter it on each synchronization.

If you have Palm Desktop software installed, the Adaptive Server Anywhere installation creates registry entries for the **CustDB** sample application. You can use these entries as a starting point for your own application.

For information on **stream_parms**, see "Adding HotSync or ScoutSync synchronization to Palm applications" on page 272.

❖ To configure the HotSync conduit for synchronization:

1 Right-click the HotSync Manager icon in the system tray, and choose Custom from the popup menu.

2 Select your MobiLink conduit from the list of conduit names, and click Change.

3 Enter a set of stream parameters in the Synchronization Parameters text box. These parameters are the same as those in a **stream_parms** parameter. For example:

```
stream=tcpip;host=localhost
```

For more information, see "Synchronization stream parameters" on page 399.

4 If the database is strongly encrypted, you can enter the encryption key in the Encryption Key text box. If no key is entered, you will be prompted for the encryption key on each synchronization.

5 Click OK to complete the entry. The HotSync conduit is now ready to use.

Registry location

The stream parameters and encryption key are stored in the registry in *HKEY_CURRENT_USER\Software\Sybase\Adaptive Server Anywhere\8.0\Conduit\Creator ID*, where *Creator ID* is application-dependent.

A secondary location for HotSync synchronization depends on the version of the Palm Computing Platform software you are using. They are made under the *HKEY_CURRENT_USER\Software\U.S. Robotics* or the *HKEY_CURRENT_USER\Software\Palm Computing* folder.

Location of synchronization log files

The HotSync Manager uses log files to record actions. HotSync writes log information in the user-specific subdirectory **User***HotSync.log* of your Pilot or Palmdirectory. Here, HotSync records when each synchronization takes place and whether each installed conduit worked as expected.

Configuring ScoutSync synchronization

ScoutSync support deprecated
Version 8.0.x will continue to support ScoutSync up to version 3.6, but the next major release of SQL Anywhere Studio will not support ScoutSync.

ScoutSync technology is similar to HotSync. ScoutSync technology allows multiple users to simultaneously synchronize multiple Palm devices with the consolidated database.

ScoutSync synchronization is initiated from the Palm ScoutSync client, not directly from the UltraLite application. The ScoutSync client communicates with the ScoutSync server, which loads the MobiLink ScoutSync conduit. Each instance of the conduit manages all applications on a single Palm device. Each application can specify its own **synchronization conduit**. For MobiLink synchronization, the conduit is the MobiLink ScoutSync conduit.

The MobiLink ScoutSync conduit is a COM object. Since a ScoutSync conduit is based on COM, it can be installed on any machine and not only the one running the ScoutSync Application server.

For an overview of ScoutSync synchronization, see "HotSync and ScoutSync architecture" on page 269.

Configuring the MobiLink ScoutSync conduit

During ScoutSync synchronization, the ScoutSync Application Server starts the conduit, *dbscout8.dll*, which sends the upload stream to a MobiLink synchronization server, and receives the download stream from the MobiLink synchronization server.

The MobiLink ScoutSync conduit synchronizes with MobiLink synchronization server using TCP/IP, HTTP, or HTTPS streams. You specify the stream and stream parameters in your UltraLite application **PalmExit** call or in the *ClientParms* registry entry.

Registry location

Registry entries are located under *HKEY_CURRENT_USER\Software\Sybase\Adaptive Server Anywhere\8.0\Conduit\Creator ID*, where *Creator ID* is the Creator ID of the Palm application. Each application having a different Creator ID will have its own folder.

MobiLink must be started before using ScoutSync
Before performing ScoutSync, the MobiLink synchronization server must be running and be ready to accept connections from the MobiLink ScoutSync conduit.

Setting up for ScoutSync synchronization

In order to perform ScoutSync synchronization, you must perform the following:

- Configure the ScoutSync Application Server:
  - Setup the MobiLink ScoutSync conduit.
  - Setup a user profile.
- Configure the ScoutSync Client on the Palm device.
- Configure RAS for ScoutSync Client to access ScoutSync Application Server via TCP/IP.
- Optionally configure the MobiLink ScoutSync conduit to work with HotSync manager.

Configuring the ScoutSync Application Server

The following steps cover installation of the ScoutSync conduit on the ScoutSync Application Server.

For information on the COM properties of the MobiLink ScoutSync conduit and for instructions on how to install it on another machine, see your ScoutSync documentation.

❖ **To setup the MobiLink ScoutSync conduit:**

1 Register *dbscout8.dll* using the following command:

```
regsvr32 dbscout8.dll
```

A message box appears, indicating that the registration was successful.

2 Start the Scout Server Service.

See your ScoutSync documentation for instructions.

3 Start the Scout Management console.

See your ScoutSync documentation for instructions.

4 Connect to the ScoutSync server by selecting Tools➤Connect to Server in the ScoutSync Management console.

5 Add a new conduit in the ScoutSync Management console. Click the New Conduit button. Alternatively, you can right click on Conduits under Sync Services in the hierarchy tree and select New Conduit. Enter a conduit name **ULSync**, and set the Conduit Prog ID to **ULSctSyn.ULSSCond**. Select **PalmSyncService** under Supported Services.

❖ **To setup a user profile:**

1 Add a new user profile. Click the New User Profiles button. Alternatively, you can right click on User Profiles in the hierarchy tree and select New User Profile. Click on the User Info Tab and enter a profile name, **ULSyncProfile**.

2 Click on the Conduits Tab. Select **ULSync** and click Add. ULSync will be added to the Assigned list.

Configuring the ScoutSync Client on the Palm device

❖ **Configuring the ScoutSync Client on the Palm device**

1 Install *Scout.prc* and *ScoutUpdateClient.prc*, available under the *ScoutSync\Server\Services\PalmAdminService\clients* directory of your main Scout directory, on the Palm device.

For information on how to install an application onto the Palm device, see "Deploying Palm applications" on page 291.

2 Configure the ScoutSync Client Preferences settings on the Palm device as described in the following steps. Begin by tapping the Applications silk-screen icon to open the Applications Launcher.

3 Tap the ScoutSync icon to display the main ScoutSync screen.

4 Tap the Preference button located at the bottom of the ScoutSync screen. The ScoutSync Preferences screen appears.

5 Enter the ScoutSync Server Name, Port Number and Profile. The default connection port is 8025 and should not be changed unless instructed to do so by your system administrator.

6 Tap the word Unassigned in the Password field to display the Password box. Enter a password and tap the OK button, if applicable. The word "Assigned" displays after you have entered a password.

7 Tap the OK button on the ScoutSync Preferences screen to submit your changes.

Configuring RAS TCP/IP synchronization

❖ **To configure RAS for ScoutSync Client to access ScoutSync Application Server via TCP/IP**

♦ For step by step instructions, refer to "Configuring RAS TCP/IP synchronization via serial port connection" on page 287.

Using ScoutSync for the first time

After ScoutSync has been set up, follow these step by step instructions to perform ScoutSync synchronization:

1 Launch the ScoutUpdateClient program on the Palm device. Click on the Update ScoutSync Client button to log onto the ScoutSync server and update your client.

2 Launch the ScoutSync program on the Palm and tap on the ScoutSync icon. An initial ScoutSync is required to download the list of conduits.

3 From the ScoutSync program, make sure that the **ULSync** entry is chosen in the conduit list.

Now you are ready to perform ScoutSynchronization from your Palm device.

Location of synchronization log files for ScoutSync

ScoutSync uses log files to record their actions:

♦ By default, ScoutSync writes log information into the file *user.log* in the subdirectory *ScoutSync\Server\Users\profile* of your main Scout directory, where *profile* is the name of your user profile. Here, ScoutSync records when each synchronization takes place.

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

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

☞ For a general description of how to add synchronization to UltraLite applications, see "Adding synchronization to your application" on page 94.

Transport layer security on the Palm Computing Platform

You can use transport-layer security with Palm applications built with Metrowerks CodeWarrior. However, transport-layer security is unavailable for Palm applications built with PRC Tools.

☞ For information on transport-layer security, see "Transport-Layer Security" on page 283 of the book *MobiLink Synchronization User's Guide*.

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

Palm devices can synchronize using TCP/IP, HTTP, or HTTPS communication by setting the **stream** member of the **ul_synch_info** structure to the appropriate stream, and calling **ULSynchronize** (embedded SQL) or the **ULConnection.Synchronize** method (C++ API) to carry out the synchronization.

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

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

☞ For information on the **ul_synch_info** structure, see "Synchronization stream parameters" on page 399.

Using multiple synchronization methods

Switching between two or more synchronization techniques from a single UltraLite application is only convenient when using varied connection points for synchronization. Switching techniques to connect to the same machine is usually awkward.

Configuring TCP/IP, HTTP, or HTTPS synchronization

This section describes how to configure the synchronization setup for UltraLite Palm applications using TCP/IP or HTTP synchronization.

For information on synchronization architecture for HTTP or TCP/IP communications, see "Parts of the synchronization system" on page 10 of the book *MobiLink Synchronization User's Guide*.

Configuring TCP/IP synchronization for the Palm Computing Platform

There are two ways of using TCP/IP networking in a Palm device. In either case, you must connect to a Remote Access Service (RAS). The difference lies in how you make the connection to the RAS.

- **Use a modem to dial into an ISP** The Internet Service Provider (ISP) must provide access to a Remote Access Service (RAS). The components of the connection are as follows:

```
Application
<--> Palm Net Library
<--> Palm modem
<--> NT RAS
<--> TCP/IP network
```

- **Connect via the serial port to a Windows NT machine** The components of the connection are as follows:

```
Application
<--> Palm Net Library
<--> serial cable
<--> NT RAS
<--> TCP/IP network
```

When using TCP/IP, the MobiLink synchronization server can be any machine on the network that is accessible via TCP/IP.

Before synchronization, the following conditions must be satisfied:

1. The device must be in its cradle.
2. If you are using the serial port to connect to a Windows NT machine running RAS, the HotSync Manager and other applications that use the serial port must be shut down. Windows NT only allows one application to use a serial port at a time.

3 The MobiLink synchronization server must be started. By default, the MobiLink synchronization server listens for TCP/IP communications over port 2439.

4 The Palm device must have Network settings in place so that it can connect to the network. Modem settings are also required if using a modem to dial into an ISP.

Configuring RAS TCP/IP synchronization via modem

To use this method, you must have access to a Remote Access Service when you dial in.

❖ **To configure a Palm device for RAS TCP/IP via a modem:**

1 Install the modem by plugging the Palm device into the modem module.

2 Go to the Preferences (Prefs) panel and choose Network from the dropdown list at the top right of screen.

3 Choose the Windows RAS service.

4 Set the dial-in username and password.

5 Set the phone number to the number at which the Remote Access Service can be reached. Obtain this number from your ISP.

6 Tap on Details.

7 Set the connection type (usually PPP).

8 Set the DNS and IP addresses as recommended by your network administrator.

9 Tap on Script and enter the script recommended by your ISP. This script will be similar to the following sample.

```
Wait For: Username:
Delay: 1
Send UserID:
Send CR:

Wait For: Password:
Delay: 1
Send Password:
Send CR:

Wait For: >
Delay: 1
Send: ppp
Send CR:

End:
```

Tap on OK until you are back to the Network Preferences.

At this point, you are ready to test your TCP/IP connection.

Configuring RAS TCP/IP synchronization via serial port connection

This procedure involves actions both on Windows NT and on the Palm Computing device.

❖ **To configure Windows NT for RAS TCP/IP via serial port:**

1. From the Control Panel, open Modems. Make sure that a modem is defined for **Dial-Up Networking Serial Cable between 2 PCs** on the COM port to which the cradle is connected.
2. Set the speed for this modem to the baud rate you are using. The default is 19200.
3. Make sure TCP/IP protocol is installed. Select Start➤Settings➤Control Panel and double-click the Network icon. Click on the Protocols tab. If there is no TCP/IP entry, choose Add to install it.
4. Enable IP Forwarding (in the Routing tab of TCP/IP properties)
5. Under the Services tab, make sure that Remote Access Service is installed. If there is no entry for Remote Access Service, choose Add to install it.

   In Remote Access Service Properties, add **Dial Up Network serial cable between 2 pc's** for that COM port if the cradle's COM port is not in the list of ports.
6. Configure this entry to receive calls. In the RAS Network properties set encryption settings to **Allow any authentication including clear text**. In the RAS Network properties allow only TCP/IP client.
7. Configure TCP/IP. Allow clients to access the entire network. Assigning the TCP/IP addresses depends on your network. Contact your network administrator for details.
8. Add a user for dial-in access. Select Start➤Programs➤Administrative Tools➤User Manager. Uncheck **User Must Change Password at Next Logon**. Choose the Dialin button, and grant dialin permission to user with No Call Back.
9. If the RAS COM port is the same one that HotSync Manager uses, shut down the HotSync Manager or any other applications that use the COM port.
10. Start the Remote Access Administrator. Select Start➤Programs➤Administrative Tools➤Remote Access Admin.

11 Start the RAS service. Select Server➤Start Remote Access Service. Choose to start the service on the local machine.

HotSync Manager or any other applications that use the serial port and the RAS service will not run at the same time. One must be shut down first for the other to run, as Windows NT prevents two different applications from accessing the same serial port. You have to stop the RAS service (Server➤Stop Remote Access Service from the Remote Access Admin) before you can restart the HotSync Manager. Alternatively, you can use separate serial ports.

Once the RAS service is running, it is ready to receive connection requests via the serial port.

❖ **To configure a Palm device for RAS TCP/IP via serial port:**

1 Go to the Preferences (Prefs) panel and choose Network from the dropdown list at the top right of screen.

2 Choose the Windows RAS service.

3 Set the dial-in username and password.

4 Set the Palm to use the serial port.

- For Palm OS 3.3 and above, select **Direct serial**.
- For earlier versions of the Palm OS, set the phone number to **00** (zero zero). This is a special phone number that tells the Palm to use the serial port directly, instead of a modem.

5 Tap on Details.

6 Set the connection type (usually PPP).

7 Set the DNS and IP addresses as recommended by your network administrator.

8 Tap on Script and enter the following script:

```
Send: CLIENT
Send CR:
Delay: 1
Send: CLIENT
END
```

Tap on OK until you are back to the Network Preferences

At this point, you are ready to test your TCP/IP connection.

Testing and troubleshooting

❖ **To test the connection:**

- **via modem** Connect the Palm device to the modem and follow the instructions provided by your ISP for connecting to their network. Once connected, tap the Connect button in Prefs➤ Network on the Palm device.
- **via serial port** Ensure RAS is running on the Windows NT machine. Place the Palm device in the cradle and connect the cradle to the correct COM port on the Windows NT machine. Tap the Connect button in Prefs➤ Network on the Palm device.

With TCP/IP, there are two levels of service. At the minimum level, you can connect to another TCP/IP host using an IP number of the following form.

```
NNN.NNN.NNN.NNN
```

At the next level, when a DNS server is properly configured, you are able to connect to another host by name.

```
some_host_machine.any_company.com
```

Having a DNS service is more convenient, since most people are better at remembering a name than a number. As long as you have the minimum TCP/IP service, and an IP number, you can synchronize an UltraLite application using TCP/IP.

There are a number of steps you can take to troubleshoot TCP/IP connections on the Palm device.

- Hitting the scroll down button on the Palm device during the connection phase displays the progress of the connection.
- The connection log is accessible from the Network Preferences panel. Choose View Log from the Options menu to see information about the network connection. The log is an interactive utility for controlling and viewing your connection information. Enter ? for help.
- There are several tools for testing a TCP/IP connection from the Palm. You can find most of them at the following locations:

```
http://www.roadcoders.com
```

```
http://www.palmcentral.com
```

There are also steps you can take for troubleshooting on Windows NT:

- In the Remote Access Admin, double-click on the running server.
- Select the appropriate port and choose Port Status. The Port Status dialog shows you the Line condition (connected or waiting for a call) and lets you watch the byte counts for both directions.

Configuring HTTP or HTTPS synchronization for the Palm Computing platform

To use HTTP or HTTPS synchronization, you must first configure RAS TCP/IP synchronization. For information on configuring RAS, see "Configuring TCP/IP synchronization for the Palm Computing Platform" on page 285.

When using HTTP or HTTPS, the MobiLink synchronization server can be any machine on the network that is accessible via the protocol.

❖ **To synchronize using HTTP or HTTPS:**

1. Place the Palm device in its cradle.
2. If you are using the serial port to connect to a Windows NT machine running RAS, shut down the HotSync Manager and other applications that use the serial port. Windows NT only allows one application to use a serial port at a time.
3. Start the MobiLink synchronization server.
4. Ensure that the network settings on the Palm device are configured so that it can connect to the network. Modem settings are also required if using a modem to dial into an ISP.

   For more information, see "Configuring TCP/IP synchronization for the Palm Computing Platform" on page 285.

Deploying Palm applications

This section describes the following aspects of deploying Palm applications:

- Deploying the application.

  See "Deploying applications on the Palm Computing Platform" on page 291.

- Deploying the MobiLink synchronization conduit for HotSync.

  See "Deploying the MobiLink synchronization conduit" on page 291.

- Deploying an initial copy of the UltraLite database.

  See "Deploying UltraLite databases on the Palm Computing Platform" on page 292.

Deploying applications on the Palm Computing Platform

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

❖ **To install an application on a Palm device:**

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

Deploying the MobiLink synchronization conduit

For applications using HotSync or ScoutSync synchronization, each end user must have the MobiLink synchronization conduit installed on their desktop. This installation requires the following:

- **Deploy the conduit files** The files for the conduit must be installed into a location in the end user's system path.

  For a list of conduit files, see "HotSync conduit files" on page 275.

- **Install the conduit** You can deploy the conduit installation utility to your end users and provide instructions for them to run it, or you can use the HotSync Manager to install the conduit.

  For instructions, see "Adding the MobiLink conduit into HotSync manager" on page 275.

- **Configure the conduit** If you did not include a **stream_parms** parameter in your UltraLite **ul_synch_info** structure, enter these parameters from the HotSync manager. Also, if you are using an encrypted database, you may want to enter the encryption key.

  For instructions, see "Configuring conduit synchronization" on page 277.

Deploying UltraLite databases on the Palm Computing Platform

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

For more information, see "The UltraLite utility" on page 426.

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

For information on installing the synchronization conduit, see "Configuring HotSync synchronization" on page 274.

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

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

For more information, see "UL_STORE_PARMS macro" on page 428.

CHAPTER 12

Developing Applications for Windows CE

About this chapter

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

Contents

Introduction

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

☞ For a list of supported host platforms and development tools for Windows CE development, and for a list of supported target Windows CE platforms, see "Supported platforms for C/C++ applications" on page 6.

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

Preparing for Windows CE development

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

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

A first application

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

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

- Compiler settings:
  - Add $(ASANY8)\h to the include path.
  - Define appropriate compiler directives. For example, the UNDER_CE macro should be defined for eMbedded Visual C++ projects.
- Linker settings:
  - Add "*$(ASANY8)\ultralite\ce\processor\lib\ulrt.lib*"

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

    ```
    "$(ASANY8)\win32\sqlpp" -q -c "dsn=UltraLite 8.0
    Sample" $(InputPath) ul_database.cpp
    ```

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

Choosing how to link the runtime library

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

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

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

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

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

Building the CustDB sample application

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

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

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

❖ To build the CustDB sample application

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

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

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

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

The build process uses the SQL preprocessor, *sqlpp*, to preprocess the file *CustDB.sqc* into the file *CustDB.c*. This one-step process is useful in smaller UltraLite applications where all the embedded SQL can be confined to one source module. In larger UltraLite applications, you need to use multiple *sqlpp* invocations followed by one *ulgen* command to create the customized remote database.

For more information, see "Preprocessing your embedded SQL files" on page 201.

Storing persistent data

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

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

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

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

Example

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

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

Deploying Windows CE applications

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

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

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

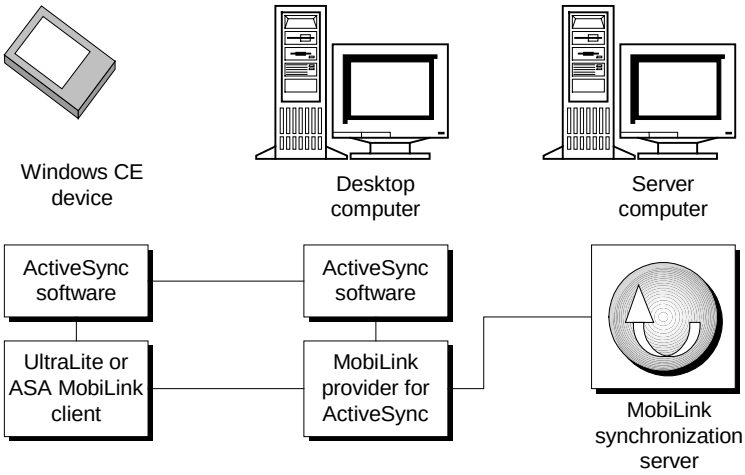
   The UltraLite runtime DLL is in chip-specific directories under the *UltraLite\ce* subdirectory of your SQL Anywhere directory.

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

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

Deploying applications that use ActiveSync

Applications that use ActiveSync synchronization must be registered with ActiveSync as well as copied onto the device. Also, each desktop machine must have the MobiLink provider for ActiveSync installed. The architecture for ActiveSync is illustrated in the following diagram.



❖ **To deploy ActiveSync applications:**

1 Install the MobiLink provider for ActiveSync on each end user's machine.

An ActiveSync provider install utility is provided with SQL Anywhere. This is the *dbasinst.exe* command-line utility.

For information, see "Installing the MobiLink provider for ActiveSync" on page 301, and "ActiveSync provider installation utility" on page 610 of the book *MobiLink Synchronization User's Guide*.

2 Register the application for use with ActiveSync.

You can register the application either by using ActiveSync, or by using the ActiveSync provider installation utility *dbasinst.exe*.

For information see "Registering applications for use with ActiveSync" on page 302.

3 Copy the application onto the device.

If your application is a single executable, statically linked with the runtime library, you can use the ActiveSync provider installation utility *dbasinst.exe* to copy the application to the device.

If the application includes multiple files (for example, if you use the UltraLite runtime DLL rather than the static runtime library), you must copy the files onto the device in some other way.

Installing the MobiLink provider for ActiveSync

Before you register your application for use with ActiveSync, you must install the MobiLink provider for ActiveSync using the installation utility (*dbasinst.exe*).

The MobiLink provider for ActiveSync includes a desktop component and a device component. You must install the provider for each device that synchronizes through your desktop machine.

When you have installed the MobiLink provider for ActiveSync you must register each application separately. For instructions, see "Registering applications for use with ActiveSync" on page 302.

❖ To install the MobiLink provider for ActiveSync:

1. Ensure that you have the ActiveSync software on your machine, and that the Windows CE device is connected.
2. Enter the following command to install the MobiLink provider:

   ```
   dbasinst -k desk-path -v dev-path
   ```

   where *desk-path* is the location of the desktop component of the provider (*dbasdesk.dll*) and *dev-path* is the location of the device component (*dbasdev.dll*).

   If you have SQL Anywhere installed on your machine, *dbasdesk.dll* is in the *win32* subdirectory of your SQL Anywhere directory and *dbasdev.dll* is in a platform-specific directory in the *CE* subdirectory. These directories are default search locations, and you can omit both `-k` and `-v` command-line switches.

   For more information, see "ActiveSync provider installation utility" on page 610 of the book *MobiLink Synchronization User's Guide*.
3. Restart your machine.

   ActiveSync does not recognize new providers until the machine is restarted.
4. Enable the MobiLink provider.
   - From the ActiveSync window, click Options.
   - Check the MobiLink item in the list and click OK to activate the provider.
   - To see a list of registered applications, click Options again, choose the MobiLink provider, and click Settings.

For more information on registering applications, see "Registering applications for use with ActiveSync" on page 302.

Registering applications for use with ActiveSync

You can register you application for use with ActiveSync either by using the ActiveSync provider install utility or using the ActiveSync software itself. This section describes how to use the ActiveSync software.

For information on the alternative approach, see "ActiveSync provider installation utility" on page 610 of the book *MobiLink Synchronization User's Guide*.

❖ **To register an application for use with ActiveSync:**

1. Ensure that the MobiLink provider for ActiveSync is installed.

   For information, see "Installing the MobiLink provider for ActiveSync" on page 301.

2. Start the ActiveSync software on your desktop machine.
3. From the ActiveSync window, choose Options.
4. From the list of information types, choose MobiLink and click Settings.
5. In the MobiLink Synchronization dialog, click New. The Properties dialog appears.
6. Enter the following information for your application:
   - **Application name** A name identifying the application to be displayed in the ActiveSync user interface.
   - **Class name** The registered class name for the application.

     See "Assigning class names for applications" on page 303
   - **Path** The location of the application on the device.
   - **Arguments** Any command-line arguments to be used when ActiveSync starts the application.
7. Click OK to register the application.

Assigning class names for applications

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

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

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

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

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

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

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

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

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

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

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

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

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

   ```
   m_pMainWnd = &dlg;
   ```

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

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

5 Save your changes.

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

6 Modify the resource file for your project.

- Open your resource file (which has an extension of *.rc*) in a text editor such as notepad.
- Locate the resource ID of your main dialog.
- Change the main dialog's definition to use the new window class as in the following example. The *only* change that you should make is the addition of the **CLASS** line:

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

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

- Save the *.rc* file.

7 Reopen eMbedded Visual C++ and load your project.

8 Add code to catch the synchronization message.

For information, see "Adding ActiveSync synchronization (MFC)" on page 306.

Synchronization on Windows CE

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

- ActiveSync See "Adding ActiveSync synchronization to your application" on page 305
- TCP/IP See "TCP/IP, HTTP, or HTTPS synchronization from Windows CE" on page 308.
- HTTP See "TCP/IP, HTTP, or HTTPS synchronization from Windows CE" on page 308.

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

For information on adding synchronization to your application, see "Adding synchronization" on page 71. For detailed information on synchronization parameters, see "Synchronization stream parameters" on page 399.

Adding ActiveSync synchronization to your application

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

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

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

For information on setting up ActiveSync synchronization, see "Deploying applications that use ActiveSync" on page 299.

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

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

Adding ActiveSync synchronization (Windows API)

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

Here is an example of how to handle the message:

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

  switch( uMsg ) {

  // code to handle other windows messages

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

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

For more information, see "ULIsSynchronizeMessage function" on page 243.

Adding ActiveSync synchronization (MFC)

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

Your application must create and register a custom window class name for notification. See "Assigning class names for applications" on page 303.

❖ **To add ActiveSync synchronization in the main dialog class:**

♦ Add a registered message and declare a message handler.

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

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

♦ Implement the message handler.

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

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

The return value of this function should be 0.

For information on handling the synchronization message, see "ULIsSynchronizeMessage function" on page 243.

❖ **To add ActiveSync synchronization in the Application class:**

1 Open up the Class Wizard for the application class.

2 In the Messages list, highlight PreTranslateMessage and then click the Add Function button.

3 Click the Edit Code button. The PreTranslateMessage function appears. Change it to read as follows:

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

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

For information on handling the synchronization message, see "ULIsSynchronizeMessage function" on page 243.

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

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

CHAPTER 13

Developing Applications for VxWorks

About this chapter

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

Contents

Introduction

The following instructions pertain to writing and building UltraLite applications for use with the VxWorks platform.

☞ For a list of supported host platforms and development tools for VxWorks development, and for a list of supported target VxWorks platforms, see "Supported platforms for C/C++ applications" on page 6.

Installation directory
Tornado/VxWorks development tools may not support the use of directories with spaces in the name. As UltraLite header and include files are installed under the Adaptive Server Anywhere installation directory, which by default has spaces, you may wish to reinstall. An alternative is to reset your ASANY8 environment variable to use the short form of the directory name (*c:\progra~1\...*)

Features and limitations

Follow the *VxWorks Programmer's Guide, Tornado User's Guide* as well as BSP-specific documentation for instructions on setting up your system and troubleshooting.

You should also note the following when developing UltraLite applications for the VxWorks platform.

Synchronization

You may use TCP/IP or HTTP for UltraLite synchronization. The TCP/IP components are typically included and initialized by default in VxWorks.

Resolving host names

The default VxWorks configuration does not include resolving of host names via DNS. Therefore, if you use a host name to specify the location of the MobiLink synchronization server for synchronization, you must define INCLUDE_DNS_RESOLVER and associated macros when building VxWorks. However, if you use only the IP number to specify your host machine, you do not need to include DNS support. Without DNS support, you may get a warning regarding an undefined `_resolvGetHostByName` symbol, but this warning will not affect the running or synchronization of your application as long as only an IP number is used.

Persistent storage

UltraLite requires a dosFs (MS-DOS-compatible file system) device or a functionally equivalent device to store the persistent data file. UltraLite defaults to using a device named **ULDEV:**. You can configure a storage device with this name and it will be used to store persistent data for the application, or you can override the default filename and specify a different device.

Persistent storage for CustDB sample

If a **ULDEV:** device does not exist when you run the CustDB sample, the application creates a **ULDEV:** device using a RAM disk and dosFs file system. The VxWorks RAM disk driver component is required, and you can include the component by defining INCLUDE_RAMDRV when building VxWorks.

For more information on persistent data storage, see "Storing persistent data" on page 318.

Building the CustDB sample application

The following are general guidelines for building the sample application included in your Adaptive Server Anywhere installation.

☞ For instructions on how to build *your own* UltraLite application, see "Building UltraLite VxWorks applications" on page 316.

CustDB is a simple sales-status application. It has already been built into an executable *custdb.out* that is located in the *UltraLite\vxw\platform* subdirectory, where *platform* indicates the chip for which the sample is compiled. Other files specific to the VxWorks platform are located in the *Samples\UltraLite\CustDB\vxw* directory. The *gnuvxw.bat* file in this subdirectory is a batch file used to build the sample application in embedded SQL or C++ API form, and the *custio.c* file contains the source code for the user interface of the sample application. The sample application uses standard I/O via **printf()** and **getchar()**.

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

Although the sample application has already been built into an executable, you can build it yourself by doing the following:

❖ To build the sample application

1. Open a Command Prompt window.
2. Run *torvars.bat* (included with Tornado) to set up the environment to include the Tornado compiler and environment variables. Minimally, you must set your WIND_BASE environment variable.
3. Change to the subdirectory *CustDB\vxw*, located in the Adaptive Server Anywhere *Samples\UltraLite\CustDB* directory.

   ```
   cd "%ASANY8%\Samples\UltraLite\CustDB\vxw"
   ```

4. Run *gnuvxw.bat* to generate an embedded SQL executable with name *custdb.out*, or a C++ API executable with the name *custdbapi.out*.

   This command runs the SQL preprocessor (*sqlpp.exe*) to preprocess the embedded SQL source file *custdb.sqc* and to generate source code that implements the SQL in the application.

Downloading the sample application to the device

Download the sample application module using the Tornado shell.

Downloading the sample application

The CustDB sample application has been built into an executable *custdb.out*, located in the *UltraLite\vxw\386* subdirectory. You can download this file to VxWorks using the following Tornado shell command:

```
ld(0,0,"c:/sybase/asa8/ultralite/vxw/386/custdb.out");
```

Running the sample application

To run the CustDB sample application after downloading, call the custDB function by typing **custDB()** at the Tornado Shell prompt.

Viewing the data in the sample application

When you start the CustDB sample application, the application prompts you for a method of synchronization. Press T if you wish to synchronize via TCP/IP and D if you wish to use the default method. The default method of synchronization for the sample application is TCP/IP to *localhost*. Note that the default VxWorks configuration does not automatically map *localhost* to your host machine and you may have to specify the name or IP address of your machine directly, after typing T.

The application subsequently prompts you for an employee ID. Enter a value of 50. The application also allows values of 51, 52, or 53. After synchronization, a set of customers, products, and orders are downloaded to your target machine.

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

For information about synchronization, see "Synchronization on the VxWorks platform" on page 319.

Here are some actions you can perform with the sample application:

- **Scroll through the outstanding orders** The application holds information about a set of orders. Scroll forward through the orders by pressing F and scroll backward by pressing B. For each order, this includes the ID number, the customer, the product, the quantity, and other information. Also included are a *status* column and a *notes* column, which you can modify from the application. You can approve, deny, add as well as delete an order.

  Only unapproved orders for the customers that you list in the *ulEmpCust* table 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.

- **Display a list of customers** The UltraLite application holds the complete list of customers from the consolidated database. To see this list, press C.

- **Display a list of products** The UltraLite application holds the complete list of products from the consolidated database. To see this list, press P.
- **Synchronize with the consolidated database** Press S to synchronize with the consolidated database.
- **Display help** Press ? to display a list that indicates which key to press for each task.

Enter keys in lower case
Enter keys in lower case when you perform actions with the sample application.

Building UltraLite VxWorks applications

The following are general guidelines for building *your own* UltraLite application.

For instructions on how to build the *sample* application included in your Adaptive Server Anywhere installation, see "Building the CustDB sample application" on page 312.

❖ To build an UltraLite VxWorks application:

1. Start the Adaptive Server Anywhere personal database server, specifying your reference database.
2. Preprocess your embedded SQL files using the SQL preprocessor *sqlpp*. When *sqlpp* is invoked, a C/C++ source file is generated for each *.sqc* file.

   If you have only one embedded SQL source file, *sqlpp* automatically runs the UltraLite analyzer in addition to preprocessing the SQL file. The analyzer generates more C/C++ code to implement your application database. You may also skip step 3.

   For more information, refer to "Preprocessing your embedded SQL files" on page 201.
3. Run the UltraLite generator *ulgen* to perform code generation. The generator creates a C/C++ source file.

   For more information, see "Generating the UltraLite data access code" on page 91.
4. Set up the environment to include the Tornado compiler and environment variables by running *torvars.bat.*
5. Invoke the compiler to compile all source files. For example, cc386 is the cross compiler for the Intel x86 BSP. This process generates a *.o* file for each C/C++ file. Note that C++ files require use of the *munch* tool.

   Code compiled for VxWorks should have the UNDER_VXW macro defined. For more information, see "UNDER_VXW macro" on page 433.

   For more information, refer to cross-development in the *Tornado User's Guide*.
6. Link all the object files along with the runtime library *libulrt.a* as follows:

   ```
   ld386 –o myapp.out myapp.o util.o –r -lulrt –L
   %ASANY8%\ultralite\vxw\386\lib
   ```

The runtime library *libulrt.a* is located in the *UltraLite\vxw\platform\lib* subdirectory. *myapp.out* will include the *.o* files (generated by the compiler) as well as the necessary modules acquired from the runtime library.

Note
ld will look for *libulrt.a* when *–lulrt* is specified.

Preserving data when upgrading

To avoid the possibility of losing data stored in your UltraLite database, it is recommended that you synchronize your existing UltraLite application with the consolidated database before upgrading your application.

If your new application introduces obfuscation or encryption of the database, or if the new schame is incompatible with the older version, data in the database is lost on upgrading.

Storing persistent data

UltraLite requires a dosFs (MS-DOS-compatible file system) device or a functionally equivalent device to store the persistent data. UltraLite defaults to using a device named *ULDEV:*. Unless you override the default persistent storage filename, this device must exist before **db_init** is called.

The following lines illustrate how to create such a device using a RAM disk and dosFs file system:

```
pBlkDev = ramDevCreate( 0, 512, 2048, 2048, 0 );
pVolDesc = dosFsMkfs( "ULDEV:", pBlkDev );
```

By default, the UltraLite runtime saves persistent data to the file *ULDEV:/ul_<project_name>.udb*, where the filename *ul_<project_name>* is truncated to 8 characters.

☞ You can configure various aspects of the database store using the UL_STORE_PARMS macro. For information on persistent-storage parameters, see "Configuring and managing database storage" on page 45.

Synchronization on the VxWorks platform

The VxWorks UltraLite applications can synchronize via TCP/IP or HTTP.

For information on synchronization architecture for TCP/IP see "Parts of the synchronization system" on page 10 of the book *MobiLink Synchronization User's Guide*.

To synchronize a VxWorks UltraLite application using a TCP/IP socket connection, call **ULSynchronize** with the **ULSocketStream()** stream. The following embedded SQL example illustrates the arguments:

```
ul_synch_info synch_info;
ULInitSynchInfo( &synch_info );
synch_info.user_name = UL_TEXT( "50" );
synch_info.version = UL_TEXT( "custdb" );
synch_info.stream = ULSocketStream();
synch_info.stream_parms =
   UL_TEXT("host=localhost");
ULSynchronize( &synch_info );
```

When using TCP/IP sockets, the MobiLink synchronization server can be any machine on the network that is accessible via TCP/IP. Before **ULSynchronize** is called, the MobiLink synchronization server must be started as follows:

```
dbmlsrv8 -c "DSN=UltraLite 8.0 Sample" ...
```

By default, the MobiLink synchronization server listens on port 2439.

Synchronization for CustDB sample application

TCP/IP is the default method of synchronization for the CustDB sample application. In addition, UltraLite specifies *localhost* as the hostname by default. Note that the default VxWorks configuration does not automatically map *localhost* to your host machine and you may need to specify the name or IP address of your machine explicitly.

Transport-layer security on VxWorks

You can use transport-layer security on VxWorks on Intel x86 chips and on the Windows VxSim emulator.

For information on transport-layer security, see "Transport-Layer Security" on page 283 of the book *MobiLink Synchronization User's Guide*. For information on synchronization parameters, see "Synchronization stream parameters" on page 399.

You must ensure that the time is set properly on the client VxWorks device, as certificate dates are checked during synchronization. The time is not set automatically on VxWorks devices. You can set the time on the device by using the **clock_settime()** function. When the time is incorrect, MobiLink returns an error.

PART THREE

Developing UltraLite Java Applications

This part focuses on details of the development process that are specific to Java. It explains how to write and build Java applications and provides a tutorial that guides you through the development process.

CHAPTER 14

Tutorial: Build an Application Using Java

About this chapter

This chapter provides a tutorial that guides you through the process of developing a Java UltraLite application. The first section describes how to build a very simple Java UltraLite application. The second section describes how to add synchronization to your application.

For an overview of the development process and background information on the UltraLite database, see "Designing UltraLite Applications" on page 41.

For information on developing Java UltraLite Applications, see "Developing UltraLite Java Applications" on page 337.

Contents

Introduction

This tutorial describes how to construct a very simple application using UltraLite Java. The application is a command-line application, developed using the Sun JDK, which queries data in the *ULProduct* table of the *UltraLite 8.0 Sample* database.

In this tutorial, you create a Java source file, create a project in a reference database, and use these sources to build and run your application. The early lessons describe a version of the application without synchronization. Synchronization is added in a later lesson.

To follow the tutorial, you should have a Java Development Kit installed.

Overview

In the first lesson, you write and build an application that carries out the following tasks.

1. Connects to an UltraLite database, consisting of a single table. The table is a subset of the *ULProduct* table of the UltraLite Sample database.
2. Inserts rows into the table. Initial data is usually added to an UltraLite application by synchronizing with a consolidated database. Synchronization is added later in the chapter.
3. Writes the rows of the table to standard output.

In order to build the application, you must carry out the following steps:

1. Create an Adaptive Server Anywhere reference database.

   Here we use the UltraLite sample database (CustDB).
2. Add the SQL statements to be used in your application to the reference database.
3. Run the UltraLite generator to generate the Java code and also an additional source file for this UltraLite database.

   The generator writes out a *.java* file holding the SQL statements, in a form you can use in your application, and a *.java* file holding the code that executes the queries.
4. Write source code that implements the logic of the application.

   Here, the source code is a single file, named *Sample.java*.
5. Compile and run the application.

In the second lesson you add synchronization to your application.

Create a directory to hold your files

In this tutorial, you will be creating a set of files, including source files and executable files. You should make a directory to hold these files. In addition, you should make a copy of the UltraLite sample database so that you can work on it, and be sure you still have the original sample database for other projects.

Copies of the files used in this tutorial can be found in the *Samples\UltraLite\JavaTutorial* subdirectory of your SQL Anywhere directory.

❖ **To prepare a tutorial directory:**

- Create a directory to hold the files you will create. In the remainder of the tutorial, we assume that this directory is *c:\JavaTutorial*.
- Make a backup copy of the UltraLite 8.0 Sample database into the tutorial directory. The UltraLite 8.0 Sample database is the file *custdb.db*, in the *UltraLite\Samples\CustDB* subdirectory of your SQL Anywhere installation directory. In this tutorial, we use the original UltraLite 8.0 Sample database, and at the end of the tutorial you can copy the untouched version from the *APITutorial* directory back into place.

Lesson 1: Add SQL statements to your reference database

The reference database for this tutorial is the UltraLite 8.0 Sample database. In a later step, you use this same directory as a consolidated database for synchronization. These two uses are separate, and in your work you may use different databases for the two roles.

Add the SQL statements to the reference database using the *ul_add_statement* stored procedure. In this simple application, use the following statements:

- **Insert** An INSERT statement adds an initial copy of the data into the *ULProduct* table. This statement is not needed when synchronization is added to the application.
- **Select** A SELECT statement queries the *ULProduct* table.

When you add a SQL statement, you must associate it with an UltraLite project. Here, we use a project name of *Product*. You must also add a name for the statement, which by convention is in upper case.

❖ To add the SQL statements to the reference database:

1. Start Sybase Central, and connect to the UltraLite 8.0 Sample data source using the Adaptive Server Anywhere plug-in.
2. Add a project to the database:
   - In Sybase Central, open the custdb database.
   - Open the UltraLite projects folder.

     The folder contains one project already: the custapi project used for the sample application. You must create a new project.
   - Double-click Add UltraLite Project.
   - Enter **Product** as the project name, and click Finish.
3. Add the INSERT statement to the Product project.
   - Double-click Product to open the project.
   - Double-click Add UltraLite Statement.
   - Enter **InsertProduct** as the statement name. Click Next.
   - Enter the statement text:

     ```
     INSERT INTO ULProduct ( prod_id, price, prod_name)
     VALUES (?,?,?)
     ```

The first argument is the project name, the second is the statement name, and the third is the SQL statement itself. The question marks in the SQL statement are placeholders, and you can supply values at runtime.

- Click Finish to complete the operation.

This operation in Sybase Central is equivalent to executing the following stored procedure call:

```
call ul_add_statement( 'Product', 'InsertProduct',
   'INSERT INTO ULProduct( prod_id, price, prod_name)
    VALUES (?,?,?) ' )
```

4 Add the SELECT statement to the Product project.

- From the Product project, double-click Add UltraLite Statement.
- Enter **SelectProduct** as the statement name. Click Next.
- Enter the statement text:

  ```
  SELECT prod_id, prod_name, price FROM ULProduct
  ```

- Click Finish to complete the operation.

This operation in Sybase Central is equivalent to executing the following stored procedure call:

```
call ul_add_statement( 'Product', 'SelectProduct',
  'SELECT prod_id, prod_name, price FROM ULProduct')
```

5 Close Sybase Central.

You have now added the SQL statements to the database, and you are ready to generate the UltraLite database.

For more information, see "ul_add_project system procedure" on page 412, and "ul_add_statement system procedure" on page 411.

Lesson 2: Run the UltraLite generator

The UltraLite generator writes out two Java files. One contains the SQL statements, as an interface definition, which is here named *ISampleSQL.java*. You can use this interface definition in your main application code. The second file holds the code that implements the queries and the database, and is here named *SampleDB.java*.

❖ **To generate the UltraLite database code:**

1 Open a command prompt, and go to your *JavaTutorial* directory.

2 Run the UltraLite generator with the following arguments (all on one line):

```
ulgen –a -t java -c "dsn=UltraLite 8.0 Sample"
-j Product -s ISampleSQL -f SampleDB
```

The arguments have the following meanings:

- **-a** Generate SQL string names in upper case. The **InsertProduct** and **SelectProduct** statements come to **INSERT_PRODUCT** and **SELECT_PRODUCT**.
- **-t** The language of the generated code. Generate Java code instead of C code.
- **-c** The connection string to connect to the database.
- **-j** The UltraLite project name. This name corresponds to the project name you provided when you added the SQL statement to the database. The generator produces code only for those statements associated with this project.
- **-s** The name of the interface that contains the SQL statements as strings.
- **-f** The name of the file that holds the generated database code and query execution code.

Lesson 3: Write the application code

The following code listing holds a very simple UltraLite application.

You can copy the code into a new file and save it as *Sample.java* in your *c:\JavaTutorial* directory, or open a new file and type the content. You can find this source code in *Samples\UltraLite\JavaTutorial\Sample.java*.

```
// (1) Import required packages
import java.sql.*;
import ISampleSQL.*;
import ianywhere.ultralite.jdbc.*;
import ianywhere.ultralite.support.*;

// (2) Class implements the interface containing SQL
statements
public class Sample implements ISampleSQL
{
  public static void main( String[] args )
  {
    try{
        // (3) Connect to the database
        java.util.Properties p = new
            java.util.Properties();
        p.put( "persist", "file" );
        SampleDB db = new SampleDB( p );
        Connection conn = db.connect();

        // (4) Initialize the database with data
        PreparedStatement pstmt1 =
            conn.prepareStatement( INSERT_PRODUCT );
        pstmt1.setInt(1, 1);
        pstmt1.setInt(2, 400);
        pstmt1.setString(3, "4x8 Drywall x100");
        int rows1=pstmt1.executeUpdate();
        pstmt1.setInt(1, 2);
        pstmt1.setInt(2, 3000);
        pstmt1.setString(3, "8’ 2x4 Studs x1000");
        int rows2=pstmt1.executeUpdate();

        // (5) Query the data and write out the results
        Statement stmt = conn.createStatement();
        ResultSet result = stmt.executeQuery(
             SELECT_PRODUCT );
        while( result.next() ) {
            int id = result.getInt( 1 );
            String name = result.getString( 2 );
            int price = result.getInt( 3 );
            System.out.println ( name +
                " \tId=" + id +
                " \tPrice=" + price );
        }
```

```
            // (6) Close the connection to end
            conn.close();
        } catch (SQLException e) {
            Support.printException( e );
        }
      }
    }
```

Explanation of the sample program

Although too simple to be useful, this example contains elements that must be present in all Java programs used for database access. The following describes the key elements in the sample program. Use these steps as a guide when creating your own Java UltraLite application.

The numbered steps correspond to the numbered comments in the source code.

1 Import required packages.

The sample program utilizes JDBC interfaces and classes and therefore must import this package. It also requires the UltraLite runtime classes, and the generated interface that contains the SQL statement strings.

2 Define the class.

The SQL statements used in the application are stored in a separate file, as an interface. The class must declare that it implements the interface to be able to use the SQL statements for the project. The class names are based on the statement names you provided when adding the statements to the database.

3 Connect to the database.

The connection is established using an instance of the database class. The database name must match the name of the generated Java class (in this case **SampleDB**). The `file` value of the `persist` Properties object states that the database should be persistent.

4 Insert sample data.

In a production application, you would generally not insert sample data. Instead, you would obtain an initial copy of data by synchronization. In the early stages of development, it can simplify your work to directly insert data.

- Create a **PreparedStatement** object using the **prepareStatement()** method.

- To execute SQL commands, you must create a **Statement** or **PreparedStatement** object. Use a **Statement** object to execute simple SQL commands without any parameters and a **PreparedStatement** object to execute SQL commands with parameters. The sample program first creates a **PreparedStatement** object to execute an insert command:

  ```
  PreparedStatement pstmt1 =
  conn.prepareStatement( INSERT_PRODUCT );
  ```

  The **prepareStatement** method takes a SQL string as an argument; this SQL string is included from the generated interface.

5 Execute a select SQL command using a Statement object

- Create a **Statement** object using the **createStatement()** method.

  Unlike the **PreparedStatement** object, you do not need to supply a SQL statement when you create a **Statement** object. Therefore, a single **Statement** object can be used to execute more than one SQL statement.

  ```
  Statement stmt = conn.createStatement();
  ```

- Execute your SQL query.

  Use the **executeQuery()** method to execute a select query. A select statement returns a **ResultSet** object.

- Implement a loop to sequentially obtain query results.

  The **ResultSet** object maintains a cursor that initially points just before the first row. The cursor is incremented by one row each time the **next()** method is called. The **next()** method returns a true value when the cursor moves to a row with data and returns a false value when it has moved beyond the last row.

  ```
  while(result.next()) {
  …
  }
  ```

- Retrieve query results using the **get***xxx***()** methods.

  Supply the column number as an argument to these methods. The sample program uses the **getInt()** method to retrieve the product ID and price from the first and second columns respectively, and the **getString()** method to retrieve the product name from the third.

  ```
  int id = result.getInt( 1 );
  int price = result.getInt( 2 );
  String name = result.getString( 3 );
  ```

6 End your Java UltraLite program

- Close the connection to the database, using the **Connection.close()** method:

  ```
  conn.close();
  ```

Lesson 4: Build and run the application

After you have created a source file *Sample.java* using the sample code in the previous section, you are ready to build your UltraLite application.

❖ **To build your application:**

1. Start the Adaptive Server Anywhere personal database server.

   By starting the database server, the UltraLite generator has access to your reference database. Start the database server from the Start menu:

   Start➤Programs➤Sybase SQL Anywhere 8➤UltraLite➤Personal Server Sample for UltraLite.

2. Compile your Java source files.

   Include the following locations in your classpath:

   - The current directory (use a dot in your classpath).
   - The Java runtime classes. For JDK 1.2, include the *jre\lib\rt.jar* file in your classpath. For JDK 1.1, include the *classes.zip* file from your Java installation.
   - The UltraLite runtime classes. These classes are in the following location

     ```
     %ASANY8%\UltraLite\java\lib\ulrt.jar
     ```

     where *%ASANY8%* represents your SQL Anywhere directory.

   Use the *javac* function of the Java development kit as follows:

   ```
   javac *.java
   ```

You are now ready to run your application.

❖ **To run your application:**

1. Go to a command prompt in the *Javatutorial* directory.
2. Include the same classes in the classpath as in the earlier step.
3. Enter the following command to run the application

   ```
   java Sample
   ```

   The list of two items is written out to the screen, and the application terminates.

You have now built and run your first UltraLite Java application. The next step is to add synchronization to the application.

Lesson 5: Add synchronization to your application

Once you have tested that your program is functioning properly, you can remove the lines of code that manually insert data into the *ULProduct* table. Replace these statements with a call to the **JdbcConnection.synchronize()** function to synchronize the remote database with the consolidated database. This process will fill the tables with data and you can subsequently execute a select query.

Adding synchronization actually simplifies the code. Your initial version of *Sample.java* uses the following lines to insert data into your UltraLite database.

```
PreparedStatement pstmt1 = conn.prepareStatement(
ADD_PRODUCT_1 );
   pstmt1.setInt(1, 1);
   pstmt1.setInt(2, 400);
   pstmt1.setString(3, "4x8 Drywall x100");
   int rows1=pstmt1.executeUpdate();
   pstmt1.setInt(1, 2);
   pstmt1.setInt(2, 3000);
   pstmt1.setString(3, "8’ 2x4 Studs x1000");
   int rows2=pstmt1.executeUpdate();
```

This code is included to provide an initial set of data for your application. In a production application, you would not insert an initial copy of your data from source code, but would carry out a synchronization.

❖ To add synchronization to your application:

1. Replace the hard-coded inserts with a synchronization call.
   - Delete the instructions listed above, which insert code.
   - Add the following line in their place:

     ```
     UlSynchOptions synch_opts = new UlSynchOptions();
     synch_opts.setUserName( "50" );
     synch_opts.setPassword( "pwd50" );
     synch_opts.setScriptVersion( "custdb" );
     synch_opts.setStream( new UlSocketStream() );
     synch_opts.setStreamParms( "host=localhost" );
     ( (JdbcConnection)conn ).synchronize( synch_opts );
     ```

     The **ULSocketStream** argument instructs the application to synchronize over TCP/IP, to a MobiLink synchronization server on the current machine (**localhost**), using a MobiLink user name of 50.
2. Compile and link your application.

Enter the following command, with a CLASSPATH that includes the current directory, the UltraLite runtime classes, and the Java runtime classes:

```
javac *.java
```

3 Start the MobiLink synchronization server running against the sample database.

From a command prompt in your *JavaTutorial* directory, enter the following command:

```
start dbmlsrv8 -c "dsn=UltraLite 8.0 Sample"
```

4 Run your application.

From a command prompt in your *JavaTutorial* directory, enter the following command:

```
java Sample
```

The application connects, synchronizes to receive data, and writes out information to the command line. The output is as follows:

```
Connecting to server:port = localhost(a.b.c.d):2439
4x8 Drywall x100         Id=1    Price=400
8′ 2x4 Studs x1000       Id=2    Price=3000
Drywall Screws 10lb      Id=3    Price=40
Joint Compound 100lb     Id=4    Price=75
Joint Tape x25x500       Id=5    Price=100
Putty Knife x25          Id=6    Price=400
8′ 2x10 Supports x 200   Id=7    Price=3000
400 Grit Sandpaper       Id=8    Price=75
Screwmaster Drill        Id=9    Price=40
200 Grit Sandpaper       Id=10   Price=100
```

In this lesson, you have added synchronization to a simple UltraLite application.

For more information on the **JdbcConnection.synchronize()** function, see "Adding synchronization to your application" on page 352.

Lesson 6: Undo the changes you have made

To complete the tutorial, you should shut down the MobiLink synchronization server and restore the UltraLite 8.0 Sample database.

❖ **To finish the tutorial:**

1. Close down the MobiLink synchronization server.
2. Restore the UltraLite 8.0 Sample database.
   - Delete the *custdb.db* and *custdb.log* files in the *Samples\UltraLite\custdb* subdirectory of your SQL Anywhere directory.
   - Copy the *custdb.db* file from your *Javatutorial* directory to the *Samples\UltraLite\custdb* directory.
3. Delete the UltraLite database.
   - The UltraLite database is in the same directory as the jar file, and has a *.udb* extension. The application will initialize a new database next time the application is run.

CHAPTER 15

Developing UltraLite Java Applications

About this chapter

This chapter provides details of the UltraLite development process that are specific to Java. It explains how to write UltraLite applications using Java and provides instructions on building and deploying a Java UltraLite application.

Contents

Before you begin

This chapter assumes that you are familiar with Java programming and JDBC at an elementary level. You can learn about Java from the book *Thinking in Java*, included with SQL Anywhere Studio in PDF format.

Introduction

UltraLite applications can be written in the Java language using JDBC for database access.

The UltraLite development process for Java is similar to that for other development models. For a description, see "Developing UltraLite Applications" on page 67.

This chapter describes only those aspects of application development that are specific to UltraLite Java applications. It assumes an elementary familiarity with Java and JDBC.

The UltraLite Java sample application

This section describes how to compile and run the UltraLite Java version of the CustDB sample application.

The sample application is provided in the *Samples\UltraLite\CustDB\java* subdirectory of your SQL Anywhere directory.

The applet version of the sample uses the Sun appletviewer to view the file *custdb.html*, which contains a simple <APPLET> tag.

The appletviewer security restrictions require the applet to be downloaded from a Web server, rather than to be run from the file system, for socket connections to be permitted and synchronization to succeed.

The application version of CustDB persists its data to a file, while the applet version does not use persistence.

For a walkthrough of the C/C++ version of the application, which has very similar features, see "Tutorial: A Sample UltraLite Application" on page 15.

The UltraLite Java sample files

The code for the UltraLite Java sample application is held in the *Samples\UltraLite\CustDB\java* subdirectory of your SQL Anywhere directory.

The directory holds the following files:

- **Data access code** The file *CustDB.java* holds the UltraLite-specific data access logic. The SQL statements are stored in *SQL.sql*.
- **User interface code** The files *DialogDelOrder.java*, *Dialogs.java*, *DialogNewOrder.java*, and *DialogUserID.java* all hold user interface features.
- **readme.txt** A text file containing detailed, release-dependent information about the sample.
- **Subdirectories** There are two subdirectories in which you can run the sample. These are *java11* (for Java 1) and *java13* (for Java 2). You should make the former your current directory if you are using a 1.1.x version of the JDK, and the latter if you are using 1.2.x or later. These subdirectories contain batch files to run the samples. In each directory, the batch files depend on the JAVA_HOME environment variable, which should be set to the directory containing the JDK. For example:

```
SET JAVA_HOME=c:\jdk1.3.1
```

- **Batch files to build the application** The files *build.bat* and *clean.bat* compile the application and delete all files except the source files, respectively.
- **Files to run the sample as an application** The *Application.java* file contains instructions necessary for running the example as a Java application, and *run.bat* runs the sample application.
- **Files to run the sample as an applet** The *Applet.java* file contains instructions necessary for running the example as a Java applet, and *avapplet.bat* runs the sample applet using the appletviewer, with *custdb.html* as the Web page.

You must install and start a Web server to run the sample as an applet. The applet can be run using the appletviewer utility or by using a Web browser. For more information, see the *Samples\UltraLite\CustDB\Java\readme.txt* file.

Building the UltraLite Java sample

This section describes how to build the UltraLite Java sample application for the Sun Java 1 or 2 environment.

❖ **To build the UltraLite Java sample:**

1. Ensure you have the right JDK.

   You must have JDK 1.1 or JDK 1.3 to build the sample application, and the JDK tools must be in your path.

2. Open a command prompt.
3. Change to the *Samples\UltraLite\CustDB\java\java13* subdirectory of your SQL Anywhere directory, or the *java11* directory if you are using Java 1.
4. Build the sample:

   - Set the JAVA_HOME environment variable. For example:

     ```
     SET JAVA_HOME=c:\jdk1.3.1
     ```

   - From the command prompt, enter the following command:

     ```
     build
     ```

   The build procedure carries out the following operations:

   - Loads the SQL statements into the UltraLite sample database.

This step uses Interactive SQL, the *SQL.sql* file, and relies on the UltraLite 8.0 Sample data source.

- Generates the Java database class **custdb.Database**.

  This step uses the UltraLite generator and the UltraLite 8.0 Sample data source.

- Compiles the Java files.

  This step uses the JDK compiler (*javac*) and *jar* utility.

Running the UltraLite Java sample

You can run the sample application as a Java application or as an applet. In either case, you need to prepare to run the sample by starting the MobiLink synchronization server running on the same machine that the application is running on.

❖ To prepare to run the sample:

- Start the MobiLink synchronization server running on the UltraLite sample database:

  From the Start menu, choose Programs➤Sybase SQL Anywhere 8➤MobiLink➤Synchronization Server Sample.

❖ To run the sample as an application:

1. Open a command prompt in the *Samples\UltraLite\CustDB\java\java13* directory (or the *java11* directory if you are using Java 1).
2. Run the sample:
   - Set the JAVA_HOME environment variable. For example:

     ```
     SET JAVA_HOME=c:\jdk1.3.1
     ```

   - Enter the following command:

     ```
     run
     ```

   The application starts and the Enter ID dialog is displayed.
3. Enter the employee ID.

   Enter an employee ID of 50, and click OK.

   The UltraLite Customer Demonstration window is displayed. If you have run the sample as either an application or applet before, there is data in the database.

4 If there is no data in the database, synchronize.

From the Actions menu, choose Synchronize. The application synchronizes, and the window displays an order.

You can now carry out operations on the data in the database.

For more information on the sample database and the UltraLite features it demonstrates, see "Tutorial: A Sample UltraLite Application" on page 15.

❖ **To run the sample as an applet using appletviewer:**

1 Start a Web server and ensure that the appropriate subdirectory is configured as the default directory for the server, or as one of the virtual directories.

2 Open a command prompt in the *UltraLite\samples\CustDB\java\java13* directory, or *java11* if you are using Java 1.

3 Enter the following command:

```
avapplet
```

The applet starts and a field to enter an employee ID is displayed.

4 Enter the employee ID.

Enter an employee ID of 50, and click OK.

The UltraLite Customer Demonstration window is displayed. The first time you run the sample, there is no data in the database. If you have run the sample as either an application or applet before, there is data in the database.

5 Synchronize the application:

From the Actions menu, choose Synchronize. The application synchronizes, and the window displays an order.

You can now carry out operations on the data in the database.

❖ **To run the sample as an applet using A Web browser:**

1 Start a Web server and ensure that the appropriate subdirectory is configured as the default directory for the server, or as one of the virtual directories.

2 Start a Web browser and enter the URL for the *Samples\UltraLite\CustDB\java\custdb.htm* file into the browser.

The applet starts and a field to enter an employee ID is displayed.

3 Enter the employee ID.

Enter an employee ID of 50, and click OK.

The UltraLite Customer Demonstration window is displayed. The first time you run the sample, there is no data in the database. If you have run the sample as either an application or applet before, there is data in the database.

4 Synchronize the application:

From the Actions menu, choose Synchronize. The application synchronizes, and the window displays an order.

For more information on the sample database and the UltraLite features it demonstrates, see "Tutorial: A Sample UltraLite Application" on page 15.

Resetting the sample

You can delete all compiled files, the sample database, and the generated code by running the *clean.bat* file.

Connecting to and configuring your UltraLite database

This section describes how to connect to an UltraLite database. It describes the recommended UltraLite method for connecting to your database, and also how you can use the standard JDBC connection model to connect.

Connections to UltraLite databases have no user IDs or passwords. For more information, see "User authentication for UltraLite databases" on page 442.

In multi-threaded applications, connections cannot be shared among threads.

UltraLite Java databases can be **persistent** (stored in a file when the application closes) or **transient** (the database vanishes when the application is closed). By default, they are transient.

You configure the persistence of your UltraLite database when connecting to it. This section describes how to configure your UltraLite database.

Using the UltraLite JdbcDatabase.connect method

The generated UltraLite database code is in the form of a class that extends **JdbcDatabase**, which has a **connect** method that establishes a connection.

The following example illustrates typical code, for a generated database class called **SampleDB**:

```
try {
   SampleDB db = new SampleDB();
   java.sql.Connection conn = db.connect();
} catch( SQLException e ){
// error processing here
}
```

The generated database class is supplied on the UltraLite generator command line, using the `-f` option.

If you wish to use a persistent database, the characteristics are specified on the connection as a **Properties** object. The following example illustrates typical code:

```
java.util.Properties p = new java.utils.Properties();
p.put( "persist", "file" );
p.put( "persistfile", "c:\\dbdir\\database.udb" );
SampleDB db = new SampleDB( p );
java.sql.Connection conn = db.connect( );
```

The Properties are used on the database constructor. You cannot change the persistence model of the database between connections.

The two properties specify that the database is persistent, and is stored in the file *c:\dbdir\database.udb*.

For more information on the properties you can specify in the URL, see "UltraLite JDBC URLs" on page 346.

For more information see "Configuring the UltraLite Java database" on page 348, and "The generated database class" on page 373.

Loading and registering the JDBC driver

The UltraLite **JdbcDatabase.connect()** method discussed in the previous section provides the simplest method of connecting to an UltraLite database. However, you can also establish a connection in the standard JDBC manner, and this section describes how to do so.

UltraLite applications connect to their database using a JDBC driver, which is included in the UltraLite runtime classes (*ulrt.jar*). You must load and register the JDBC driver in your application before connecting to the database. Use the **Class.forName()** method to load the driver. This method takes the driver package name as its argument:

```
Class.forName( "ianywhere.ultralite.jdbc.JdbcDriver");
```

The JDBC driver automatically registers itself when it is loaded.

Loading multiple drivers

Although there is typically only one driver registered in each application, you can load multiple drivers in one application. Load each driver using the same methods as above. The **DriverManager** decides which driver to use when connecting to the database.

getDriver method

The **DriverManager.getDriver(url)** method returns the **Driver** for the specified URL.

Error handling

To handle the case where the driver cannot be found, catch **ClassNotFoundException** as follows:

```
try{
   Class.forName(
      "ianywhere.ultralite.jdbc.JdbcDriver");
} catch(ClassNotFoundException e){
   System.out.println( "Exception: " + e.getMessage() );
   e.printStackTrace();
}
```

Connecting to the database using JDBC

Once the driver is declared, you can connect to the database using the standard JDBC **DriverManager.getConnection** method.

getConnection prototypes

The JDBC **DriverManager.getConnection** method has several prototypes. These take the following arguments:

```
DriverManager.getConnection( String url, Properties info )
DriverManager.getConnection( String url )
```

The UltraLite driver supports each of these prototypes. The arguments are discussed in the following sections.

Driver Manager

The **DriverManager** class maintains a list of the **Driver** classes that are currently loaded. It asks each driver in the list if it is capable of connecting to the URL. Once such a driver is found, the **DriverManager** attempts to use it to connect to the database.

Error handling

To handle the case where a connection cannot be made, catch the **SQLException** as follows:

```
try{
   Class.forName(
      "ianywhere.ultralite.jdbc.JdbcDriver");
   Connection conn = DriverManager.getConnection(
      "jdbc:ultralite:asademo" );
   } catch(SQLException e){
   System.out.println( "Exception: " + e.getMessage() );
   e.printStackTrace();
   }
```

UltraLite JDBC URLs

The URL is a required argument to the **DriverManager.getConnection** method used to connect to UltraLite databases.

For an overview of connection methods, see "Connecting to the database using JDBC" on page 345.

The syntax for UltraLite JDBC URLs is as follows:

```
jdbc:ultralite:[database:persist:persistfile][;option=value...]
```

The components are all case sensitive, and have the following meanings:

- **jdbc** Identifies the driver as a JDBC driver. This is mandatory.
- **ultralite** Identifies the driver as the UltraLite driver. This is mandatory.
- **database** The class name for the database. It is required and must be a fully-qualified name: if the database class is in a package, you must include the package name.

For example, the URL `jdbc:ultralite:MyProject` causes a class named `MyProject` to be loaded.

As Java classes share their name with the *.java* file in which they are defined, this component is the same as the output file parameter from the UltraLite generator.

For more information, see "The UltraLite generator" on page 419.

- **persist** Specifies whether or not the database should be persistent. By default, it is transient.

  For more information, see "Configuring the UltraLite Java database" on page 348.

- **persistfile** For persistent databases, specifies the filename.

  For more information, see "Configuring the UltraLite Java database" on page 348. The UltraLite Java properties are very similar to those for C/C++ applications. Their names differ only in the absence of underscore characters., except that **persistfile** is analogous to **file_name**. See "UL_STORE_PARMS macro" on page 428.

- **options** The following options are provided:

  - **uid** A user ID.
  - **pwd** A password for the user ID.

Alternatively, you can connect using a Properties object. The following properties may be specified. Each have the same meaning as in the explicit URL syntax above:

- database
- persist
- persistfile
- user
- password

Using a Properties object to store connection information

You can use a **Properties** object to store connection information, and supply this object as an argument to **getConnection** along with the URL.

For an overview of connection methods, see "Connecting to the database using JDBC" on page 345.

The following components of the URL, described in "UltraLite JDBC URLs" on page 346, can be supplied either as part of the URL or as a member of a **Properties** object.

- **persist**
- **persistfile**

The `jdbc:ultralite` components must be supplied in the URL.

If you wish to encrypt your database, you can do so by supplying a **key** property. For more information, see "Encrypting UltraLite databases" on page 45.

Connecting to multiple databases

UltraLite Java applications can connect to more than one database, unlike UltraLite C/C++ applications.

To connect to more than one database, simply create more than one connection object.

For more information see "Connecting to the database using JDBC" on page 345.

Configuring the UltraLite Java database

You can configure the following aspects of the UltraLite Java database:

- Whether the database is transient or persistent.
- If the database is persistent, you can supply a filename.
- If the database is transient, you can supply a URL for an initializing database.
- You can set an encryption key.

These aspects can be configured by supplying special values in the database URL, or by supplying a Properties object when creating the database. The encryption key cannot be set on the URL, but must be set in a Properties object.

For more information, see "Using the UltraLite JdbcDatabase.connect method" on page 344, and "Using a Properties object to store connection information" on page 347.

Transient and persistent databases

By default, UltraLite Java databases are transient: they are initialized when the database object is instantiated, and vanish when the application closes down. The next time the application starts, it must reinitialize a database.

You can make UltraLite Java databases persistent by storing them in a file. You do this by specifying the **persist** and **persistfile** elements of the JDBC URL, or by supplying **persist** and **persistfile** Properties to the database **connect** method.

Initializing transient databases

The database for C/C++ UltraLite applications is initialized on the first synchronization call. For UltraLite Java applications that use a transient database, there is an alternative method of initializing the database. The URL for an UltraLite database that is used as the initial database is supplied in the **persistfile** component to the URL.

Configuring the database

The database configuration components of the URL are as follows:

- **persist** This can take one of the following values:
  - **none** In this case, the database is transient. It is stored in memory while the application runs, but vanishes once the application terminates.
  - **file** In this case, the database is stored as a file. The default filename is *database.udb*, where *database* is the database class name.

  The default setting is **none**.
- **persistfile** The meaning of this component depends on the setting for **persist**.
  - If the **persist** component has a value of **none**, the **persistfile** component is a URL for an UltraLite database file that is used to initialize the database.

    Both the schema and the data from the URL are used to initialize the application database, but there is no further connection between the two. Any changes made by your application apply only to the transient database, not to the initializing database.

    The following JDBC URL is an example:

```
jdbc:ultralite:transient:none:http://www.address.com/transient.udb
```

    You can prepare the initializing database with an application that uses the persistent form of the URL to create the database, synchronize, and exit.

- If the **persist** component has a value of **file**, the **persistfile** component is a filename for the persistent UltraLite database. The filename should include any extension (such as *.udb*) that you wish to use.

Including SQL statements in UltraLite Java applications

This section describes how to add SQL statements to your UltraLite application.

For information on SQL features that can and cannot be used in UltraLite application, see "SQL features and limitations of UltraLite applications" on page 437.

The SQL statements to be used in your application must be added to the reference database. The UltraLite generator writes out an interface that defines these SQL statements as **public static final strings**. You invoke the statements in your application by implementing the interface and referencing the SQL statement by its identifier, or by referencing it directly from the interface.

Defining the SQL statements for your application

The SQL statements to be included in the UltraLite application, and the structure of the UltraLite database itself, are defined by adding the SQL statements to the reference database for your application.

For information on reference databases, see "Preparing a reference database" on page 72.

Defining projects

Each SQL statement stored in the reference database is associated with a **project**. A project is a name, defined in the reference database, which groups the SQL statements for one application. You can store SQL statements for multiple applications in a single reference database by defining multiple projects.

For information on creating projects, see "Creating an UltraLite project" on page 80.

Adding statements to your project

The data access statements you are going to use in your UltraLite application must be added to your project.

For information on adding SQL statements to your database, see "Adding SQL statements to an UltraLite project" on page 81.

Adding synchronization to your application

This section describes how to initiate synchronization from an UltraLite Java application.

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

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

UltraLite Java applications synchronize in a very similar fashion to other UltraLite applications. For general information about synchronization, see "Adding synchronization to your application" on page 94.

Initializing the synchronization options

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

Before synchronizing, initialize the synchronization parameters as follows:

```
UlSynchOptions opts = new UlSynchOptions();
```

The **UlSynchOptions()** object has a set of methods to set and get its fields. For a list, see "Synchronization parameters" on page 380. Use these methods to set the required synchronization parameters before synchronizing. For example:

```
opts.setUserName( "50" );
opts.setScriptVersion( "default" );
opts.setStream( new UlSocketStream() );
```

Notes

The synchronization streams for UltraLite Java applications are objects, and are set by their constructors. The available streams are as follows:

- **UlSocketStream** TCP/IP synchronization

- **UlSecureSocketStream** TCP/IP synchronization with Certicom elliptic-curve transport-layer security.
- **UlSecureRSASocketStream** TCP/IP synchronization with Certicom RSA transport-layer security.
- **UlHTTPStream** HTTP synchronization.
- **UlHTTPSStream** HTTPS synchronization.

The following line sets the stream to TCP/IP:

```
synch_opts.setStream( new UlSocketStream() );
```

For more information, see "Synchronization parameters" on page 380.

Separately-licensable option required
Use of **UlHTTPSStream, UlSecureSocketStream** and **UlSecureRSASocketStream** require Certicom technology, which in turn 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" on page 4 of the book *Introducing SQL Anywhere Studio.*

Initiating synchronization

Once you have initialized the synchronization parameters, and set them to the values needed for your application, you can initiate synchronization using the **JdbcConnection.synchronize()** method.

The method takes a **UlSynchOptions** object as argument. The set of calls needed to synchronize is as follows:

```
UlSynchOptions opts = new UlSynchOptions;
opts.setUserName( "50" );
opts.setScriptVersion( "default" );
opts.setStream( new UlSocketStream() );
opts.setStreamParms( "host=123.45.678.90" );
conn.synchronize( opts );
```

Using transport-layer security from UltraLite Java applications

For additional security during synchronization, you can use transport-layer security encrypt messages as they pass between UltraLite application and the consolidated database.

☞ For information about encryption technology, see "Transport-Layer Security" on page 283 of the book *MobiLink Synchronization User's Guide*.

Transport-layer security from UltraLite Java client applications uses a separate synchronization stream. You must set up your MobiLink synchronization server as well as your UltraLite client to use this synchronization stream.

Client changes

At the client, you need to choose the **UlSecureSocketStream** or **UlSecureRSASocketStream** synchronization stream, and supply a set of stream parameters. The stream parameters include parameters that control security.

Set the parameter as follows:

```
UlSynchOptions opts = new UlSynchOptions;
opts.setStream(new UlSecureSocketStream() );
opts.setStreamParms( "host=myserver;"
   + "port=2439;"
   + "certificate_company=Sybase Inc.;"
   + "certificate_unit="MEC;"
   + "certificate_name=Mobilink");
// set other options here
conn.synchronize( opts );
```

☞ For details on the stream parameters, see "UlSecureSocketStream synchronization parameters" on page 409.

Setting up the MobiLink server

As the secure synchronization streams for Java applications are separate streams, you must ensure that the MobiLink synchronization server is listening for it. To do this, you must supply the **java_certicom_tls** or **java_rsa_tls** synchronization streams, to match your choice on the client.

The following command line is an example:

```
dbmlsrv8 -x java_certicom_tls(certificate=mycertificate.crt;port=1234)
```

The security parameters for the **java_certicom_tls** and **java_rsa_tls** streams are as follows:

- **certificate** The name of the certificate file that contains the server's identity. This file needs to include the server's certificate, the certificates of all the certificate authorities in the certificate signing chain, and the server's private key.

  The certificate parameter defaults to *sample.crt* for **java_certicom_tls** and *rsaserver.crt* for **java_rsa_tls**, which is the default identity for MobiLink. These files are distributed with SQL Anywhere Studio, in the same directory as the MobiLink server.

- **certificate_password** The password used to encrypt the private key in the certificate file.

The default is the password for the private key in *sample.crt* and *rsaserver.crt*, which is **test**.

Monitoring and canceling synchronization

UltraLite provides the following features for monitoring synchronization:

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

Monitoring synchronization

To monitor synchronization from an UltraLite Java application, you write a class that implements the **UlSynchObserver** interface. This interface contains a single method:

```
void updateSynchronizationStatus( UlSynchStatus status )
```

The overall process for monitoring synchronization is as follows:

- Register your **UlSynchObserver** object using the **UlSynchOptions** class.
- Call the **synchronize()** method to synchronize.
- The **updateSynchronizationStatus** method of your observer class is called whenever the synchronization state changes. The following section describes the synchronization state.

Example

Here is a typical sequence of instructions for synchronization. In this example, the class **MyObserver** implements the **UlSynchObserver** interface:

```
UlSynchObserver observer = new MyObserver ();
UlSynchOptions opts = new UlSynchOptions();
// set options
opts.setUserName( "mluser" );
opts.setPassword( "mlpwd" );
opts.setStream( new UlSocketStream() );
opts.setStreamParms( "localhost" );
opts.setObserver( observer );
opts.setUserData( myDataObject );
// synchronize
conn.synchronize( opts );
```

Implementing the UlSynchObserver interface

In the class that implements **UlSynchObserver**, the **UlSynchStatus** object holds synchronization information. This object is filled by UltraLite with synchronization status information each time your **updateSynchronizationStatus** method is called.

UlSynchStatus methods

The **UlSynchStatus** object has the following methods:

```
int getState()
int getTableCount()
int getTableIndex()
Object getUserData()
UlSynchOptions getSynchOptions()
UlSqlStmt getStatement()
int getErrorCode()
boolean isOKToContinue()
void cancelSynchronization()
```

These methods have the following meanings:

- **getState** Returns a constant indicating the state of the synchronization. The constant is one of the following values:
  - **STARTING** No synchronization actions have yet been taken.
  - **CONNECTING** The synchronization stream has been built, but not yet opened.
  - **SENDING_HEADER** The synchronization stream has been opened, and the header is about to be sent.
  - **SENDING_TABLE** A table is being sent.
  - **RECEIVING_UPLOAD_ACK** An acknowledgement that the upload is complete is being received.
  - **RECEIVING_TABLE** A table is being received.
  - **SENDING_DOWNLOAD_ACK** An acknowledgement that download is complete is being sent.
  - **DISCONNECTING** The synchronization stream is about to be closed.
  - **DONE** Synchronization has completed successfully.
  - **ERROR** Synchronization has completed, but with an error.

  For a description of the synchronization process, see "The synchronization process" on page 24 of the book *MobiLink Synchronization User's Guide*.
- **getTableCount** Returns the number of tables being synchronized. For each table there is a sending and receiving phase.
- **getTableIndex** Returns the current table index for sending and receiving, starting at 0.
- **getUserData** Returns the user data object.
- **getSynchOptions** Returns the **UlSynchOptions** object.

- **getStatement** Returns the statement that called the synchronization. The statement is an internal UltraLite statement, and this method is unlikely to be of practical use, but is included for completion.
- **getErrorCode** When the synchronization state is set to ERROR, this method returns a diagnostic error code.
- **isOKToContinue** This is set to **false** when **cancelSynchronization** is called. Otherwise, it is **true**.
- **cancelSynchronization** The SQL exception SQLE_INTERRUPTED is set, and the synchronization stops as if a communications error had occurred. The observer is *always* called with either the DONE or ERROR state so that it can do proper cleanup.

Example

The following code snippet illustrates a very simple observer:

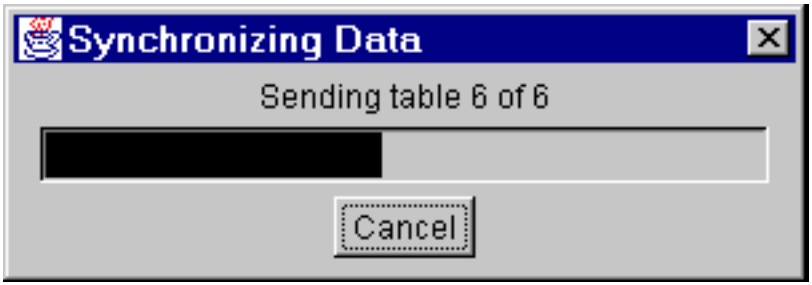
```
void updateSynchronizationStatus( UlSynchStatus status )
{
    int state = status.getState();
    System.out.println( "Sync status: " + state );
    if( state == UlSynchStatus.SENDING_TABLE ||
       state == UlSynchStatus.RECEIVING_TABLE ){
          System.out.println( "send/receive table " +
          ( status.getTableIndex() + 1 ) +
            " of " + status.getTableCount() );
    }
}
```

Using the progress viewer

The UltraLite runtime library includes two progress viewer classes, which provide an implementation of synchronization monitoring, together with the ability for end users to cancel synchronization. The progress viewer classes are as follows:

- **ianywhere.ultralite.ui.SynchProgressViewer** A heavyweight AWT version.
- **ianywhere.ultralite.ui.SynchProgressViewer** A Swing version of the viewer that respects the Swing threading model.

The two classes are used identically. The viewer displays a modal or modeless dialog, which shows a series of messages and a progress bar. Both the messages and the bar are updated during synchronization. The viewer also provides a Cancel button. If the user clicks the Cancel button, synchronization stops and the SQL exception SQLE_INTERRUPTED is thrown.



Threading issues

In a Java application, all events occur on a single thread called the **event thread**. Also, all user interface objects are created on the event thread, even if the application is on a different thread at the time. There is only one event thread in an application.

The event thread must never block. Consequently, you should not perform long operations on the event thread, as this leads to painting aberrations. Even calling the **show()** method on a modal dialog suspends execution of the event thread. You must therefore avoid calling the **synchronize()** method on the event thread.

Displaying a modal viewer

The following code snippet illustrates how to invoke a modal instance of the viewer. The **import** statement uses the AWT version:

```
import ianywhere.ultralite.ui.SynchProgressViewer;
// create a frame to display a dialog
java.awt.Frame frame = ...;
// get UltraLite connection
Connection conn = ...;
// set synchronization options
UlSynchOptions options = new UlSynchOptions();
options.setUserName( "my_user" );
…
// create the viewer
SynchProgressViewer viewer = new SynchProgressViewer( frame );
viewer.synchronize( frame, options );
// execution stops here until synchronization is complete
```

When invoked in this manner, the viewer carries out the following operations:

1. registers itself as a synchronization observer,
2. spawns a thread to do the synchronization,
3. displays itself, blocking the current thread.
4. When synchronization finishes, the observer callback disposes of the dialog, which lets the thread continue.

Displaying a modeless viewer

The following code snippet illustrates how to invoke a modeless instance of the viewer:

```
SynchProgressViewer viewer = new SynchProgressViewer( frame, false );
options.setObserver( viewer );
conn.synchronize( options );
```

In this case, you must ensure that the synchronization occurs on a thread other than the event thread, so that the viewer is not blocked.

Notes

- All messages come from the **SynchProgressViewerResources** resource bundle.
- The viewer implements the **UlSynchObserver** interface so it can hook into the synchronization process.
- The CustDB sample application includes a progress viewer. The CustDB sample code is in the *UltraLite\samples\CustDB\java* subdirectory of your SQL Anywhere directory.

UltraLite Java development notes

This section provides notes for development of UltraLite Java applications.

Creating UltraLite Java applets

If you create your JDBC program as an applet, your application can only synchronize with the machine from which the applet is loaded, which is usually the same as the HTML.

Including an applet in an HTML page

The following is a sample HTML page used to create an UltraLite applet:

```
<html>
   <head>
   </head>
   <body bgcolor="FFFF00">
      <applet code="CustDbApplet.class" width=440
         height=188 archive="custdb.zip,ulrt.jar" >
      </applet>
   </body>
</html>
```

The applet tag specifies the following:

- The class that the applet starts:

  ```
  code="CustDbApplet.class"
  ```

- The size of the window in the web browser to display the applet to.

  ```
  width=440 height=188
  ```

- The zip files that are necessary in order to run the applet.

  ```
  archive="custdb.zip,ulrt.jar"
  ```

  In this case, the *custdb.zip* file and the UltraLite runtime zip file are necessary in order to run the UltraLite CustDB sample application.

Building UltraLite Java applications

This section covers the following subjects:

Generating UltraLite Java classes

When you have prepared a reference database, defined an UltraLite project for your application, and added SQL statements to define your data access features, all the information the generator needs is inside your reference database.

For general information on the UltraLite generator, see "Generating the UltraLite data access code" on page 91. For command-line options, see "The UltraLite generator" on page 419.

The generator output is a Java source file with a filename of your choice. Depending on the design of your database and the sophistication of the database functionality your application requires, this file can vary greatly in both size and content.

There are several ways to customize the UltraLite generator output, depending on the nature of your application.

Overview

You generate the classes by running the UltraLite generator against the reference database.

❖ **To run the UltraLite generator:**

- Enter the following command at a command-prompt:

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

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

Common command-line combinations

When you are generating Java code, there are several options you may want to specify:

- **-t java** Generate Java code. The generator is the same tool used for C/C++ development, so this option is required for all Java use.
- **-i** Some Java compilers do not support inner classes correctly, and so the default behavior of the generator is not to generate Java code that includes inner classes. If you wish to take advantage of a compiler that does support inner classes, use this option.

- **-p** It is common to include your generated classes in a package, which may include other classes from your application. You can use this switch to instruct the generator to include a package name for the classes in the generated files
- **-s** In addition to the code for executing the SQL statements, generate the SQL statements themselves as an interface. Without this option, the strings are written out as members of the database class itself.
- **-a** Make SQL string names upper case. If you choose the –a option, the identifier used in the generated file to represent each SQL statement is derived from the name you gave the statement when you added it to the database. It is a common convention in Java to use upper case letters to represent constants. As the SQL string names are constants in your Java code, you should use this option to generate string identifiers that conform to the common convention.

Example

- The following command (which should be all on one line) generates code that represents the SQL statements in the **CustDemo** project, and the required database schema, with output in the file *uldemo.java*.

```
ulgen -c "dsn=Ultralite 8.0 Sample;uid=DBA;pwd=SQL"
–a -t java -s IStatements CustDemo uldemo.java
```

Compiling UltraLite Java applications

❖ **To compile the generated file:**

1. Set your classpath

   When you compile your UltraLite Java application, the Java compiler must have access to the following classes:

   - The Java runtime classes.
   - The UltraLite runtime classes
   - The target classes (usually in the current directory).

   The following classpath gives access to these classes.

```
%JAVA_HOME%\jre\lib\rt.jar;%ASANY8%\ultralite\java\l
ib\ulrt.jar;.
```

   where JAVA_HOME represents your Java installation directory, and ASANY8 represents your SQL Anywhere installation directory.

   For JDK 1.1 development, *ulrt.jar* is in a *jdk11\lib* subdirectory of the *UltraLite\java* directory.

2. Compile the classes.

With the classpath set as in step one, use *javac* and enter the following command (on a single line):

```
javac file.java
```

The compiler creates the class files for *file.java*.

The compilation step produces a number of class files. You must include *all* the generated *.class* files in your deployment.

Deploying Java applications

Your UltraLite application consists of the following:

- Class files you created to implement your application.
- Generated class files.
- The Java core classes (*rt.jar*).
- UltraLite runtime JAR file (*ulrt.jar*).

Your UltraLite application can be deployed in whatever manner is appropriate. You may wish to package together these class files in a JAR file, for ease of deployment.

Your UltraLite application automatically initializes its own database the first time it is invoked. At first, your database will contain no data. You can add data explicitly using INSERT statements in your application, or you can import data from a consolidated database through synchronization. Explicit INSERT statements are especially useful when developing prototypes.

For more information, see "Adding synchronization to your application" on page 352.

UltraLite API reference

This section describes extensions to the JDBC interface provided by UltraLite, and also describes JDBC features unsupported in UltraLite.

JDBC features in UltraLite

The following are features and limitations specific to the development of JDBC UltraLite applications.

The UltraLite API is modeled on JDBC 1.2, with the addition of the following **ResultSet** methods from JDBC 2.0:

- absolute( ),
- afterLast( ),
- beforeFirst( ),
- first( ),
- isAfterLast( ),
- isBeforeFirst( ),
- isFirst( ),
- isLast( ),
- last( ),
- previous( ),
- relative( )

The following features are incompatible with the UltraLite development model and are not supported by UltraLite.

- There is only limited support for metadata access (system table access). Therefore, you cannot use the **DatabaseMetaData** Interface. Metadata access is limited to the number and type of columns.
- Java objects cannot be stored in the database
- There is no support for stored procedures or stored functions.
- Only static SQL statements are supported and they must be added to the database so that the UltraLite generator can generate them.

Unsupported JDBC methods

UltraLite does not support the following JDBC 1.2 methods. An attempt to use any of the following methods results in a **SQLException** with a vendor code indicating that the feature is not supported in UltraLite.

Connection interface

- getCatalog
- getMetaData
- getTransactionIsolation
- setCatalog
- setTransactionIsolation

ResultSet interface

- getMetaData

Statement interface

- cancel
- getMaxFieldSize
- getMaxRows
- setMaxFieldSize
- setMaxRows

Class JdbcConnection

Package ianywhere.ultralite.jdbc

Description Represents an UltraLite database connection. Most methods are inherited from the JDBC Connection class. Unsupported methods throw an unsupported feature exception.

In a multi-threaded application, each thread must obtain a separate connection. For more information, see "Developing multi-threaded applications" on page 93.

getDefragIterator method

Prototype JdbcDefragIterator **getDefragIterator( )**

Description Initializes and returns a defragmentation iterator.

Parameters **user_name** The MobiLink user name. See "user_name synchronization parameter" on page 397.

password The password associated with user_name. See "password synchronization parameter" on page 384.

script_version The script version. See "version synchronization parameter" on page 397.

stream_defn The stream to use for synchronization. See "stream synchronization parameter" on page 389.

parms Any user-supplied parameters used for the synchronization.

See "stream_parms synchronization parameter" on page 394, and "Synchronization stream parameters" on page 399.

Returns The defragmentation iterator.

Throws java.sql.SQLException

See also "Defragmenting UltraLite databases" on page 51

getLastIdentity method

Prototype long **getLastIdentity( )**

Description Returns the most recent identity value used. This function is equivalent to the following SQL statement:

```
SELECT @@identity
```

The function is particularly useful in the context of global autoincrement columns.

Returns The last identity value.

See also "Determining the most recently assigned value" on page 61
"Global autoincrement default column values" on page 58

globalAutoincUsage method

Prototype short **globalAutoincUsage( )**

Description Returns the maximum global autoincrement counter percentage of all tables in the database. The value is useful when deciding whether to set a database ID.

Returns The percentage of global autoincrement values that have been used.

Throws java.sql.SQLException

See also "Global autoincrement default column values" on page 58
"setDatabaseID method" on page 368

setDatabaseID method

Prototype void **setDatabaseID(** int *value* **)**

Description Sets the

Parameters **value** The integer value to use as the global database identifier.

Throws java.sql.SQLException

See also "Global autoincrement default column values" on page 58
"globalAutoincUsage method" on page 367

synchronize method

Prototype void **synchronize(**
java.lang.String *user_name*,
java.lang.String *password*,
java.lang.String *script_version*,
UlStream *stream_defn*,
java.lang.String *parms* **)**

Description Synchronizes data with a MobiLink synchronization server.

Parameters **user_name** The MobiLink user name. See "user_name synchronization parameter" on page 397.

password The password associated with user_name. See "password synchronization parameter" on page 384.

script_version The script version. See "version synchronization parameter" on page 397.

stream_defn The stream to use for synchronization. See "stream synchronization parameter" on page 389.

parms Any user-supplied parameters used for the synchronization.

See "stream_parms synchronization parameter" on page 394, and "Synchronization stream parameters" on page 399.

Throws java.sql.SQLException

startSynchronizationDelete method

Prototype void **startSynchronizationDelete( )**

Description Restart logging of deletes for MobiLink synchronization

Throws java.sql.SQLException

See also "START SYNCHRONIZATION DELETE statement [MobiLink]" on page 556 of the book *ASA SQL Reference Manual*

stopSynchronizationDelete method

Prototype void **stopSynchronizationDelete( )**

Description Prevent logging of deletes for MobiLink synchronization.

Throws java.sql.SQLException

See also "STOP SYNCHRONIZATION DELETE statement [MobiLink]" on page 563 of the book *ASA SQL Reference Manual*

Class JdbcDatabase

Package ianywhere.ultralite.jdbc

Description The **JdbcDatabase** is used directly only for obfuscating databases. The generated database class extends **JdbcDatabase** and provides an object that represents the UltraLite database. Most JdbcDatabase methods are used from the generated database class.

For more information, see "The generated database class" on page 373.

changeEncryptionKey method

Prototype Connection **changeEncryptionKey( )**

Description Changes the encryption key for an UltraLite database.

Returns A JDBC Connection object.

Throws java.sql.SQLException

See also "Encrypting UltraLite databases" on page 45

close method

Prototype void **close( )**

Description Closes all connections to an UltraLite database. This method must be executed before an UltraLite database can be deleted.

Returns void

Throws java.sql.SQLException

connect method

Prototype

Connection **connect( )**

Connection **connect(** String *user***,** String *password* **)**

Connection **connect(** String *user***,** String *password***,** Properties *info* **)**

Description

Connects to an UltraLite database. The user name and password are checked only when user authentication has been enabled with **JdbcSupport.enableUserAuthentication**.

Parameters

user A user name that can connect to the database.

password A string that must be entered as a password when connecting.

info A **Properties** object holding the user name and password.

Returns

A JDBC Connection object.

Throws

java.sql.SQLException

countUploadRows method

Prototype

long **countUploadRows(** UlSqlStmt *stmt***,** int *mask***,** long *threshold* **)**

Description

Returns the number of rows that need to be uploaded when the next synchronization takes place.

You can use this function to determine if a synchronization is needed.

Parameters

stmt The statement for which the upload rows are to be counted.

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

```
UL_PUB_PUB1 | UL_PUB_PUB2
```

For more information on publication masks, see "publication synchronization parameter" on page 386.

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

Returns

The number of rows to be uploaded.

Throws

java.sql.SQLException

drop method

Prototype void **drop( )**

Description Deletes an UltraLite database file. This method should be used with care, and can be executed only after the JdbcDatabase.close() method has been called.

Returns void

Throws java.sql.SQLException

See also "close method" on page 369

getLastDownloadTimeCalendar method

Prototype java.util.Calendar **getLastDownloadTimeCalendar(** ULSqlStmtint *stmt*, int *mask* **)**

Description Returns the last time changes to the result set of a given statement were downloaded.

Parameters

stmt The statement for which the download time is to be checked.

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

```
UL_PUB_PUB1 | UL_PUB_PUB2
```

For more information on publication masks, see "publication synchronization parameter" on page 386.

Returns The last time the statement was downloaded.

getLastDownloadTimeDate method

Prototype java.util.Date **getLastDownloadTimeDate(** ULSqlStmtint *stmt*, int *mask* **)**

Description Returns the last time changes to the result set of a given statement were downloaded.

Parameters

stmt The statement for which the download time is to be checked.

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

```
UL_PUB_PUB1 | UL_PUB_PUB2
```

☞ For more information on publication masks, see "publication synchronization parameter" on page 386.

Returns The last time the statement was downloaded.

getLastDownloadTimeLong method

Prototype long **getLastDownloadTimeLong(** ULSqlStmt *stmt*, int *mask* **)**

Description Returns the last time changes to the result set of a given statement were downloaded.

Parameters

stmt The statement for which the download time is to be checked.

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

```
UL_PUB_PUB1 | UL_PUB_PUB2
```

☞ For more information on publication masks, see "publication synchronization parameter" on page 386.

Returns The last time the statement was downloaded.

grant method

Prototype void **grant(** String *user***,** String *password* **)**

Description Grants a user name and password permission to connect to an UltraLite database. To take effect, this method requires that user authentication has been enabled with **JdbcSupport.enableUserAuthentication**.

Parameters

user A string that must be entered as a user name when connecting.

password A string that must be entered as a password when connecting.

Returns void.

Throws java.sql.SQLException

revoke method

Prototype void **revoke(** String *user* **)**

Description Revokes permission to connect to an UltraLite database from a user name. To take effect, this method requires that user authentication has been enabled with **JdbcSupport.enableUserAuthentication**.

Parameters **user** The user name that can no longer connect to the database.

Returns void.

Throws java.sql.SQLException

setDefaultObfuscation method

Prototype **setDefaultObfuscation (true | false)**

Description Obfuscates the database

See also "Obfuscating an UltraLite database" on page 46

The generated database class

Description The generated database class extends **JdbcDatabase**. It provides an object that represents the UltraLite database. JdbcDatabase methods are typically used on the generated database class.

Constructor new *database-name*(Properties *props*)

where *database-name* is the name of the generated database class. You can specify the class name using the UltraLite generator `-f` command-line option.

For more information, see "The UltraLite generator" on page 419.

Parameters **props** A Properties object containing some or all of the following items:

- persist
- persistfile
- key

For more information, see "Using a Properties object to store connection information" on page 347.

Class JdbcDefragIterator

Package ianywhere.ultralite.jdbc

Description Provides an object used for explicit defragmentation of the database store.

ulStoreDefragStep method

Prototype boolean **ulStoreDefragStep(** UlConnection *conn* **)**

Description Defragments a portion of an UltraLite database.

Parameters

conn The current connection, as a JdbcConnection object.

Returns

true if successful.

false in unsuccessful.

Throws java.sql.SQLException

See also "STOP SYNCHRONIZATION DELETE statement [MobiLink]" on page 563 of the book *ASA SQL Reference Manual*

Class JdbcSupport

Package ianywhere.ultralite.jdbc

Description A static class that provides methods to enable UltraLite features.

enableUserAuthentication method

Prototype void **enableUserAuthentication( )**

Description Sets the UltraLite database so that user authentication is required to connect to it. Must be called before the database object is created.

Parameters None.

Returns Void.

Throws java.sql.SQLException

See also "Java user authentication example" on page 89

disableUserAuthentication method

Prototype void **disableUserAuthentication( )**

Description Sets the UltraLite database so that user authentication is not required to connect to it. Must be called before the database object is created.

Parameters None.

Returns Void.

Throws java.sql.SQLException

See also "enableUserAuthentication method" on page 374

PART FOUR

Reference

This part provides reference material that applies to more than one of the UltraLite development models.

CHAPTER 16

UltraLite Reference

About this chapter

This chapter provides reference information about the UltraLite utility programs and synchronization parameters.

Contents

Synchronization parameters

The synchronization parameters are grouped into a structure (C/C++) or object (Java) that is provided as an argument in the call to synchronize. The C/C++ structure has the following members, and the Java **UlSynchOptions** object has equivalent access methods.

String parameters are null-terminated strings in C/C++, and **String** objects in Java.

> **Use UL_TEXT around constant strings in C/C++ applications**
> The **UL_TEXT** macro allows 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 supplied as members of the **ul_synch_info** structure so that the compiler handles these parameters correctly.

For C/C++ users, the **ul_synch_info** structure that holds the synchronization parameters is defined in *ulglobal.h* as follows:

```
struct ul_synch_info {
    ul_char *             user_name;
    ul_char *             password;
    ul_char *             new_password;
    ul_char *             version;
    p_ul_stream_defn      stream;
    ul_stream_error       stream_error;
    ul_char *             stream_parms;
    p_ul_stream_defn      security;
    ul_char *             security_parms;
    ul_synch_observer_fn  observer;
    ul_void *             user_data;
    ul_bool               upload_only;
    ul_bool               download_only;
    ul_bool               upload_ok;
    ul_bool               ignored_rows;
    ul_auth_status        auth_status;
    ul_bool               send_download_ack;
    ul_publication_mask   publication;
    ul_bool               send_column_names;
    ul_s_long             auth_value;
    ul_bool               checkpoint_store;
    ul_bool               ping;
    p_ul_synch_info       init_verify;
};
```

The **init_verify** field is reserved for internal use.

auth_status synchronization parameter

Function

Reports the status of MobiLink user authentication. The MobiLink synchronization server provides this information to the client.

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 read-only.

C/C++ usage

After synchronization, the **auth_status** member of **ul_synch_info** holds one of the following values:

Constant	Value	Description
UL_AUTH_STATUS_UNKNOWN	0	Authorization status is unknown, possibly because the connection has not yet synchronized.
UL_AUTH_STATUS_VALID	1000	User ID and password were valid at the time of synchronization.
UL_AUTH_STATUS_VALID_BUT_EXPIRES_SOON	2000	User ID and password were valid at thetime of synchronization but will expire soon.
UL_AUTH_STATUS_EXPIRED	3000	Authorization failed: user ID or password have expired.
UL_AUTH_STATUS_INVALID	4000	Authorization failed: bad user ID or password.
UL_AUTH_STATUS_IN_USE	5000	Authorization failed: user ID is already in use.

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" on page 446 of the book *MobiLink Synchronization User's Guide*.

Access the parameter as follows:

```
ul_synch_info info;
// ...
returncode = info.auth_status;
```

Java usage

Retrieve the authorization status using **UlSynchOptions.getAuthStatus()**.

```
UlSynchOptions opts = new UlSynchOptions;
// set options here
conn.synchronize( opts );
returncode = opts.getAuthStatus();
```

The constants are the same as for C/C++, but prefixed with **UlDefn**.

See also

"Authenticating MobiLink Users" on page 251 of the book *MobiLink Synchronization User's Guide*.

auth_value synchronization parameter

Function

Provides a place to hold return values from custom user authentication synchronization scripts.

Default

The values set by the default MobiLink user authentication mechanism are described in "auth_status synchronization parameter" on page 381

C/C++ usage

Get the parameter as follows:

```
ul_synch_info info;
// ...
returncode = info.auth_value;
```

Java usage

The Java access method is **getAuthValue**.

Get the parameter as follows:

```
UlSynchOptions opts = new UlSynchOptions;
// set other options here
conn.synchronize( opts );
returncode = opts.getAuthValue();
```

See also

"authenticate_user connection event" on page 446 of the book *MobiLink Synchronization User's Guide*
"authenticate_user_hashed connection event" on page 450 of the book *MobiLink Synchronization User's Guide*
"auth_status synchronization parameter" on page 381

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, limited checkpointing is done.

C/C++ usage Set the parameter as follows:

```
ul_synch_info info;
// ...
info.checkpoint_store = ul_true ;
```

Java usage Not used by Java applications.

download_only synchronization parameter

Function Do not upload any changes from the UltraLite database during this synchronization.

Default The parameter is an optional Boolean value, and by default is false.

C/C++ usage Set the parameter as follows:

```
ul_synch_info info;
// ...
info.download_only = ul_true;
```

Java usage The Java access methods are **getDownloadOnly** and **setDownloadOnly**.

Set the parameter as follows:

```
UlSynchOptions opts = new UlSynchOptions;
opts.setDownloadOnly( true );
 // set other options here
conn.synchronize( opts );
```

See also "Including read-only tables in an UltraLite database" on page 78.
"upload_only synchronization parameter" on page 396

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.

new_password synchronization parameter

Function Sets a new MobiLink password associated with the **user_name**.

Default The parameter is optional, and is a string.

C/C++ usage

Set the parameter as follows:

```
ul_synch_info info;
// ...
info.password = UL_TEXT( "myoldpassword" );
info.new_password = UL_TEXT( "mynewpassword" );
```

Java usage

The Java access methods are **getNewPassword** and **setNewPassword**.

Set the parameter as follows:

```
UlSynchOptions opts = new UlSynchOptions;
opts.setUserName( "50" );
opts.setPassword( "mypassword" );
opts.setNewPassword( "mynewpassword" );
// set other options here
conn.synchronize( opts );
```

See also

"Authenticating MobiLink Users" on page 251 of the book *MobiLink Synchronization User's Guide*.

observer synchronization parameter

Function

A pointer to a callback function that monitors synchronization.

The Java access method is **setObserver**, which takes a **UlSynchObserver** object as argument.

See also

"Monitoring and canceling synchronization" on page 98
"user_data synchronization parameter" on page 396

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.

C/C++ usage

Set the parameter as follows:

```
ul_synch_info info;
// ...
info.password = UL_TEXT( "mypassword" );
```

Java usage

The Java access methods are **getPassword** and **setPassword**.

Set the parameter as follows:

```
UlSynchOptions opts = new UlSynchOptions;
opts.setUserName( "50" );
opts.setPassword( "mypassword" );
// set other options here
conn.synchronize( opts );
```

See also "Authenticating MobiLink Users" on page 251 of the book *MobiLink Synchronization User's Guide*.

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.

C/C++ usage Set the parameter as follows:

```
ul_synch_info info;
// ...
info.ping = ul_true;
```

Java usage The Java access method is **setPing**.

Set the parameter as follows:

```
UlSynchOptions opts = new UlSynchOptions;
opts.setUserName( "50" );
opts.setPing( true );
// set other options here
conn.synchronize( opts );
```

See also "-pi option" on page 427 of the book *MobiLink Synchronization User's Guide*

publication synchronization parameter

Function Specifies the publications to be synchronized.

Default If you do not specify a publication, all data is synchronized.

C/C++ usage The UltraLite generator identifies the publications specified on the *ulgen* `-v` command line option as upper case constants with the name UL_PUB_pubname, where pubname is the name given to the -v option. For example, the following command line generates a publication identified by the constant UL_PUB_SALES

```
ulgen -v sales ...
```

When synchronizing, set the publication parameter to a **publication mask**: an OR'd list of publication constants. For example:

```
ul_synch_info info;
// ...
info.publication = UL_PUB_MYPUB1 | UL_PUB_MYPUB2 ;
```

The special publication mask **UL_SYNC_ALL** describes all the tables in the database, whether in a publication or not. The mask **UL_SYNC_ALL_PUBS** describes all tables in publications in the database.

Java usage The UltraLite generator identifies the publications specified on the *ulgen* `-v` command line option as upper case constants with the name UL_PUB_pubname, where pubname is the name given to the -v option. These constants are fields of the generated project class. For example, the following command line generates a publication identified by the constant `salesproject.UL_PUB_SALES`.

```
ulgen -j salesproject -v sales ...
```

When synchronizing, use the **setSynchPublication** method to set the parameter to an OR'd list .

Set the parameter as follows:

```
UlSynchOptions opts = new UlSynchOptions;
opts.setSynchPublication(
  projectname.UL_PUB_MYPUB1 |
  projectname.UL_PUB_MYPUB2 );
// set other options here
conn.synchronize( opts );
```

where *projectname* is the name of the main project class generated by the analyzer.

See also

"The UltraLite generator" on page 419
"Designing sets of data to synchronize separately" on page 76

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" on page 4 of the book *Introducing SQL Anywhere Studio*.

Default

The **security** parameter is null by default, corresponding to no transport-layer security.

C/C++ usage

The following security streams are supported:

- **ULSecureCerticomTLSStream()** Elliptic-curve transport-layer security provided by Certicom.
- **ULSecureRSATLSStream()** RSA transport-layer security provided by Certicom.

For C/C++ applications, the security stream is specified in addition to the synchronization stream. For example, in embedded SQL:

```
ul_synch_info info;
...
info.stream = ULSocketStream();
info.security = ULRSATLSStream();
```

Java usage

To use secure synchronization from UltraLite Java applications, choose a separate stream. For more information, see "Initializing the synchronization options" on page 352.

See also "Transport-Layer Security" on page 283 of the book *MobiLink Synchronization User's Guide*.

security_parms synchronization parameter

Function Sets the parameters required when using transport-layer security. This parameter must be used together with the **security** parameter.

For more information, see "security synchronization parameter" on page 387.

C/C++ 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, in embedded SQL:

```
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 419.

Java usage To use secure synchronization from UltraLite Java applications, choose a separate stream. For more information, see "Initializing the synchronization options" on page 352.

send_column_names synchronization parameter

Function

When **send_column_names** is set to **ul_true** UltraLite sends each column name to the MobiLink synchronization server. By default UltraLite does not send column names.

This parameter is typically used together with the `-za` or `-ze` switch on the MobiLink synchronization server for automatically generating synchronization scripts.

Java usage

This parameter is not available for UltraLite Java applications.

See also

"-za option" on page 402 of the book *MobiLink Synchronization User's Guide*

send_download_ack synchronization parameter

Function

Set this boolean parameter to **false** to instruct the MobiLink synchronization server that the client will not provide a download acknowledgement.

If the client does send download acknowledgement, the MobiLink synchronization server worker thread must wait for the client to apply the download. If the client does not sent a download acknowledgement, the MobiLink synchronization server is freed up sooner for its next synchronization.

stream synchronization parameter

Function

Set the MobiLink synchronization stream to use for synchronization.

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_parms synchronization parameter" on page 394.

Default

The parameter has no default value, and must be explicitly set.

C/C++ usage

For embedded SQL, set the parameter as in the following example:

```
ul_synch_info info;
...
info.stream = ULSocketStream();
```

C++ API usage is as follows:

```
Connection conn;
auto ul_synch_info info;
...
conn.InitSynchInfo( &info );
info.stream = ULSocketStream();
```

When the type of stream requires a parameter, pass that parameter using the **stream_parms** parameter; otherwise, set the **stream_parms** parameter to null.

The following C/C++ stream functions are available, but may not be supported on all target platforms:

Stream	Description
ULActiveSyncStream()	ActiveSync synchronization (Windows CE only). For a list of stream parameters, see "ActiveSync parameters" on page 399.
ULConduitStream()	Synchronize via HotSync or ScoutSync stream (C/C++ only, Palm Computing Platform only). This function is deprecated. You can supply UL_NULL to synchronize via HotSync or ScoutSync. For a list of stream parameters, see "HotSync and ScoutSync parameters" on page 401.
ULHTTPStream()	Synchronize via HTTP. The HTTP stream uses TCP/IP as its underlying transport. UltraLite applications act as Web browsers and the MobiLink synchronization server acts as a Web server. UltraLite applications send POST requests to send data to the server and GET requests to read data from the server. For a list of stream parameters, see "HTTP stream parameters" on page 403.
ULHTTPSStream()	Synchronize via the HTTPS synchronization stream. The HTTPS stream uses SSL or TLS as its underlying protocol. It operates over Internet protocols (HTTP and TCP/IP). The HTTPS stream requires the use of technology supplied by Certicom. Use of Certicom technology requires that you obtain the separately-licensable SQL Anywhere Studio security option and is subject to export regulations. For more information on this option, see "Welcome to SQL Anywhere Studio" on page 4 of the book *Introducing SQL Anywhere Studio*. For a list of stream parameters, see "HTTPS stream parameters" on page 406.

Stream	Description
ULPalmDBStream()	Synchronize via HotSync or ScoutSync stream (C/C++ only, Palm Computing Platform only). This function is deprecated. You can supply UL_NULL to synchronize via HotSync or ScoutSync. For a list of stream parameters, see "HotSync and ScoutSync parameters" on page 401.
ULSocketStream()	Synchronize via TCP/IP. For a list of stream parameters, see "TCP/IP stream parameters" on page 402.

Java usage

The Java access methods are **getStream** and **setStream**. The stream itself is an object, and the stream names differ slightly from the C/C++ versions.

Stream	Description
UlHTTPStream()	HTTP synchronization. For a list of stream parameters, see "HTTP stream parameters" on page 403.
UlHTTPSStream()	HTTPS synchronization. For a list of stream parameters, see "HTTPS stream parameters" on page 406.
UlSecureSocketStream()	TCP/IP or HTTP synchronization with transport-layer security using elliptic curve encryption. For a list of stream parameters, see "UlSecureSocketStream synchronization parameters" on page 409.
UlSecureRSASocketStream()	TCP/IP or HTTP synchronization with transport-layer security using RSA encryption. For a list of stream parameters, see "UlSecureRSASocketStream synchronization parameters" on page 408..
UlSocketStream()	TCP/IP synchronization. For a list of stream parameters, see "TCP/IP stream parameters" on page 402.

Set the parameter as follows:

```
UlSynchOptions opts = new UlSynchOptions;
opts.setStream(new UlSocketStream() );
opts.setStreamParms( "host=myserver;port=2439" );
// set other options here
conn.synchronize( opts );
```

For information on Java synchronization streams, see "Initializing the synchronization options" on page 352.

stream_error synchronization parameter

Function Set a structure to hold communications error reporting information.

Default The parameter has no default value, and must be explicitly set.

Description The **stream_error** field is a structure of type **ul_stream_error**.

```
typedef struct ss_error {
    ss_stream_id         stream_id;
    ss_stream_context    stream_context;
    ss_error_code        stream_error_code;
    asa_uint32           system_error_code;
    rp_char              *error_string;
    asa_uint32           error_string_length;
} ss_error, *p_ss_error;
```

The structure is defined in *sserror.h*, in the *h* subdirectory of your SQL Anywhere directory.

The **ul_stream_error** fields are as follows:

- **stream_id** The network layer reporting the error. This enumeration has the following constants:

  ```
  STREAM_ID_TCPIP
  STREAM_ID_HTTP
  STREAM_ID_CERTICOM_TLS
  STREAM_ID_PALM_CONDUIT
  STREAM_ID_ACTIVESYNC
  ```

- **stream_context** The basic network operation being performed, such as open, read, or write. For details, see *sserror.h*.

- **stream_error_code** The error reported by the stream itself. The **stream_error_code** is of type **ss_error_code**. The stream error codes are all prefixed with STREAM_ERROR_. A write error, for example, is STREAM_ERROR_WRITE.

  For a listing of error numbers, see "MobiLink Communication Error Messages" on page 631 of the book *MobiLink Synchronization User's Guide*. For the error code suffixes, see *sserror.h*.

In this version, to find the constant associated with each number you must count down the number of lines prefixed by DO_STREAM_Error in *sserror.h*. For example, to find the constant for error number 10, you use the tenth DO_STREAM_ERROR entry in *sserror.h*, which is as follows:

```
DO_STREAM_ERROR( WRITE )
```

Th constant associated with this error is therefore STREAM_ERROR_WRITE.

- **stream_error** The network operation being performed (the context) and the error itself as an enumeration constant.
- **stream_error_code** A system-specific error code.
- **error_string** An application-provided error message

C/C++ usage

For embedded SQL, check for SQLE_COMMUNICATIONS_ERROR as follows:

```
ul_char  error_buff[ 100 ];
ul_synch_info info;
...
ULInitSynchInfo( &info );
info.stream_error.error_string = error_buff;
info.stream_string_error.error_length = sizeof(
error_buff );
...
ULSynchronize( &sqlca, &info )
if( SQLCODE == SQLE_COMMUNICATIONS_ERROR ){
    printf( error_buff );
...// more error handling here
```

C++ API usage is as follows:

```
Connection conn;
auto ul_synch_info info;
...
conn.InitSynchInfo( &info );
info.stream_error.error_string = error_buff;
info.stream_error.error_length = sizeof( error_buff );
if( !conn.Synchronize( &synch_info ) ){
    if( SQLCODE == SQLE_COMMUNICATIONS_ERROR ){
        printf( error_buff );
    // more error handline here
```

Java usage

This feature is not available for Java applications.

stream_parms synchronization parameter

Function

Sets parameters to configure the synchronization stream.

A semi-colon separated list of parameter assignments. Each assignment is of the form *keyword=value*, where the allowed sets of keywords depends on the communications protocol.

For more information, see the following sections:

- "HotSync and ScoutSync parameters" on page 401
- "HTTP stream parameters" on page 403
- "ActiveSync parameters" on page 399
- "TCP/IP stream parameters" on page 402

Default

The parameter is optional, is a string, and by default is null.

C/C++ usage

For embedded SQL, set the parameter as follows:

```
ul_synch_info info;
// ...
info.stream_parms= UL_TEXT( "host=myserver;port=2439" );
```

For the C++ API, set the parameter as follows:

```
Connection conn;
auto ul_synch_info info;
...
conn.InitSynchInfo( &info );
info.stream+parms = UL_TEXT( "host=myserver;port=2439" )
```

Java usage

Set the parameter as follows:

```
UlSynchOptions synch_options = new UlSynchOptions();
synch_opts.setStream( new UlSocketStream() );
synch_opts.setStreamParms( "host=myserver;port=2439" );
```

See also

"Synchronization stream parameters" on page 399.

upload_ok synchronization parameter

Function

Reports the status of MobiLink uploads. The MobiLink synchronization server provides this information to the client.

The parameter is read-only.

C/C++ usage

After synchronization, the **upload_ok** member of **ul_synch_info** holds **true** if the upload was successful, and **false** otherwise.

Access the parameter as follows:

```
ul_synch_info info;
// ...
returncode = info.upload_ok;
```

Java usage Retrieve the authorization status using **UlSynchOptions.getAuthStatus()**.

```
UlSynchOptions opts = new UlSynchOptions;
// set options here
conn.synchronize( opts );
returncode = opts.getUploadOK();
```

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.

C/C++ usage Set the parameter to true as follows:

```
ul_synch_info info;
// ...
info.upload_only = ul_true;
```

Java usage The Java access methods are **setUploadOnly** and **getUploadOnly**.

See also "Synchronizing high-priority changes" on page 78
"download_only synchronization parameter" on page 383

user_data synchronization parameter

Function Make application-specific information available to the synchronization observer class (Java) or the synchronization observer callback function (C++ API).

C/C++ usage When implementing the synchronization observer callback function **observer**, you may wish to make application-specific information available. You do this by providing information using **user_data**.

Java usage When implementing the synchronization observer interface **UlSynchObserver**, you may wish to make application-specific information to the synchronization observer class. You do this by providing an object in the **setUserData** method.

See also "observer synchronization parameter" on page 384
"Monitoring and canceling synchronization" on page 356

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.

C/C++ usage Set the parameter as follows:

```
ul_synch_info info;
// ...
info.user_name= UL_TEXT( "uluser" );
```

Java usage The Java access methods are **getUserName** and **setUserName**.

Set the parameter as follows:

```
UlSynchOptions synch_options = new UlSynchOptions();
synch_opts.setUserName( "myname" );
```

See also
"Authenticating MobiLink Users" on page 251 of the book *MobiLink Synchronization User's Guide*.
"The MobiLink user" on page 22 of the book *MobiLink Synchronization User's Guide*.

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.

C/C++ usage Set the parameter as follows:

```
ul_synch_info info;
// ...
info.version = UL_TEXT( "default" );
```

Java usage The Java access methods are **getScriptVersion** and **setScriptVersion**.

Set the parameter as follows:

```
UlSynchOptions synch_options = new UlSynchOptions();
synch_opts.setVersion( "default" );
```

See also

"Script versions" on page 61 of the book *MobiLink Synchronization User's Guide*.

Synchronization stream parameters

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. This section lists the stream parameter values for each stream.

> **Meaning differs for HotSync and ActiveSync**
> For HotSync and ScoutSync synchronization, the meaning of the synchronization stream parameters is different than for other streams. For information, see "HotSync and ScoutSync parameters" on page 401 and "ActiveSync parameters" on page 399.

Setting a stream

For C/C++ applications, the synchronization stream parameters are supplied in the **stream_parms** member of the **ul_synch_info** structure, as a string. The following embedded SQL code is an example for TCP/IP synchronization:

```
ul_synch_info info;
...
info.stream = ULSocketStream();
info.stream_parms = UL_TEXT( "host=myserver" );
```

For 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 a list of synchronization streams and how to set a synchronization stream, see "stream synchronization parameter" on page 389. For syntax information, see "stream_parms synchronization parameter" on page 394.

ActiveSync parameters

The ActiveSync synchronization stream is accessible from C/C++ applications running on Windows CE.

Meaning of synchronization stream parameters

The **stream_parms** values control the connection from the MobiLink ActiveSync provider, running on the desktop machine, to the MobiLink synchronization server.

The **stream_parms** argument has 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**

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_parms** argument for the protocol in use. For the given stream, the *provider_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 snippet 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 402, and "HTTP stream parameters" on page 403.

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:

```
info.stream_parms =
"stream=tcpip;security=certicom_tls{trusted_certificates
=trusted.crt}";
```

For more information, see "CREATE SYNCHRONIZATION USER statement [MobiLink]" on page 335 of the book *ASA SQL Reference Manual*.

HotSync and ScoutSync parameters

To choose HotSync synchronization, 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 on choosing a HotSync synchronization stream, see "Understanding HotSync and ScoutSync synchronization" on page 269.

Meaning of synchronization stream parameters

For HotSync and ScoutSync synchronization, the **stream_parms** values 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 **stream_parms** 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**

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 snippet 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 402, and "HTTP stream parameters" on page 403.

Null value and default settings

If you use HotSync or ScoutSync synchronization, and supply a null value to **stream_parms**, 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_parms** parameter is:

```
{stream=tcpip;host=localhost}
```

For information on registry locations, see "Configuring conduit synchronization" on page 277.

Adding encryption to HotSync and ScoutSync 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:

```
info.stream_parms =
"stream=tcpip;security=certicom_tls{trusted_certificates
=trusted.crt}";
```

For more information, see "CREATE SYNCHRONIZATION USER statement [MobiLink]" on page 335 of the book *ASA SQL Reference Manual*.

TCP/IP stream parameters

The TCP/IP synchronization stream is accessible from C/C++ applications by using the **ULSocketStream()** function , or from Java applications by using the **UlSocketStream** object.

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.
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.
keep_alive	In some circumstances, MobiLink worker threads become unavailable when connections disappear during synchronization. These blocked worker threads are waiting for replies from the MobiLink client. If all worker threads reach this state, MobiLink cannot process synchronizations. Similarly, MobiLink clients can become blocked if the connection disappears. The keep_alive parameter manages liveness. The default is 1 (On). Set the parameter to 0 (Off) to disable liveness checking for this connection.
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 C/C++ applications by using the **ULHTTPStream()** function , or from Java applications by using the **UlHTTPStream** object.

Synchronization stream parameters for the HTTP 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.
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.
keep_alive	In some circumstances, MobiLink worker threads become unavailable when connections disappear during synchronization. These blocked worker threads are waiting for replies from the MobiLink client. If all worker threads reach this state, MobiLink cannot process synchronizations. Similarly, MobiLink clients can become blocked if the connection disappears. The keep_alive parameter manages liveness. The default is 1 (On). Set the parameter to 0 (Off) to disable liveness checking for this connection.
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 80.
proxy_host= *proxy_hostname*	The host name of the proxy server.
proxy_port=	The port number of the proxy server. The default

Parameter	Description
proxy_portnumber	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 C/C++ applications by using the **ULHTTPStream()** function , or from Java applications by using the **UlHTTPStream** 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" on page 4 of the book *Introducing SQL Anywhere Studio*.

Synchronization stream parameters for the HTTPS 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.
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.
keep_alive	In some circumstances, MobiLink worker threads become unavailable when connections disappear during synchronization. These blocked worker threads are waiting for replies from the MobiLink client. If all worker threads reach this state, MobiLink cannot process synchronizations. Similarly, MobiLink clients can become blocked if the connection disappears. The keep_alive parameter manages liveness. The default is 1 (On). Set the parameter to 0 (Off) to disable liveness checking for this connection.
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.

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.
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 RSA encryption is accessed from Java applications as a separate stream, accessed using the **UlSecureRSASocketStream** object. This is different behavior from C/C++ 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" on page 4 of the book *Introducing SQL Anywhere Studio*.

The synchronization parameters for **UlSecureRSASocketStream** are identical to those for **UlSecureSocketStream**. For a complete listing, see "UlSecureSocketStream synchronization parameters" on page 409.

For more information, see "stream synchronization parameter" on page 389, and "Using transport-layer security from UltraLite Java applications" on page 353.

UlSecureSocketStream synchronization parameters

Transport-layer security using elliptic curve encryption is accessed from Java applications as a separate stream, accessed using the **UlSecureSocketStream** object. This is different behavior from C/C++ 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" on page 4 of the book *Introducing SQL Anywhere Studio*.

For more information, see "stream synchronization parameter" on page 389, and "Using transport-layer security from UltraLite Java applications" on page 353.

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.
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.
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.

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 412
"ul_delete_statement system procedure" on page 412

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 412

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 412
"ul_delete_statement system procedure" on page 412

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 412
"ul_add_statement system procedure" on page 411

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 411
"Explicitly assigning segments" on page 265

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 411..

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' )
```

The HotSync conduit installation utility

Function

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

Syntax

dbcond8 [*switches*] *id*

Switch	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.

Switches

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:

```
dbcond8 -n CustDB Syb2
```

The SQL preprocessor

Function

The SQL preprocessor processes a C or C++ program containing embedded SQL, before the compiler is run.

Syntax

sqlpp [*switches*] *SQL-filename* [*output-filename*]

Switch	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" on page 164 of the book *ASA Programming Guide*

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
```

Switches

–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 switch 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

Function

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

Syntax

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

Switch	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

Switch	Description
-s *filename*	Generate a list of SQL strings in an interface definition [Java]
-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 SQL Remote 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 switch.

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
```

Switches

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 switch 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" on page 380, and "Transport-Layer Security" on page 283 of the book *MobiLink Synchronization User's Guide*.

-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 switch.

-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 "Designing synchronization for UltraLite applications" on page 55.

-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 switch 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 switch has no effect when generating Java code.

The UltraLite segment utility

Function

The UltraLite segment utility is for use when building applications for the Palm Computing Platform using the GCC PRC-Tools tool chain.

Syntax

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

Switch	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 utility

Function

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

Description

The UltraLite utility is installed as the following file:

```
%ASANY8%\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 428.

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.

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"$(VCDIR)\include" \
/I"$(ASANY8)\h"

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

where *VCDIR* is your Visual C++ directory and *ASANY8* 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" on page 305

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 45

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" on page 85

UL_ENABLE_GNU_SEGMENTS macro

Function Instructs the compiler to generate multi-segment code for Palm Computing Platform applications using the PRC Tools development environment.

The UL_ENABLE_SEGMENTS macro must also be defined.

Applies to The generated database code.

See also "Enabling multi-segment code generation" on page 264
"UL_ENABLE_SEGMENTS macro" on page 428

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" on page 264

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.

cache_size 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. For example, the following string sets the cache size to 128 kb.

```
#define UL_STORE_PARMS UL_TEXT("cache_size=128k")
```

The default cache size for 16-bit architectures is 8 K. The default cache size for 32-bit architectures is 64 K. The minimum cache size is 4 K.

This parameter does not apply to the Palm Computing Platform.

file_name Defines the full pathname of the file-based persistent store. No substitutions are performed on this value. In addition, you must ensure that this directory exists when **db_init** is called. The directory is not created automatically.

```
#define UL_STORE_PARMS
   UL_TEXT("file_name=\\uldb\\my own name.udb" )
```

Under Windows CE, the filename must include the absolute path. Under other Windows operating systems and VxWorks, the path may either be absolute, or relative to the current directory.

An alias for this parameter is **DBF**.

You must escape any backslash characters in the path.

This parameter does not apply to the Palm Computing Platform.

key Define an encryption key for strong encryption of the database store.

An alias for this parameter is **DBKey**.

For more information, see "Encrypting UltraLite databases" on page 45.

page_size UltraLite databases are stored in pages. I/O operations are carried out a page at a time. The default page size for UltraLite databases is 4kb. You can specify 2 kb pages using the following storage parameters string:

```
#define UL_STORE_PARMS UL_TEXT("page_size=2k")
```

This parameter is ignored when starting an existing database. It can be used on any C/C++ target platform. Setting a page size of 2 kb reduces the maximum number of tables to approximately 500.

palm_allow_backup 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 utility" on page 426.

The following string sets the parameter.

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

reserve_size Reserves file system space for storage of UltraLite persistent data.

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.

Note that **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.

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

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

This example ensures that the persistent store file is at least 2 Mb upon startup.

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

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

"Configuring and managing database storage" on page 45
"Encrypting UltraLite databases" on page 45

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 386
"ULGetLastDownloadTime function" on page 239
"ULCountUploadRows function" on page 234
"UL_SYNC_ALL_PUBS macro" on page 431

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 386
"ULGetLastDownloadTime function" on page 239
"ULCountUploadRows function" on page 234
"UL_SYNC_ALL macro" on page 431

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.

See also "Choosing how to link the runtime library" on page 295.

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 Applications for Windows CE" on page 293.

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" on page 257.

See also "Developing Applications for the Palm Computing Platform" on page 253.

UNDER_VXW macro

Function Use this directive when compiling UltraLite code for VxWorks.

See also "Developing Applications for VxWorks" on page 309.

APPENDIX A

UltraLite Features and Limitations

About this appendix

This background information is provided to help you better understand the features supported by UltraLite databases.

Contents

UltraLite data types

UltraLite supports all Adaptive Server Anywhere data types, with the following exceptions.

- Java data types.
- CHAR(*n*), VARCHAR(*n*), BINARY(*n*), VARBINARY(*n*) data types, where *n* > 2048. You can use LONG VARCHAR and LONG BINARY data types to hold this kind of information.
- The maximum size of LONG VARCHAR and LONG BINARY values is 64 kb.
- Domains (user-defined data types) that include DEFAULT values or CHECK constraints. You can use IMAGE or TEXT data types.

As you design your database, you should also confirm that MobiLink synchronization supports the features you wish to use.

☞ For a list of Adaptive Server Anywhere data types, see "SQL Data Types" on page 51 of the book *ASA SQL Reference Manual*.

SQL features and limitations of UltraLite applications

The following SQL statements can be used in 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 83.

- **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" on page 156 of the book *MobiLink Synchronization User's Guide*.

Limitations

Some features of Adaptive Server Anywhere cannot be used in UltraLite databases. You cannot use the following Adaptive Server Anywhere SQL features in your UltraLite applications:

- **Dynamic SQL** All SQL in UltraLite applications must be known at compile time (**static** SQL), so that the analyzer can generate code to process the statements. You can not include code in your application that generates and executes arbitrary SQL statements. You can, however, use parameterized SQL statements to control the behavior of your statements at run time.

  If you need the capability to execute dynamic SQL, or need other features not present in UltraLite, consider using Adaptive Server Anywhere. Adaptive Server Anywhere is a full-featured database that has a footprint small enough for many mobile and embedded applications.

- **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.
- **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" on page 116 of the book *MobiLink Synchronization User's Guide*.
- **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.
- **System table access** There are no system tables in an UltraLite database.
- **Stored procedures** You cannot call stored procedures or user-defined functions in an UltraLite application.
- **Java in the database** You cannot include Java methods in your queries or make any other use of Java in the database.
- **SQL variables** You cannot use SQL variables in UltraLite applications, including global variables.

  The *@@identity* global variable is an exception, and can be used within UltraLite applications.
- **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.

The SQL error message Feature not available in UltraLite is reported when an UltraLite program attempts to use a SQL statement or feature that is not supported in UltraLite.

For information on other UltraLite limitations, see "UltraLite data types" on page 436, and "Size and number limitations for UltraLite databases" on page 440.

Size and number limitations for UltraLite databases

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 *
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**
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)

* 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.

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

For other limitations, see "UltraLite data types" on page 436, and "SQL features and limitations of UltraLite applications" on page 437.

UltraLite tables must have primary keys

Each table in your 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.

User authentication for UltraLite databases

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.

For information on implementing user IDs, see "Adding user authentication to your application" on page 85.

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.

Index

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z