



UltraLite™ C/C++ User's Guide

Part number: DC50023-01-0902-01
Last modified: October 2004

Copyright © 1989–2004 Sybase, Inc. Portions copyright © 2001–2004 iAnywhere Solutions, Inc. All rights reserved.

No part of this publication may be reproduced, transmitted, or translated in any form or by any means, electronic, mechanical, manual, optical, or otherwise, without the prior written permission of iAnywhere Solutions, Inc. iAnywhere Solutions, Inc. is a subsidiary of Sybase, Inc.

Sybase, SYBASE (logo), AccelaTrade, ADA Workbench, Adaptable Windowing Environment, Adaptive Component Architecture, Adaptive Server, Adaptive Server Anywhere, Adaptive Server Enterprise, Adaptive Server Enterprise Monitor, Adaptive Server Enterprise Replication, Adaptive Server Everywhere, Adaptive Server IQ, Adaptive Warehouse, AnswerBase, Anywhere Studio, Application Manager, AppModeler, APT Workbench, APT-Build, APT-Edit, APT-Execute, APT-Library, APT-Translator, ASEP, AvantGo, AvantGo Application Alerts, AvantGo Mobile Delivery, AvantGo Mobile Document Viewer, AvantGo Mobile Inspection, AvantGo Mobile Marketing Channel, AvantGo Mobile Pharma, AvantGo Mobile Sales, AvantGo Pylon, AvantGo Pylon Application Server, AvantGo Pylon Conduit, AvantGo Pylon PIM Server, AvantGo Pylon Pro, Backup Server, BayCam, Bit-Wise, BizTracker, Certified PowerBuilder Developer, Certified SYBASE Professional, Certified SYBASE Professional Logo, ClearConnect, Client Services, Client-Library, CodeBank, Column Design, ComponentPack, Connection Manager, Convoy/DM, Copernicus, CSP, Data Pipeline, Data Workbench, DataArchitect, Database Analyzer, DataExpress, DataServer, DataWindow, DB-Library, dbQueue, Developers Workbench, Direct Connect Anywhere, DirectConnect, Distribution Director, Dynamic Mobility Model, Dynamo, e-ADK, E-Anywhere, e-Biz Integrator, EC Gateway, ECMAP, ECRTP, eFulfillment Accelerator, Electronic Case Management, Embedded SQL, EMS, Enterprise Application Studio, Enterprise Client/Server, Enterprise Connect, Enterprise Data Studio, Enterprise Manager, Enterprise Portal (logo), Enterprise SQL Server Manager, Enterprise Work Architecture, Enterprise Work Designer, Enterprise Work Modeler, eProcurement Accelerator, eremote, Everything Works Better When Everything Works Together, EWA, E-Whatever, Financial Fusion, Financial Fusion (and design), Financial Fusion Server, Formula One, Fusion Powered e-Finance, Fusion Powered Financial Destinations, Fusion Powered STP, Gateway Manager, GeoPoint, GlobalFIX, iAnywhere, iAnywhere Solutions, ImpactNow, Industry Warehouse Studio, InfoMaker, Information Anywhere, Information Everywhere, InformationConnect, InstaHelp, Intelligent Self-Care, InternetBuilder, iremote, iScript, Jaguar CTS, jConnect for JDBC, KnowledgeBase, Logical Memory Manager, Mail Anywhere Studio, MainframeConnect, Maintenance Express, Manage Anywhere Studio, MAP, M-Business Channel, M-Business Network, M-Business Server, MDI Access Server, MDI Database Gateway, media.splash, Message Anywhere Server, MetaWorks, MethodSet, ML Query, MobicATS, My AvantGo, My AvantGo Media Channel, My AvantGo Mobile Marketing, MySupport, Net-Gateway, Net-Library, New Era of Networks, Next Generation Learning, Next Generation Learning Studio, O DEVICE, OASis, OASis logo, ObjectConnect, ObjectCycle, OmniConnect, OmniSQL Access Module, OmniSQL Toolkit, Open Biz, Open Business Interchange, Open Client, Open Client/Server, Open Client/Server Interfaces, Open ClientConnect, Open Gateway, Open Server, Open ServerConnect, Open Solutions, Optima++, Orchestration Studio, Partnerships that Work, PB-Gen, PC APT Execute, PC DB-Net, PC Net Library, PhysicalArchitect, Pocket PowerBuilder, PocketBuilder, Power Through Knowledge, power.stop, Power++, PowerAMC, PowerBuilder, PowerBuilder Foundation Class Library, PowerDesigner, PowerDimensions, PowerDynamo, Powering the New Economy, PowerJ, PowerScript, PowerSite, PowerSocket, Powersoft, Powersoft Portfolio, Powersoft Professional, PowerStage, PowerStudio, PowerTips, PowerWare Desktop, PowerWare Enterprise, ProcessAnalyst, QAnywhere, Rapport, Relational Beans, RepConnector, Replication Agent, Replication Driver, Replication Server, Replication Server Manager, Replication Toolkit, Report Workbench, Report-Execute, Resource Manager, RW-DisplayLib, RW-Library, S.W.I.F.T. Message Format Libraries, SAFE, SAFE/PRO, SDF, Secure SQL Server, Secure SQL Toolset, Security Guardian, SKILLS, smart.partners, smart.parts, smart.script, SQL Advantage, SQL Anywhere, SQL Anywhere Studio, SQL Code Checker, SQL Debug, SQL Edit, SQL Edit/TPU, SQL Everywhere, SQL Modeler, SQL Remote, SQL Server, SQL Server Manager, SQL Server SNMP SubAgent, SQL Server/CFT, SQL Server/DBM, SQL SMART, SQL Station, SQL Toolset, SQLJ, Stage III Engineering, Startup.Com, STEP, SupportNow, Sybase Central, Sybase Client/Server Interfaces, Sybase Development Framework, Sybase Financial Server, Sybase Gateways, Sybase Learning Connection, Sybase MPP, Sybase SQL Desktop, Sybase SQL Lifecycle, Sybase SQL Workgroup, Sybase Synergy Program, Sybase User Workbench, Sybase Virtual Server Architecture, SybaseWare, Syber Financial, SyberAssist, SybMD, SyBooks, System 10, System 11, System XI (logo), SystemTools, Tabular Data Stream, The Enterprise Client/Server Company, The Extensible Software Platform, The Future Is Wide Open, The Learning Connection, The Model For Client/Server Solutions, The Online Information Center, The Power of One, TotalFix, TradeForce, Transact-SQL, Translation Toolkit, Turning Imagination Into Reality, UltraLite, UltraLite.NET, UNIBOM, Unilib, Uninull, Unisep, Unistring, URK Runtime Kit for UniCode, Versacore, Viewer, VisualWriter, VQL, Warehouse Control Center, Warehouse Studio, Warehouse WORKS, WarehouseArchitect, Watcom, Watcom SQL, Watcom SQL Server, Web Deployment Kit, Web.PB, Web.SQL, WebSights, WebViewer, WorkGroup SQL Server, XA-Library, XA-Server, and XP Server are trademarks of Sybase, Inc. or its subsidiaries.

Certicom and SSL Plus are trademarks and Security Builder is a registered trademark of Certicom Corp. Copyright © 1997–2001 Certicom Corp. Portions are Copyright © 1997–1998, Consensus Development Corporation, a wholly owned subsidiary of Certicom Corp. All rights reserved. Contains an implementation of NR signatures, licensed under U.S. patent 5,600,725. Protected by U.S. patents 5,787,028; 4,745,568; 5,761,305. Patents pending.

All other trademarks are property of their respective owners.

Contents

About This Manual	xi
SQL Anywhere Studio documentation	xii
Documentation conventions	xv
The CustDB sample database	xvii
Finding out more and providing feedback	xviii
I Introduction	1
1 Introduction to UltraLite for C/C++ Developers	3
UltraLite and the C/C++ programming languages	4
System requirements and supported platforms	9
UltraLite C++ Component architecture	10
II Application Development	11
2 Developing Applications Using the UltraLite C++ Component	13
Using the UltraLite namespace	14
UltraLite database schemas	15
Connecting to a database	17
Accessing data using dynamic SQL	21
Accessing data with the Table API	26
Managing transactions	32
Accessing schema information	33
Handling errors	34
Authenticating users	35
Encrypting data	36
Synchronizing data	37
Compiling and linking your application	38
3 Developing Applications Using the Static C++ API	41
Introduction	42
Defining features for your application	43
Connecting to a database	45
Accessing data	46
Authenticating users	47

Encrypting data	49
Synchronizing data	51
Building Static C++ API applications	58
4 Developing Applications Using Embedded SQL	61
Introduction	62
Initializing the SQL Communications Area	64
Connecting to a database	66
Using host variables	68
Fetching data	80
Authenticating users	85
Encrypting data	87
Adding synchronization to your application	89
Building embedded SQL applications	97
5 Common Features of UltraLite C/C++ Interfaces	105
Understanding the SQL Communications Area	106
Combining UltraLite C/C++ interfaces	108
Defragmenting UltraLite databases	110
6 Developing UltraLite Applications for the Palm Computing Platform	113
Introduction	114
Developing UltraLite applications with Metrowerks CodeWarrior	115
Saving state in UltraLite Palm applications	120
Building multi-segment applications	122
Adding HotSync synchronization to Palm applications	125
Adding TCP/IP, HTTP, or HTTPS synchronization to Palm applications	127
Deploying Palm applications	128
7 Developing UltraLite Applications for Windows CE	131
Introduction	132
Building the CustDB sample application	134
Storing persistent data	136
Deploying Windows CE applications	137
Synchronization on Windows CE	140
III Tutorials	145
8 Tutorial: Build an Application Using the C++ Component	147
Introduction	148
Lesson 1: Connect to the database	149

Lesson 2: Insert data into the database	156
Lesson 3: Select the rows from the table	158
Lesson 4: Add synchronization to your application	160
Lesson 5: Deploy to a Windows CE device	162
9 Tutorial: Build an Application Using the Static C++ API	163
Introduction to the UltraLite static C++ API	164
Lesson 1: Getting started	166
Lesson 2: Create an UltraLite database template	167
Lesson 3: Run the UltraLite generator	168
Lesson 4: Write the application source code	169
Lesson 5: Build and run your application	172
Lesson 6: Add synchronization to your application	174
Restore the sample database	176
10 Tutorial: Build an Application Using Embedded SQL	177
Introduction	178
Lesson 1: Configure eMbedded Visual C++	179
Lesson 2: Write an embedded SQL source file	180
Lesson 3: Build the sample embedded SQL UltraLite appli- cation	186
Lesson 4: Add synchronization to your application	187
11 Tutorial: Build an Application Using ODBC	189
Introduction to UltraLite ODBC	190
Lesson 1: Getting started	191
Lesson 2: Create an UltraLite database schema file	193
Lesson 3: Connect to the database	194
Lesson 4: Insert data into the database	197
Lesson 5: Query the database	198
IV API Reference	201
12 UltraLite C/C++ Common API Reference	203
Callback function for ULRegisterErrorCallback	204
Callback function for ULRegisterSchemaUpgradeObserver	206
ULEnableFileDB function	208
ULEnableGenericSchema function (deprecated)	209
ULEnablePalmRecordDB function	210
ULEnableStrongEncryption function	211
ULEnableUserAuthentication function	212
ULRegisterErrorCallback function	213
ULRegisterSchemaUpgradeObserver function	216

ULStoreDefragFini function	218
ULStoreDefragInit function	219
ULStoreDefragStep function	220
Macros and compiler directives for UltraLite C/C++ applications	221
13 UltraLite C++ Component API Reference	227
Class ULSqlca	229
Class ULSqlcaBase	230
Class ULSqlcaWrap	233
Class UltraLite_Connection	234
Class UltraLite_Connection_iface	236
Class UltraLite_Cursor_iface	245
Class UltraLite_DatabaseManager	249
Class UltraLite_DatabaseManager_iface	250
Class UltraLite_DatabaseSchema	252
Class UltraLite_DatabaseSchema_iface	253
Class UltraLite_IndexSchema	256
Class UltraLite_IndexSchema_iface	257
Class UltraLite_PreparedStatement	260
Class UltraLite_PreparedStatement_iface	261
Class UltraLite_ResultSet	263
Class UltraLite_ResultSet_iface	264
Class UltraLite_ResultSetSchema	265
Class UltraLite_RowSchema_iface	266
Class UltraLite_SQLObject_iface	269
Class UltraLite_StreamReader	271
Class UltraLite_StreamReader_iface	272
Class UltraLite_StreamWriter	275
Class UltraLite_Table	276
Class UltraLite_Table_iface	278
Class UltraLite_TableSchema	285
Class UltraLite_TableSchema_iface	286
Class ULValue	291
14 UltraLite Static C++ API Reference	303
C++ API class hierarchy	304
C++ API language elements	305
ULConnection class	306
ULCursor class	319
ULData class	331
ULResultSet class	340
ULTable class	341
Generated result set class	347
Generated statement class	350

Generated table class	352
15 Embedded SQL API Reference	357
db_fini function	359
db_init function	360
db_start_database function	361
db_stop_database function	362
ULActiveSyncStream function	363
ULChangeEncryptionKey function	364
ULClearEncryptionKey function	365
ULCountUploadRows function	366
ULDropDatabase function	367
ULGetDatabaseID function	368
ULGetLastDownloadTime function	369
ULGetSynchResult function	370
ULGlobalAutoincUsage function	372
ULGrantConnectTo function	373
ULHTTPStream function	374
ULHTTPStream function	375
ULIsSynchronizeMessage function	376
ULPalmDBStream function (deprecated)	377
ULPalmExit function (deprecated)	378
ULPalmLaunch function (deprecated)	379
ULResetLastDownloadTime function	380
ULRetrieveEncryptionKey function	381
ULRevokeConnectFrom function	382
ULRollbackPartialDownload function	383
ULSaveEncryptionKey function	384
ULSetDatabaseID function	385
ULSetSynchInfo function	386
ULSocketStream function	387
ULSynchronize function	388
16 UltraLite ODBC API Reference	389
SQLAllocHandle function	391
SQLBindCol function	392
SQLBindParameter function	393
SQLConnect function	394
SQLDescribeCol function	395
SQLDisconnect function	396
SQLEndTran function	397
SQLExecDirect function	398
SQLExecute function	399
SQLFetch function	400

SQLFetchScroll function	401
SQLFreeHandle function	402
SQLGetCursorName function	403
SQLGetData function	404
SQLGetDiagRec function	405
SQLGetInfo function	406
SQLNumResultCols function	407
SQLPrepare function	408
SQLRowCount function	409
SQLSetCursorName function	410
SQLSetConnectionName function	411
SQLSetSuspend function	412
SQLSynchronize function	413

17 Synchronization Parameters Reference 415

Synchronization parameters	417
auth_parms parameter	418
auth_status parameter	419
auth_value synchronization parameter	420
checkpoint_store synchronization parameter	421
disable_concurrency synchronization parameter	422
download_only synchronization parameter	423
keep_partial_download synchronization parameter	424
ignored_rows synchronization parameter	425
new_password synchronization parameter	426
num_auth_parms parameter	427
observer synchronization parameter	428
partial_download_retained synchronization parameter	429
password synchronization parameter	430
ping synchronization parameter	431
publication synchronization parameter	432
resume_partial_download synchronization parameter	433
security synchronization parameter	434
security_parms synchronization parameter	435
send_column_names synchronization parameter	436
send_download_ack synchronization parameter	437
stream synchronization parameter	438
stream_error synchronization parameter	440
stream_parms synchronization parameter	443
upload_ok synchronization parameter	444
upload_only synchronization parameter	445
user_data synchronization parameter	446
user_name synchronization parameter	447

version synchronization parameter	448
Index	449



About This Manual

Subject	This manual describes UltraLite C and C++ programming interfaces. With UltraLite you can develop and deploy database applications to handheld, mobile, or embedded devices.
Audience	This manual is intended for C and C++ application developers who wish to take advantage of the performance, resource efficiency, robustness, and security of an UltraLite relational database for data storage and synchronization.

SQL Anywhere Studio documentation

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

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

-
- ◆ **SQL Anywhere Studio Security Guide** This book provides information about security features in Adaptive Server Anywhere databases. Adaptive Server Anywhere 7.0 was awarded a TCSEC (Trusted Computer System Evaluation Criteria) C2 security rating from the U.S. Government. This book may be of interest to those who wish to run the current version of Adaptive Server Anywhere in a manner equivalent to the C2-certified environment.
 - ◆ **MobiLink Administration Guide** This book describes how to use the MobiLink data synchronization system for mobile computing, which enables sharing of data between a single Oracle, Sybase, Microsoft or IBM database and many Adaptive Server Anywhere or UltraLite databases.
 - ◆ **MobiLink Clients** This book describes how to set up and synchronize Adaptive Server Anywhere and UltraLite remote databases.
 - ◆ **MobiLink Server-Initiated Synchronization User's Guide** This book describes MobiLink server-initiated synchronization, a feature of MobiLink that allows you to initiate synchronization from the consolidated database.
 - ◆ **MobiLink Tutorials** This book provides several tutorials that walk you through how to set up and run MobiLink applications.
 - ◆ **QAnywhere User's Guide** This manual describes MobiLink QAnywhere, a messaging platform that enables the development and deployment of messaging applications for mobile and wireless clients, as well as traditional desktop and laptop clients.
 - ◆ **iAnywhere Solutions ODBC Drivers** This book describes how to set up ODBC drivers to access consolidated databases other than Adaptive Server Anywhere from the MobiLink synchronization server and from Adaptive Server Anywhere remote data access.
 - ◆ **SQL Remote User's Guide** This book describes all aspects of the SQL Remote data replication system for mobile computing, which enables sharing of data between a single Adaptive Server Anywhere or Adaptive Server Enterprise database and many Adaptive Server Anywhere databases using an indirect link such as e-mail or file transfer.
 - ◆ **SQL Anywhere Studio Help** This book includes the context-sensitive help for Sybase Central, Interactive SQL, and other graphical tools. It is not included in the printed documentation set.
 - ◆ **UltraLite Database User's Guide** This book is intended for all UltraLite developers. It introduces the UltraLite database system and provides information common to all UltraLite programming interfaces.

-
- ◆ **UltraLite Interface Guides** A separate book is provided for each UltraLite programming interface. Some of these interfaces are provided as UltraLite components for rapid application development, and others are provided as static interfaces for C, C++, and Java development.

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

Documentation formats SQL Anywhere Studio provides documentation in the following formats:

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

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

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

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

The PDF books are accessible from the online books, or from the Windows Start menu.

- ◆ **Printed books** The complete set of books is available from Sybase sales or from eShop, the Sybase online store, at <http://eshop.sybase.com/eshop/documentation>.

Documentation conventions

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

Syntax conventions

The following conventions are used in the SQL syntax descriptions:

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

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

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

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

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

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

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

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

```
RELEASE SAVEPOINT [ savepoint-name ]
```

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

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

```
[ ASC | DESC ]
```

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

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

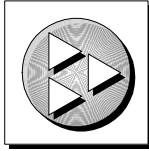
```
[ QUOTES { ON | OFF } ]
```

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

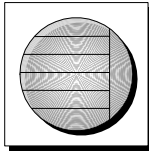
Graphic icons

The following icons are used in this documentation.

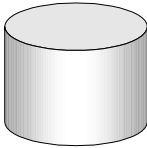
- ◆ A client application.



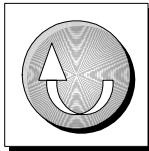
- ◆ A database server, such as Sybase Adaptive Server Anywhere.



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



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



- ◆ A programming interface.



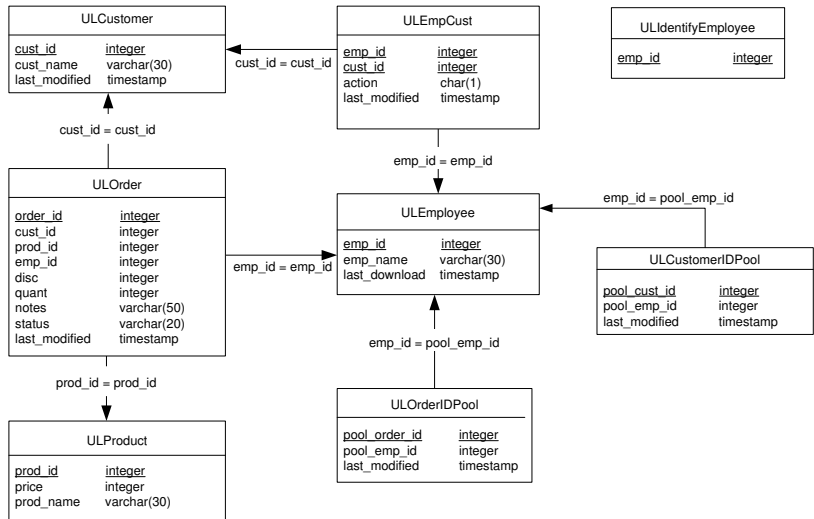
The CustDB sample database

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

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

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

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



Finding out more and providing feedback

Finding out more

Additional information and resources, including a code exchange, are available at the iAnywhere Developer Network at <http://www.ianywhere.com/developer/>.

If you have questions or need help, you can post messages to the iAnywhere Solutions newsgroups listed below.

When you write to one of these newsgroups, always provide detailed information about your problem, including the build number of your version of SQL Anywhere Studio. You can find this information by typing **dbeng9 -v** at a command prompt.

The newsgroups are located 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](#)
- ◆ [ianywhere.public.sqlanywhere.qanywhere](#)

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 ensure its operation and availability.

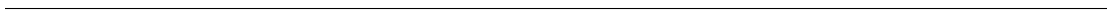
iAnywhere Solutions Technical Advisors as well as other staff assist on the newsgroup service when they have time available. They offer their help on a volunteer basis and may not be available on a regular basis to provide solutions and information. Their ability to help is based on their workload.

Feedback

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

You can e-mail comments and suggestions to the SQL Anywhere documentation team at iasdoc@ianywhere.com. Although we do not reply to e-mails sent to that address, we read all suggestions with interest.

In addition, you can provide feedback on the documentation and the software through the newsgroups listed above.



PART I

INTRODUCTION

This part introduces UltraLite for C/C++ programmers.

CHAPTER 1

Introduction to UltraLite for C/C++ Developers

About this chapter

This chapter introduces you to UltraLite for C/C++. It assumes that you are familiar with the features of UltraLite, as described in [“Welcome to UltraLite”](#) [*UltraLite Database User’s Guide*, page 3].

☞ For more information about creating applications using UltraLite for C/C++, see [“Common Features of UltraLite C/C++ Interfaces”](#) on page 105.

☞ For a hands-on tutorial introducing UltraLite for C/C++, see [“Tutorial: Build an Application Using the C++ Component”](#) on page 147.

Contents

Topic:	page
UltraLite and the C/C++ programming languages	4
System requirements and supported platforms	9
UltraLite C++ Component architecture	10

UltraLite and the C/C++ programming languages

C and C++ provides the following benefits for UltraLite developers targeting small devices:

- ◆ a robust relational database store
- ◆ the power, efficiency, and flexibility of C or C++
- ◆ deployment on the Windows CE, Palm OS, and Windows XP platforms

☞ For more information about the features of UltraLite databases, see [“UltraLite Databases” \[UltraLite Database User’s Guide, page 27\]](#).

UltraLite programming interfaces are either static or dynamic. Static and dynamic interfaces have different benefits and limitations. They also have a different development model.

UltraLite developers using C++ have several options available:

- ◆ The UltraLite C++ Component.
- ◆ Embedded SQL (static interface).
- ◆ The Static C++ API (static interface).
- ◆ The ODBC programming interface (component interface).

UltraLite developers using C must use embedded SQL or ODBC.

☞ For more information, see [“Choosing between components and static interfaces” \[UltraLite Database User’s Guide, page 11\]](#).

Benefits and limitations of the C++ Component

The benefits and disadvantages of the UltraLite C++ Component are as follows:

- ◆ **Dynamic SQL** The UltraLite C++ Component broadens the scope of the applications you can build by providing access to dynamic SQL. Dynamic SQL permits SQL statements to be defined at runtime. By contrast, the static interfaces, such as embedded SQL and the Static C++ API, require all SQL statements to be specified at compile time.
- ◆ **Development model** You can use the UltraLite Schema Painter to create an UltraLite database when you use the UltraLite C++ Component. Also, no preprocessing is required. By contrast, the embedded SQL and Static C++ API interfaces require a preprocessing step before compiling your application. They also require that you construct an Adaptive Server Anywhere reference database.

- ◆ **Efficiency** For databases with few tables, applications built with the static interfaces provide a smaller footprint than those built using the component interfaces. For databases with more complex schemas, this advantage of the static interfaces is lost.
- ◆ **Multi-process support** The UltraLite C++ Component provides a version of the UltraLite runtime engine that supports connections from multiple applications.

Advanced developers can combine features of the UltraLite C++ Component API with those of the Static C++ API and even embedded SQL.

☞ For more information, see [“Combining UltraLite C/C++ interfaces” on page 108](#).

Benefits and limitations of the Static C++ API

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

The Static C++ API has the following advantages:

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

The Static C++ API has the following disadvantages:

- ◆ **Knowledge of C or C++ required** If you are not familiar with C or C++ programming, you may wish to use one of the other UltraLite interfaces. UltraLite components provide interfaces from several popular programming languages and tools.
- ◆ **Complex development model** The use of a reference database to hold the UltraLite database schema, together with the need to generate the API for your specific application, makes the Static C++ API development

process complex. The UltraLite components provide a much simpler development process.

- ◆ **SQL must be specified at design time** Only SQL statements defined at compile time can be included in your application. The UltraLite components allow dynamic use of SQL statements.

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

Developing Static C++ applications

When developing Static C++ UltraLite applications, you use a programming interface that is generated from a reference database. In order to develop these applications you should be familiar with the C or C++ programming language.

The development process for Static C++ UltraLite applications is as follows:

1. Design your database.
Prepare an Adaptive Server Anywhere reference database that contains the tables and indexes you wish to include in your UltraLite database.
2. Add SQL statements to the database.
The SQL Statements you wish to use in your application must be added to the reference database.
3. Generate the API for your application.
The UltraLite generator provides an API for your specific application.
4. Write your application.
Data access features in your application code use function calls from the generated API.
5. Compile your .cpp files.
You can compile the generated .cpp files just as you compile other .cpp files.
6. Link the .cpp files.
You must link the files against the UltraLite runtime library.

☞ For a full description of the development process, see [“Building Static C++ API applications” on page 58](#).

Benefits and limitations of embedded SQL

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

Embedded SQL has the following advantages:

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

Embedded SQL has the following disadvantages:

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

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

Developing embedded SQL applications

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

applications you should be familiar with the C or C++ programming language.

The development process for embedded SQL applications is as follows:

1. Design your database.

Prepare an Adaptive Server Anywhere reference database that contains the tables and indexes you wish to include in your UltraLite database.

2. Write your source code in an embedded SQL source file, which typically has extension `.sql`.

When you need data access in your source code, use the SQL statement you wish to execute, prefixed by the EXEC SQL keywords. For example:

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

3. Preprocess the `.sql` files.

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

4. Compile your `.c` or `.cpp` files.

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

5. Link the `.c` or `.cpp` files.

You must link the files against the UltraLite runtime library.

☞ For a full description of the embedded SQL development process, see [“Building embedded SQL applications” on page 97](#).

System requirements and supported platforms

Development platforms To develop applications using UltraLite C++, you require the following.

- ◆ Microsoft Windows NT/2000/XP.
- ◆ A supported C/C++ compiler.

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

- ◆ Windows CE 3.0 and higher, or Palm OS 3.5 or later.

☞ For more information, see [“UltraLite development platforms”](#) [*Introducing SQL Anywhere Studio*, page 99] and [“UltraLite target platforms”](#) [*Introducing SQL Anywhere Studio*, page 109].

UltraLite C++ Component architecture

The UltraLite C++ Component interface is defined in the *uliface.h* header file.

- ◆ **DatabaseManager** You create one DatabaseManager object for each application.
- ◆ **Connection** Each Connection object represents a connection to an UltraLite database. You can create one or more Connection objects.
- ◆ **Table** The Table object provides access to the data in the database.
- ◆ **PreparedStatement, ResultSet, and ResultSetSchema** These dynamic SQL objects allow you to create Dynamic SQL statements, make queries and execute INSERT, UPDATE and DELETE statements, and attain programmatic control over database result sets.
- ◆ **SyncParms** You use the SyncParms object to synchronize your UltraLite database with a MobiLink synchronization server.

The API Reference is supplied in the online books. For more information about accessing the API reference, see [“UltraLite C++ Component API Reference” on page 227](#).

PART II

APPLICATION DEVELOPMENT

This part provides development notes for UltraLite C/C++ programmers.

CHAPTER 2

Developing Applications Using the UltraLite C++ Component

About this chapter

This chapter explains how to develop applications using the UltraLite C++ component.

☞ For hands-on tutorials, see [“Tutorial: Build an Application Using the C++ Component”](#) on page 147.

Contents

Topic:	page
Using the UltraLite namespace	14
UltraLite database schemas	15
Connecting to a database	17
Accessing data using dynamic SQL	21
Accessing data with the Table API	26
Managing transactions	32
Accessing schema information	33
Handling errors	34
Authenticating users	35
Encrypting data	36
Synchronizing data	37
Compiling and linking your application	38

Using the UltraLite namespace

The UltraLite C++ Component interface provides a set of classes with names prefixed by `UltraLite_`, such as the `UltraLite_Connection` and `UltraLite_DatabaseManager` class. Most of the functions for each of these classes implement a function from an underlying interface with the string `_iface` appended to it. For example, the `UltraLite_Connection` class implements functions from `UltraLite_Connection_iface`.

When you explicitly use the UltraLite namespace, you can use a shorter name to refer to each class. Instead of declaring a connection as an `UltraLite_Connection` object, you can declare it as a `Connection` object if you are using the UltraLite namespace:

```
using namespace UltraLite;
ULSqlca sqlca;
sqlca.Initialize();
DatabaseManager * dbMgr = ULSqlca.DatabaseManager(sqlca);
Connection * conn = UL_NULL;
```

As a result of this architecture, code samples in this chapter use types such as `DatabaseManager`, `Connection`, and `TableSchema`, but links for more information direct you to `UltraLite_DatabaseManager_iface`, `UltraLite_Connection_iface`, and `UltraLite_TableSchema_iface`, respectively.

UltraLite database schemas

The database schema is a description of the database. It is the collection of tables, indexes, keys, and publications within the database, and all the relationships between them.

You do not alter the schema of an UltraLite database directly. Instead, you create a schema (.usm) file and upgrade the database schema from that file by calling a built-in UltraLite function in your application.

A schema file is also used in the initial creation of a database to specify the structure of the database.

Creating UltraLite database schema files

You can create an UltraLite schema file using the UltraLite Schema Painter or the *ulinit* utility.

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

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

☞ For more information about using the UltraLite Schema Painter, see “Lesson 1: Create an UltraLite database schema” [*UltraLite Database User’s Guide*, page 130].

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

☞ For more information about using the *ulinit* utility, see “The ulinit utility” [*UltraLite Database User’s Guide*, page 112].

Upgrading your database schema

To modify your existing database structure, use the `UpgradeSchemaFromFile` method. In most cases there will be no data loss, however, data loss can occur if columns are deleted or if the data type for a column is changed to an incompatible type.

Example

The following code applies a new schema file.

```
// issue a commit before applying a new schema
// if there are any uncommitted transactions
conn->Commit();
conn->UpgradeSchemaFromFile(
    UL_TEXT("schema_file=genup01b.usm") );
```

➔ For more information, see [“UpgradeSchemaFromFile Function” on page 244](#).

Connecting to a database

Using the Connection object

UltraLite applications must connect to a database before carrying out operations on the data in it. This section describes how to connect to an UltraLite database. You can find sample code in *Samples\UltraLite\CustDB*.

The following properties of the Connection object govern global application behavior.

- ◆ **Commit behavior** In the UltraLite C++ component, there is no autocommit mode. Each transaction must be followed by a Connection.Commit statement.
 - ☞ For more information, see [“Managing transactions” on page 32](#).
 - ◆ **User authentication** You can change the user ID and password for the application from the default values of DBA and SQL by using methods to Grant and Revoke connection permissions. Each application can have a maximum of four user IDs.
 - ☞ For more information, see [“User authentication in UltraLite” \[UltraLite Database User’s Guide, page 40\]](#) and [“Authenticating users” on page 35](#).
 - ◆ **Synchronization** A set of objects governing synchronization is accessed from the Connection object.
 - ☞ For more information, see [“Synchronizing data” on page 37](#).
 - ◆ **Tables** UltraLite tables are accessed using methods of the Connection object.
 - ☞ For more information, see [“Accessing data with the Table API” on page 26](#).
 - ◆ **Prepared statements** A set of objects is provided to handle the execution of dynamic SQL statements and to navigate result sets.
 - ☞ For more information, see [“Accessing data using dynamic SQL” on page 21](#).
- ☞ For more information about the Connection object, see [“Class UltraLite_Connection” on page 234](#).

❖ To connect to an UltraLite database

1. Use the UltraLite namespace.

Using the UltraLite namespace allows you to use simple names for classes in the C++ Component interface.

```
using namespace UltraLite;
```

2. Create and initialize a DatabaseManager object and an UltraLite SQL communications area (ULSqlca). The ULSqlca is a structure that handles communication between the application and the database.

The DatabaseManager object is at the root of the object hierarchy. You create only one DatabaseManager object per application. It is often best to declare the DatabaseManager object as global to the application.

```
ULSqlca sqlca;  
sqlca.Initialize();  
DatabaseManager * dbMgr = ULSqlca.DatabaseManager(sqlca);
```

☞ For more information, see [“Class UltraLite_DatabaseManager_iface” on page 250](#).

On the Palm OS, you should precede the call to ULSqlca.DatabaseManager by a call to ULSqlca.EnableFileDB or ULSqlca.EnablePalmRecordDB. This sets the database store to use the Palm data store or the virtual file system. For more details, see [“ULSqlca.EnablePalmRecordDB function” on page 210](#) and [“ULSqlca.EnableFileDB function” on page 208](#).

3. Declare a Connection object.

Most applications use a single connection to an UltraLite database and leave the connection open. Multiple connections are only required for multi-threaded data access. For this reason, it is often best to declare the Connection object as global to the application.

```
Connection * conn = UL_NULL;
```

☞ For more information, see [“Class UltraLite_Connection_iface” on page 236](#)

4. Open a connection to an existing database, or create a new database if the specified database file does not exist.

Most UltraLite applications deploy a schema file rather than a database file, and let UltraLite create the database file on the first connection attempt. Thus, the following code attempts to connect to an existing database. If the database file does not exist, the application creates a database file.

```

// specify the location of the database file
static ul_char * parms = "file_name=mydata.udb";
ULValue lp( parms );
conn = dbMgr->OpenConnection( sqlca, lp );
if( sqlca.GetSQLCode() ==
    SQLE_ULTRALITE_DATABASE_NOT_FOUND ) {
    static ul_char * parms = UL_TEXT("schema_file=mydata.usm")
    UL_TEXT(";file_name=mydata.udb");
    ULValue lp( parms );
    conn = dm->CreateAndOpenDatabase( sqlca, lp );
    if( sqlca.GetSQLCode() < SQLE_NOERROR ){
        printf( "Open failed with sql code: %d.\n" ,
            sqlca.GetSQLCode() );
    }
}
}

```

☞ For more information, see “[OpenConnection Function](#)” on page 251 and “[CreateAndOpenDatabase Function](#)” on page 250.

Example

The following code opens a connection to an UltraLite database named *mydata.udb*.

```

#include "uliface.h"
using namespace UltraLite;
static ul_char * parms =
UL_TEXT( ";file_name=tutcustomer.udb" )
UL_TEXT( ";schema_file=tutcustomer.usm" );
ULSqlca sqlca;
DatabaseManager * dm = UL_NULL;
Connection * conn = UL_NULL;
sqlca.Initialize();
dm = ULInitDatabaseManager( sqlca );
if( dm == UL_NULL ){
    // You may have mismatched UNICODE vs. ANSI runtimes.
    return 1;
}
ULValue lp( parms );
conn = dm->OpenConnection( sqlca, lp );
if( sqlca.GetSQLCode() ==
    SQLE_ULTRALITE_DATABASE_NOT_FOUND ) {
    conn = dm->CreateAndOpenDatabase( sqlca, lp );
    if( sqlca.GetSQLCode() < SQLE_NOERROR ) {
        printf( "Open failed with sql code: %d.\n" ,
            sqlca.GetSQLCode() );
        return NULL;
    } else {
        printf( "Connected to a new database.\n" );
    }
} else {
    printf( "Connected to an existing database.\n" );
}
return( conn );

```

Multi-threaded
applications

Each Connection and all objects created from it should be used on a single

thread. If your application requires multiple threads accessing the UltraLite database, each thread requires a separate connection.

Accessing data using dynamic SQL

UltraLite applications can access table data using dynamic SQL or the Table API. This section describes data access using dynamic SQL.

☞ For information about using the Table API, see [“Accessing data with the Table API” on page 26](#).

This section explains how to perform the following tasks using dynamic SQL.

- ◆ Inserting, deleting, and updating rows.
- ◆ Retrieving rows to a result set.
- ◆ Scrolling through the rows of a result set.

☞ This section does not describe the SQL language itself. For information about dynamic SQL features, see [“Dynamic SQL” \[UltraLite Database User’s Guide, page 159\]](#).

☞ For an overview of the sequence of operations required for any SQL operation, see [“Using dynamic SQL” \[UltraLite Database User’s Guide, page 161\]](#).

Data manipulation: INSERT, UPDATE and DELETE

With UltraLite, you can perform SQL Data Manipulation Language operations. These operations are performed using the `ExecuteStatement` method, a member of the `PreparedStatement` class.

☞ For more information, see the [“Class UltraLite_PreparedStatement” on page 260](#).

Using parameters in your prepared statements

UltraLite indicates query parameters using the `?` character. For any INSERT, UPDATE or DELETE, each `?` is referenced according to its ordinal position in the prepared statement. For example, the first `?` is referred to as 1, and the second as 2.

❖ To INSERT a row

1. Declare a `PreparedStatement`.

```
PreparedStatement * prepStmt;
```

☞ For more information, see [“PrepareStatement Function” on page 241](#).

2. Assign a SQL statement to the `PreparedStatement` object.

```
ULValue sqltext(
    "INSERT INTO MyTable(MyColumn) values (?)" );
prepStmt = conn->PrepareStatement( sqltext );
```

3. Assign input parameter values for the statement.

The following code shows a string parameter.

```
ULValue newValue( "string-value" );
prepStmt->SetParameter( 1, newValue );
```

4. Execute the statement.

The return value indicates the number of rows affected by the statement.

```
ul_s_long rowsInserted;
rowsInserted = prepStmt->ExecuteStatement();
```

5. Commit the change.

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

❖ To UPDATE a row

1. Declare a PreparedStatement.

```
PreparedStatement * prepStmt;
```

2. Assign a statement to the PreparedStatement object.

```
ULValue sqltext(
    "UPDATE MyTable SET MyColumn1 = ? WHERE MyColumn2 = ?" );
prepStmt = conn->PrepareStatement( sqltext );
```

3. Assign input parameter values for the statement.

```
ULValue newValue( new-value );
ULValue oldValue( old-value );
stmt->SetParameter( 1, newValue );
stmt->SetParameter( 2, oldValue );
```

4. Execute the statement.

```
long rowsUpdated = prepStmt->ExecuteStatement();
```

5. Commit the change.

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

❖ **To DELETE a row**

1. Declare a PreparedStatement.

```
PreparedStatement * prepStmt;
```

2. Assign a statement to the PreparedStatement object.

```
ULValue sqltext(  
    "DELETE FROM MyTable WHERE MyColumn = ?" );  
prepStmt = conn->PrepareStatement( sqltext );
```

3. Assign input parameter values for the statement.

```
ULValue deleteValue( old-value );  
prepStmt->SetParameter( 1, deleteValue );
```

4. Execute the statement.

```
long rowsDeleted = prepStmt->ExecuteStatement();
```

5. Commit the change.

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

Data retrieval: **SELECT**

The SELECT statement allows you to retrieve information from the database. When you execute a SELECT statement, the PreparedStatement.ExecuteQuery method returns a ResultSet object.

☞ For more information, see [“Class UltraLite_PreparedStatement_iface” on page 261](#).

❖ **To execute a SELECT statement**

1. Create a new prepared statement and result set.

```
PreparedStatement * prepStmt;
```

2. Assign a prepared statement to your newly created PreparedStatement object.

```
ULValue sqltext( "SELECT MyColumn FROM MyTable" );  
prepStmt = conn->PrepareStatement( sqltext );
```

3. Execute the statement.

In the following code, the result of the SELECT query contain a string, which is output to a command prompt.

```
#define MAX_NAME_LEN    100
ULValue mycol;
ResultSet * rs = stmt->ExecuteQuery();
rs->BeforeFirst();
while( rs->Next() ){
    char mycol[ MAX_NAME_LEN ];
    val = rs->Get( 1 );
    val.GetString( mycol, MAX_NAME_LEN );
    printf( "mycol= %s\n", mycol );
}
```

Navigating dynamic SQL result sets

You can navigate through a result set using methods associated with the `ResultSet` object.

The result set object provides you with the following methods to navigate a result set.

- ◆ **AfterLast()** moves to a position after the last row.
- ◆ **BeforeFirst()** moves to a position before the first row.
- ◆ **First()** moves to the first row.
- ◆ **Last()** moves to the last row.
- ◆ **Next()** moves to the next row.
- ◆ **Previous()** moves to the previous row.
- ◆ **Relative(offset)** moves a certain number of rows relative to the current row, as specified by the offset. Positive offset values move forward in the result set, relative to the current position of the cursor in the result set, and negative offset values move backward in the result set. An offset value of zero does not move the cursor, but allows you to repopulate the row buffer.

☞ For more information, see [“Class UltraLite_ResultSet_iface” on page 264](#).

Result set schema description

The `ResultSet.GetSchema` method allows you to retrieve information about a result set, such as column names, total number of columns, column scales, column sizes and column SQL types.

Example

The following example demonstrates how to use the `ResultSet.GetSchema` method to display schema information in a console window.

```
ResultSetSchema * rss = rs->GetSchema();
ULValue val;
char name[ MAX_NAME_LEN ];
for( int i = 1;
    i <= rss->GetColumnCount();
    i++ ){
    val = rss->GetColumnName( i );
    val.GetString( name, MAX_NAME_LEN );
    printf( "id= %d, name= %s \n", i, name );
}
```

➡ For more information, see [“GetSchema Function” on page 264](#).

Accessing data with the Table API

UltraLite applications can access table data using dynamic SQL or the Table API. This section describes data access using the Table API.

☞ For information about dynamic SQL, see [“Accessing data using dynamic SQL” on page 21](#).

This section explains how to perform the following tasks using the Table API.

- ◆ Scrolling through the rows of a table.
- ◆ Accessing the values of the current row.
- ◆ Using find and lookup methods to locate rows in a table.
- ◆ Inserting, deleting, and updating rows.

Navigating the rows of a table

UltraLite C++ component provides you with a number of methods to navigate a table in order to perform a wide range of navigation tasks.

The table object provides you with the following methods to navigate a table.

- ◆ **AfterLast()** moves to a position after the last row.
- ◆ **BeforeFirst()** moves to a position before the first row.
- ◆ **First()** moves to the first row.
- ◆ **Last()** moves to the last row.
- ◆ **Next()** moves to the next row.
- ◆ **Previous()** moves to the previous row.
- ◆ **Relative(offset)** moves a certain number of rows relative to the current row, as specified by the offset. Positive offset values move forward in the table, relative to the current position of the cursor in the table, and negative offset values move backward in the table. An offset value of zero does not move the cursor, but allows you to repopulate the row buffer.

☞ For more information, see [“Class UltraLite_Table_iface” on page 278](#).

Example

The following code opens the MyTable table and displays the value of the MyColumn column for each row.

```
Table * t = conn->openTable( "MyTable" );
ul_column_num colValue =
    t->GetSchema()->GetColumnID( "MyColumn" );
while ( t->Next() ){
    char lname[ MAX_NAME_LEN ];
    printf( "%s\n", colValue );
}
```

You expose the rows of the table to the application when you open the table object. By default, the rows are ordered by primary key value, but you can specify an index when opening a table to access the rows in a particular order.

Example

The following code fragment moves to the first row of the MyTable table as ordered by the ix_col index.

```
ULValue table_name( "MyTable" )
ULValue index_name( "ix_col" )
Table * t =
    conn->OpenTableWithIndex( table_name, index_name );
t.moveFirst();
```

☞ For more information, see [“Class UltraLite_Table_iface” on page 278](#).

Using UltraLite modes

UltraLite mode determines the purpose for which the values in the buffer will be used. UltraLite has the following four modes of operation, in addition to a default mode.

- ◆ **Insert mode** The data in the buffer is added to the table as a new row when the insert method is called.
- ◆ **Update mode** The data in the buffer replaces the current row when the update method is called.
- ◆ **Find mode** Used to locate a row whose value exactly matches the data in the buffer when one of the find methods is called.
- ◆ **Lookup mode** Used to locate a row whose value matches or is greater than the data in the buffer when one of the lookup methods is called.

Accessing the values of the current row

A Table object is always located at one of the following positions.

- ◆ Before the first row of the table.
- ◆ On a row of the table.

-
- ◆ After the last row of the table.

If the Table object is positioned on a row, you can use one of a set of methods appropriate for the data type to retrieve or modify the value of each column.

Retrieving column values The Table object provides a set of methods for retrieving column values. These methods take the column ID as argument.

Example The following code fragment retrieves the value of the lname column, which is a character string.

```
ULValue val;
char lname[ MAX_NAME_LEN ];
val = t->Get( lname_col_id );
val.GetString( lname, MAX_NAME_LEN );
```

The following code retrieves the value of the cust_id column, which is an integer.

```
int id = (int)(t->Get( id_col_id );
```

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

Example For example, the following code sets the value of the lname column to Kaminski.

```
ULValue lname_col( "fname" );
ULValue v_lname( "Kaminski" );
t->Set( lname_col, v_lname );
```

By assigning values to these properties you do not alter the value of the data in the database. You can assign values to the properties even if you are before the first row or after the last row of the table, but it is an error to try to access data when the current row is at one of these positions, for example by assigning the property to a variable.

```
// This code is incorrect
t.BeforeFirst();
id = t.Get( cust_id );
```

Casting values The method you choose must match the data type you wish to assign. UltraLite automatically casts database data types where they are compatible, so that you could use the getString method to fetch an integer value into a string variable, and so on.

Searching rows with find and lookup

UltraLite has several modes of operation for working with data. Two of these modes, the find and lookup modes, are used for searching. The Table

object has methods corresponding to these modes for locating particular rows in a table.

Note

The columns searched using Find and Lookup methods must be in the index used to open the table.

- ◆ **Find methods** move to the first row that exactly matches specified search values, under the sort order specified when the Table object was opened. If the search values cannot be found, the application is positioned before the first or after the last row.
- ◆ **Lookup methods** move to the first row that matches or is greater than a specified search value, under the sort order specified when the Table object was opened.

❖ **To search for a row**

1. Enter find or lookup mode.

The mode is entered by calling a method on the table object. For example, the following code enters find mode.

```
t.FindBegin();
```

2. Set the search values.

You do this by setting values in the current row. Setting these values affects the buffer holding the current row only, not the database. For example, the following code fragment sets the value in the buffer to Kaminski.

```
ULValue lname_col = t->GetSchema()->GetColumnID( "lname" );  
ULValue v_lname( "Kaminski" );  
t.Set( lname_col, v_lname );
```

3. Search for the row.

Use the appropriate method to carry out the search. For example, the following instruction looks for the first row that exactly matches the specified value in the current index.

For multi-column indexes, a value for the first column is always used, but you can omit the other columns.

```
tCustomer.FindFirst();
```

4. Search for the next instance of the row.

Use the appropriate method to carry out the search. For a find operation, FindNext() locates the next instance of the parameters in the index. For a lookup, MoveNext() locates the next instance.

☞ For more information, see “Class `UltraLite_Table_iface`” on page 278.

Updating rows

The following procedure updates a row.

❖ To update a row

1. Move to the row you wish to update.

You can move to a row by scrolling through the table or by searching the table using `find` and `lookup` methods.

2. Enter update mode.

For example, the following instruction enters update mode on `t`.

```
t.BeginUpdate();
```

3. Set the new values for the row to be updated. For example, the following instruction sets the `id` column in the buffer to 3.

```
t.SetInt( id, 3 );
```

4. Execute the Update.

```
t.Update();
```

After the update operation the current row is the row that has been updated. If you changed the value of a column in the index specified when the `Table` object was opened, the current row is undefined.

`UltraLite C++` component does not commit changes to the database until you commit them using `conn->Commit()`. For more information, see “[Managing transactions](#)” on page 32.

Caution

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

Inserting rows

The steps to insert a row are very similar to those for updating rows, except that there is no need to locate a row in the table before carrying out the insert operation. The order of row insertion into the table has no significance.

Example

The following code fragment inserts a new row.

```
t.InsertBegin();  
t.SetInt( id, 3 );  
t.SetString( lname, "Carlo" );  
t.Insert();  
t.Commit();
```

If you do not set a value for one of the columns, and that column has a default, the default value is used. If the column has no default, one of the following entries is used.

- ◆ For nullable columns, NULL.
- ◆ For numeric columns that disallow NULL, zero.
- ◆ For character columns that disallow NULL, an empty string.
- ◆ To explicitly set a value to NULL, use the setNull method.

For update operations, an insert is applied to the database in permanent storage when a commit is carried out. In AutoCommit mode, a commit is carried out as part of the insert method.

Deleting rows

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

The following procedure deletes a row.

❖ To delete a row

1. Move to the row you wish to delete.
2. Execute the Table.Delete() method.

```
t.Delete();
```

Managing transactions

☞ For background information about transaction management in UltraLite, see [“Transaction processing, recovery, and backup”](#) [*UltraLite Database User’s Guide*, page 48].

The UltraLite C++ component does not support an autocommit model. It requires that transactions be explicitly completed.

❖ To commit a transaction

1. Execute a `Connection.Commit()` statement.

❖ To roll back a transaction

1. Execute a `Connection.Rollback()` statement.

☞ For more information, see [“Class UltraLite_Connection_iface”](#) on [page 236](#).

Accessing schema information

The objects in the API represent tables, columns, indexes, and synchronization publications. Each object has a `GetSchema` method that provides access to information about the structure of that object.

You cannot modify the schema through the API. You can only retrieve information about the schema.

☞ For information about modifying the schema, see [“Upgrading your database schema” on page 15](#).

You can access the following schema objects and information.

- ◆ **DatabaseSchema** exposes the number and names of the tables in the database, as well as global properties such as the format of dates and times.

To obtain a `DatabaseSchema` object, use `Connection.GetSchema`.

☞ For more information, see [“GetSchema Function” on page 238](#).

- ◆ **TableSchema** The number and names of the columns and indexes for this table.

To obtain a `TableSchema` object, use `Table.GetSchema`.

☞ For more information, see [“GetSchema Function” on page 281](#).

- ◆ **IndexSchema** Information about the column in the index. As an index has no data directly associated with it there is no separate `Index` class, just a `IndexSchema` class.

To obtain a `IndexSchema` object, call the `TableSchema.GetIndex`, the `TableSchema.GetOptimalIndex`, or the `TableSchema.GetPrimaryKey` method.

☞ For more information, see [“Class UltraLite_Table_iface” on page 278](#).

- ◆ **PublicationSchema** A list of the tables and columns contained in a publication. Publications are also comprised of schema only, and so there is no `Publication` object.

To obtain a `PublicationSchema` object, call the `DatabaseSchema.TableSchema.GetPublicationSchema` method.

☞ For more information, see [“GetSchema Function” on page 281](#).

Handling errors

You should check for errors after each database operation. You do so by using methods of the `ULSqlca` object. For example, `LastCodeOK()` checks if the operation was successful, while `GetSQLCode()` returns the numerical value of the `SQLCode`. You can look up the meaning of these values in [“Error messages indexed by Adaptive Server Anywhere SQLCODE” \[ASA Error Messages, page 2\]](#).

In addition to explicit error handling, UltraLite supports an error callback function. If you register a callback function, then UltraLite calls the function each time an error occurs. The callback function does not control application flow, but does provide you with the ability to be notified of all errors, which is particularly helpful during application development. Use of the callback is illustrated in the tutorial: [“Tutorial: Build an Application Using the C++ Component” on page 147](#).

☞ For a sample callback function, see [“Callback function for ULRegisterErrorCallback” on page 204](#) and [“ULRegisterErrorCallback function” on page 213](#).

☞ For a list of error codes thrown by the UltraLite C++ component, see [“Error messages indexed by Adaptive Server Anywhere SQLCODE” \[ASA Error Messages, page 2\]](#).

☞ For more information, see [“Class ULSqlca” on page 229](#).

Authenticating users

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 connect as this initial user.

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

☞ For more information, see “[User authentication in UltraLite](#)” [*UltraLite Database User’s Guide*, page 40].

❖ To add a user or change a password for an existing user

1. Connect to the database as a user with DBA authority.
2. Grant the user connection authority with the desired password using the `Connection.GrantConnectTo` method.

This procedure is the same whether you are adding a new user or changing the password of an existing user.

☞ For more information, see “[Class UltraLite_Connection_iface](#)” on [page 236](#).

❖ To delete an existing user

1. Connect to the database as a user with DBA authority.
2. Revoke the user’s connection authority using the `Connection.RevokeConnectFrom` method.

Encrypting data

You can encrypt or obfuscate your UltraLite database using the UltraLite C++ component.

☞ For background information, see [“Encrypting UltraLite databases” \[UltraLite Database User’s Guide, page 36\]](#),

Encryption

To create a database with encryption, specify an encryption key by specifying the **key** connection parameter in the connection string. When you call the `CreateAndOpenDatabase` method, the database is created and encrypted with the specified key.

☞ For more information, see [“Encryption Key connection parameter ” \[UltraLite Database User’s Guide, page 75\]](#).

You can change the encryption key by specifying the new encryption key with the `UltraLite_Connection.ChangeEncryptionKey` method.

☞ For more information, see [“Class UltraLite_Connection_iface” on page 236](#).

After the database is encrypted, connections to the database must specify the correct encryption key. Otherwise, the connection fails.

Obfuscation

To obfuscate the database, specify `obfuscate=1` as a creation parameter.

☞ For more information about database encryption, see [“Encrypting UltraLite databases” \[UltraLite Database User’s Guide, page 36\]](#).

Synchronizing data

Users of SQL Anywhere Studio 9.0 can synchronize UltraLite applications with a central database. Synchronization requires the MobiLink synchronization software included with SQL Anywhere Studio.

This section provides a brief introduction to synchronization and describes some features of particular interest to users of the UltraLite C++ component.

☞ For a more detailed explanation of synchronization, see “UltraLite Clients” [*MobiLink Clients*, page 277] and “UltraLite Synchronization Parameters” [*MobiLink Clients*, page 315].

UltraLite C++ component supports TCP/IP, HTTP, and HTTPS synchronization. Synchronization is initiated by the UltraLite application. In all cases, you use methods and properties of the connection object to control synchronization.

Note

To synchronize using HTTPS or encryption over TCP/IP you must obtain the separately-licensable security option. To order this option, see the card in your SQL Anywhere Studio package or see <http://www.sybase.com/detail?id=1015780>.

☞ For more information, see “Welcome to SQL Anywhere Studio” [*Introducing SQL Anywhere Studio*, page 4].

Compiling and linking your application

A set of runtime libraries is available for some platforms when using the UltraLite C++ component. These include, for Windows CE and Windows XP, a database engine that permits multi-process access to the same database.

The runtime libraries are provided in the *UltraLite\palm*, *UltraLite\ce*, and *UltraLite\win32* subdirectories of your SQL Anywhere installation.

The runtime library for Palm OS is as follows:

- ◆ **ulrt.lib** A static library. This is located in *UltraLite\palm\68k\lib\cw*.

The Windows CE libraries are in the *UltraLite\ce\<platform>* directories, where *platform* may be one of *arm*, *386*, *mips*, or *emulator*. Both static and dynamic libraries are provided for Windows CE:

- ◆ **ulrt9.dll** A Unicode character set dynamic link library. To use this library, link your application against the import library, *UltraLite\ce\<platform>\lib\ulimp.lib*.

When linking against this library, be sure to specify the following compilation options:

```
/DUNICODE /DUL_USE_DLL
```

- ◆ **ulrt.lib** A Unicode character set static library. This is located in *UltraLite\ce\<platform>\lib*.

When linking against this library, be sure to specify the following compilation option:

```
/DUNICODE
```

- ◆ **ulrtc.lib** A Unicode character set static library for use with the UltraLite engine for multi-process access to an UltraLite database. This is located in *UltraLite\ce\<platform>\lib*.

When linking against this library, be sure to specify the following compilation option:

```
/DUNICODE
```

Runtime libraries for
Windows XP

The *UltraLite\win32\386* directory contains libraries for Windows operating systems other than Windows CE. These include the following:

- ◆ **ulrt9.dll** An ANSI character set dynamic link library. To use this library, link your application against the import library, *UltraLite\win32\386\ulimp.lib*. Databases created using this library

cannot be deployed to Windows CE devices, as Windows CE uses Unicode.

When linking against this library, be sure to specify the following compilation option:

```
/DUL_USE_DLL
```

- ◆ **ulrtw9.dll** A Unicode character set dynamic link library. To use this library, link your application against the import library, *UltraLite\win32\386\ulimpw.lib*.

When linking against this library, specify the following compilation options:

```
/DUNICODE /DUL_USE_DLL
```

- ◆ **ulrtcw9.dll** A Unicode character set dynamic link library for use with the UltraLite engine for multi-process access to an UltraLite database. To use this library, link your application against the import library, *UltraLite\win32\386\ulimpcw.lib*. Databases created using this library require distribution of the UltraLite engine, as well as the DLL.

When linking against this library, be sure to specify the following compilation options:

```
/DUNICODE /DUL_USE_DLL
```

For information about the UltraLite engine, see “Using the UltraLite engine” [*UltraLite Database User’s Guide*, page 61].

CHAPTER 3

Developing Applications Using the Static C++ API

About this chapter

This chapter describes how to develop applications using the UltraLite Static 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

Topic:	page
Introduction	42
Defining features for your application	43
Connecting to a database	45
Accessing data	46
Authenticating users	47
Encrypting data	49
Synchronizing data	51
Building Static C++ API applications	58

Introduction

This chapter provides notes for developers who are writing and building UltraLite applications using the Static 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 43](#).
- ◆ Information on generating C++ API classes from your reference database.
 - ☞ See [“Generating UltraLite C++ classes” on page 58](#).
- ◆ Notes on compiling and linking UltraLite C++ API applications.
 - ☞ See [“Compiling and linking your application” on page 59](#).

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 [“Using UltraLite Static Interfaces”](#) [*UltraLite Database User's Guide*, page 193].

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](#)” [*UltraLite Database User’s Guide*, page 202].

☞ 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](#)” [*UltraLite Database User’s Guide*, page 203].

☞ For information on using placeholders, and other aspects of writing SQL statements for UltraLite, see “[Writing UltraLite SQL statements](#)” [*UltraLite Database User’s Guide*, page 205].

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 Static 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 publication is purely a convenience for UltraLite, and does not imply any connection with SQL Remote or MobiLink synchronization.

Publications allow you to qualify which rows any 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 Static C++ API applications

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

- ◆ [“ULEnableFileDB function” on page 208](#)
- ◆ [“ULRegisterSchemaUpgradeObserver function” on page 216](#)
- ◆ [“ULEnablePalmRecordDB function” on page 210](#)
- ◆ [“ULEnableStrongEncryption function” on page 211](#)
- ◆ [“ULEnableUserAuthentication function” on page 212.](#)

Other aspects of database storage are configured using the `UL_STORE_PARMS` macro. For more information, see [“UL_STORE_PARMS macro” on page 222.](#)

Connecting to a database

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.

Accessing data

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 “[Selecting a network protocol](#)” [*MobiLink Clients*, page 288].

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.

Authenticating users

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 connect as this initial user.

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

☞ For more information, see “[User authentication in UltraLite](#)” [*UltraLite Database User’s Guide*, page 40].

❖ To enable user authentication (Static 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();
...
```

The following code fragment performs user management and authentication for a Static 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 311](#), and [“RevokeConnectFrom method” on page 315](#).

Encrypting data

You can encrypt or obfuscate your UltraLite database using the UltraLite Static C++ API.

☞ For background information, see “[Encrypting UltraLite databases](#)” [*UltraLite Database User’s Guide*, page 36],

Encryption

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

❖ To strongly encrypt an UltraLite database

1. Load the encryption module.

Call **ULEnableStrongEncryption** before opening the database.

You open a database by calling **ULData::Open**.

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, see “[UL_STORE_PARMS macro](#)” on [page 222](#).

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.

Changing the encryption key

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

❖ **To change the encryption key on an UltraLite database**

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

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

☞ For more information, see [“ULChangeEncryptionKey function” on page 364](#).

Obfuscation

❖ **To obfuscate an UltraLite database**

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

☞ For more information, see [“UL_ENABLE_OBFUSCATION macro” on page 221](#).

Synchronizing data

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

The synchronization logic that keeps UltraLite applications up to date with the consolidated database is not held in the application itself.

Synchronization scripts stored in the consolidated database, together with the MobiLink synchronization server and the UltraLite runtime library, control how changes are processed when they are uploaded and determines which changes are to be downloaded.

Overview

The specifics of each synchronization is controlled by a set of synchronization parameters. These parameters are gathered into a structure C/C++, 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 synchronization structure.

☞ For information, see [“Initializing the synchronization parameters” on page 89](#).

2. Assign the parameter values for your application.

☞ For information, see [“Network protocol options for UltraLite synchronization clients” \[MobiLink Clients, page 341\]](#).

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

☞ For information, see [“Invoking synchronization” on page 52](#).

You must ensure that there are no uncommitted changes when you synchronize. For more information, see [“Commit all changes before synchronizing” on page 53](#).

Synchronization parameters

Synchronization specifics are controlled through a set of synchronization parameters. For information on these parameters, see [“Network protocol options for UltraLite synchronization clients” \[MobiLink Clients, page 341\]](#).

Initializing the synchronization parameters

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

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 “[UltraLite Synchronization Parameters](#)” [*MobiLink Clients*, page 315].

❖ To initialize the synchronization parameters (Static C++ API)

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

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

Setting synchronization parameters

The following code initiates TCP/IP synchronization. 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 = ULStream();
synch_info.stream_parms =
    UL_TEXT( "host=localhost" );
conn.Synchronize( &synch_info );
```

Invoking synchronization

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

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

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

1. Call **Connection.InitSynchInfo()** to initialize the synchronization parameters, and call **Connection.Synchronize()** to synchronize. See “[Synchronize method](#)” on page 317.

❖ To invoke synchronization (HotSync)

1. Call **Connection.InitSynchInfo** to initialize the synchronization parameters and **ULSetSynchInfo** to manage synchronization before exiting the application.

☞ For more information, see [“ULSetSynchInfo function” on page 386](#).

HotSync synchronization is managed outside the application.

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” \[MobiLink Clients, page 320\]](#).

Adding initial data to your application

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

Development tip

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

For more synchronization development tips, see [“Development tips” \[MobiLink Administration Guide, page 47\]](#).

Monitoring and canceling synchronization

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

- ◆ An API for monitoring synchronization progress and for canceling synchronization.
- ◆ A progress indicator component that implements the interface, which you can add to your application.
- ◆ Specify the name of your callback function in the **observer** member of the synchronization structure (**ul_synch_info**).
- ◆ Call the synchronization function or method to start synchronization.
- ◆ UltraLite calls your callback function called whenever the synchronization state changes. The following section describes the synchronization state.

Handling synchronization status information

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

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

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

- ◆ **state** One of the following states:
 - **UL_SYNCH_STATE_STARTING** No synchronization actions have yet been taken.
 - **UL_SYNCH_STATE_CONNECTING** The synchronization stream has been built, but not yet opened.
 - **UL_SYNCH_STATE_SENDING_HEADER** The synchronization stream has been opened, and the header is about to be sent.
 - **UL_SYNCH_STATE_SENDING_TABLE** A table is being sent.

- **UL_SYNCH_STATE_SENDING_DATA** Schema information or data is being sent.
 - **UL_SYNCH_STATE_FINISHING_UPLOAD** The upload stage is completed and a commit is being carried out.
 - **UL_SYNCH_STATE_RECEIVING_UPLOAD_ACK** An acknowledgement that the upload is complete is being received.
 - **UL_SYNCH_STATE_RECEIVING_TABLE** A table is being received.
 - **UL_SYNCH_STATE_RECEIVING_DATA** Schema information or data is being received.
 - **UL_SYNCH_STATE_COMMITTING_DOWNLOAD** The download stage is completed and a commit is being carried out.
 - **UL_SYNCH_STATE_SENDING_DOWNLOAD_ACK** An acknowledgement that download is complete is being sent.
 - **UL_SYNCH_STATE_DISCONNECTING** The synchronization stream is about to be closed.
 - **UL_SYNCH_STATE_DONE** Synchronization has completed successfully.
 - **UL_SYNCH_STATE_ERROR** Synchronization has completed, but with an error.
 - ☞ For a description of the synchronization process, see “[The synchronization process](#)” [*MobiLink Administration Guide*, page 15].
-
- ◆ **tableCount** Returns the number of tables being synchronized. For each table there is a sending and receiving phase, so this number may be more than the number of tables being synchronized.
 - ◆ **tableIndex** The current table which is being uploaded or downloaded, starting at 0. This number may skip values when not all tables are being synchronized.
 - ◆ **info** A pointer to the **ul_synch_info** structure.
 - ◆ **sent.inserts** The number of inserted rows that have been uploaded so far.
 - ◆ **sent.updates** The number of updated rows that have been uploaded so far.
 - ◆ **sent.deletes** The number of deleted rows that have been uploaded so far.
 - ◆ **sent.bytes** The number of bytes that have been uploaded so far.

-
- ◆ **received.inserts** The number of inserted rows that have been downloaded so far.
 - ◆ **received.updates** The number of updated rows that have been downloaded so far.
 - ◆ **received.deletes** The number of deleted rows that have been downloaded so far.
 - ◆ **received.bytes** The number of bytes that have been downloaded so far.
 - ◆ **stops** Set this member to true to interrupt the synchronization. The SQL exception `SQLE_INTERRUPTED` is set, and the synchronization stops as if a communications error had occurred. The observer is *always* called with either the `DONE` or `ERROR` state so that it can do proper cleanup.
 - ◆ **getUserData** Returns the user data object.
 - ◆ **getStatement** Returns the statement that called the synchronization. The statement is an internal UltraLite statement, and this method is unlikely to be of practical use, but is included for completion.
 - ◆ **getErrorCode** When the synchronization state is set to `ERROR`, this method returns a diagnostic error code.
 - ◆ **isOKToContinue** This is set to **false** when **cancelSynchronization** is called. Otherwise, it is **true**.

Example

The following code illustrates a very simple observer function:

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

This observer produces the following output when synchronizing two tables:

```
UL_SYNC_STATE is 0: Starting
UL_SYNC_STATE is 1: Connecting
UL_SYNC_STATE is 2: Sending Header
UL_SYNC_STATE is 3: Sending Table 1 of 2
UL_SYNC_STATE is 3: Sending Table 2 of 2
UL_SYNC_STATE is 4: Receiving Upload Ack
UL_SYNC_STATE is 5: Receiving Table 1 of 2
UL_SYNC_STATE is 5: Receiving Table 2 of 2
UL_SYNC_STATE is 6: Sending Download Ack
UL_SYNC_STATE is 7: Disconnecting
UL_SYNC_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.

Building Static C++ API applications

This section covers the following subjects:

- ◆ “Generating UltraLite C++ classes” on page 58.
- ◆ “Compiling and linking your application” on page 59.

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

1. 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

1. 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**

1. 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” [*UltraLite Database User’s Guide*, page 89].

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 include 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 4

Developing Applications Using Embedded SQL

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

Before you begin  This chapter assumes an elementary familiarity with the UltraLite development process. For an overview, see [“Using UltraLite Static Interfaces”](#) [*UltraLite Database User’s Guide*, page 193].

 For reference information, see [“Embedded SQL API Reference”](#) on page 357.

 For detailed information about the SQL preprocessor, see [“The SQL preprocessor”](#) [*ASA Programming Guide*, page 203].

Contents	Topic:	page
	Introduction	62
	Initializing the SQL Communications Area	64
	Connecting to a database	66
	Using host variables	68
	Fetching data	80
	Authenticating users	85
	Encrypting data	87
	Adding synchronization to your application	89
	Building embedded SQL applications	97

Introduction

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

```
#include <stdio.h>
EXEC SQL INCLUDE SQLCA;
main( )
{
    db_init( &sqlca );
    EXEC SQL WHENEVER SQLERROR GOTO error;
    EXEC SQL CONNECT "DBA" IDENTIFIED BY "SQL";
    EXEC SQL UPDATE employee
        SET emp_lname = 'Plankton'
        WHERE emp_id = 195;
    EXEC SQL COMMIT;
    EXEC SQL DISCONNECT;
    db_fini( &sqlca );
    return( 0 );
error:
    printf( "update unsuccessful: sqlcode = %ld\n",
        sqlca.sqlcode );
    return( -1 );
}
```

Although too simple to be useful, this example demonstrates the following aspects common to all embedded SQL applications:

- ◆ Each SQL statement is prefixed with the keywords EXEC SQL.
- ◆ Each SQL statement ends with a semicolon.
- ◆ Some embedded SQL statements are not found in standard SQL. The INCLUDE SQLCA statement is one example.
- ◆ Embedded SQL provides library functions to perform some specific tasks. The functions **db_init** and **db_fini** are examples.

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

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

```
EXEC SQL INCLUDE SQLCA;
```

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

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

2. Your first executable database action must be a call to an embedded SQL **library function** named **db_init**. This function initializes the UltraLite runtime library. Only embedded SQL definition statements can be executed before this call.

☞ For more information, see [“db_init function” on page 360](#).

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 [“Authenticating users” on page 85](#), and [“User authentication in UltraLite” \[UltraLite Database User’s Guide, page 40\]](#).

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.

Initializing the SQL Communications 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.

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

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

After declaring the SQLCA (`EXEC SQL INCLUDE SQLCA;`) but before your application can carry out any operations on a database, it must initialize the communications area by calling `db_init` and passing it the SQLCA:

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

SQLCA provides error codes

You reference the SQLCA to test for a particular error code. The **sqlcode** field contains an error code when a database request causes an error (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 [ASA Error Messages](#).

- ◆ **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.

Connecting to a database

To connect to an UltraLite database from an embedded SQL application, include the EXEC SQL CONNECT statement in your code after initializing the SQLCA.

The CONNECT statement takes one of the following two forms:

```
EXEC SQL CONNECT "user-name" IDENTIFIED BY "password";
```

```
EXEC SQL CONNECT
```

```
USING "uid=user-name;pwd=password;dbf=database-filename";
```

The first syntax assumes that the startup parameters are supplied in UL_STORE_PARMS. For more information, see [“UL_STORE_PARMS macro” on page 222](#).

The second syntax ignores settings in UL_STORE_PARMS and supplies the database startup parameters explicitly (specifically the filename).

For more information on the CONNECT statement, see [“CONNECT statement \[ESQL\] \[Interactive SQL\]” \[ASA SQL Reference, page 332\]](#).

Managing multiple connections

If you want more than one database connection in your application, you can either use multiple SQLCAs or you can use a single SQLCA to manage the connections.

❖ 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 360](#).

2. The embedded SQL statement SET SQLCA is used to tell the SQL preprocessor to use a different SQLCA for database requests. Usually, a statement such as the following:

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

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

☞ For information about creating SQLCAs, see [“SET SQLCA statement \[ESQL\]” \[ASA SQL Reference, page 619\]](#).

Using a single SQLCA

As an alternative to using multiple SQLCAs, you can use a single SQLCA to manage more than one connection to a database.

Each SQLCA has a single active or current connection, but that connection can be changed. Before executing a command, use the `SET CONNECTION` statement to specify the connection on which the command should be executed.

☞ For more information, see “[SET CONNECTION statement \[Interactive SQL\] \[ESQL\]](#)” [*ASA SQL Reference*, page 610].

Using host variables

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

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[l];
} TYPE_DECIMAL;
```

- ◆ NULL-terminated blank-padded character string.

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

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

hold $n - 1$ characters.

Pointers to char, WCHAR, TCHAR

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

- ◆ NULL terminated UNICODE or wide character string.

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

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

- ◆ NULL terminated system-dependent character string.

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

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

- ◆ Fixed-length blank padded character string.

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

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

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

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

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

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

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

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

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

The **SQLDATETIME** structure can be used to retrieve fields of **DATE**, **TIME**, and **TIMESTAMP** type (or anything that can be converted to one of these). Often, applications have their own formats and date manipulation code. Fetching data in this structure makes it easier for a programmer to manipulate this data. Note that **DATE**, **TIME** and **TIMESTAMP** fields can also be fetched and updated with any character type.

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

☞ For more information, see the **DATE_FORMAT**, **TIME_FORMAT**, **TIMESTAMP_FORMAT**, and **DATE_ORDER** database options in “Database Options” [*ASA Database Administration Guide*, page 613]. While these options cannot be set during execution of an UltraLite program, their values are identical to the settings in the reference database used to generate the program.

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

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

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

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

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

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

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

The DECL_LONGBINARY struct may be used with more than 32K of data. Large data may be fetched all at once, or in pieces using the GET DATA statement. Large data may be supplied to the server all at once, or in pieces by appending to a database variable using the SET statement.

The structures are defined in the *sqlca.h* file. The VARCHAR, BINARY, and TYPE_DECIMAL types contain a one-character array and are thus not useful for declaring host variables, but they are useful for allocating variables dynamically or typecasting other variables.

DATE and TIME database types

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

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

Host variable usage

Host variables can be used in the following circumstances:

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

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

The scope of host variables

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

The preprocessor assumes all host variables are global

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

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

Examples

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

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

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

```

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

```

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

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

Using expressions as host variables

Because host variables must be simple names, the SQL preprocessor does not recognize pointer or reference expressions. For example, the following statement *does not work* because the SQL preprocessor does not understand the dot operator. The same syntax has a different meaning in SQL.

```

// Incorrect statement:
EXEC SQL SELECT LAST sales_id INTO :mystruct.mymember;

```

Although the above syntax is not allowed, you can still use an expression with the following technique:

- ◆ Wrap the SQL declaration section in an `#if 0` preprocessor directive. The SQL preprocessor will read the declarations and use them for the rest of the module because it ignores preprocessor directives.

- ◆ Define a macro with the same name as the host variable. Since the SQL declaration section is not seen by the C compiler because of the `#if` directive, no conflict will arise. Ensure that the macro evaluates to the same type host variable.

The following code demonstrates this technique to hide the *host_value* expression from the SQL preprocessor.

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

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

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

```

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

```

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

◆ pointer indirections

```

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

```

◆ array references

```

my_array[ i ]

```

◆ arbitrarily complex lvalues

Using host variables in C++

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

```

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

```


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

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

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

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

Using indicator variables

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

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

Example

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

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

Indicator variable values

The following table provides a summary of indicator variable usage.

Indicator Value	Supplying Value to database	Receiving value from database
0	Host variable value	Fetches a non-NULL value.
-1	NULL value	Fetches 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 `SQL_NO_INDICATOR` error is generated.

☞ Errors and warnings are returned in the `SQLCA` structure, as described in [“Initializing the SQL Communications Area” on page 64](#).

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 `SQL_E_TOO_MANY_RECORDS` error.
- ◆ If the query selects no rows, the `SQL_E_NOTFOUND` warning is returned.

☞ Errors and warnings are returned in the `SQLCA` structure, as described in [“Initializing the SQL Communications Area” on page 64](#).

Example

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

```

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

```

Fetching multiple rows

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

☞ For an introduction to cursors, see “[Working with cursors](#)” [ASA *Programming Guide*, page 21].

❖ To manage a cursor in embedded SQL

1. Declare a cursor for a particular SELECT statement, using the DECLARE statement.
2. Open the cursor using the OPEN statement.
3. Retrieve rows from the cursor one at a time using the FETCH statement.
 - ◆ Fetch rows until the SQLE_NOTFOUND warning is returned.
 - ☞ Error and warning codes are returned in the variable SQLCODE, defined in the SQL communications area structure.
4. Close the cursor, using the CLOSE statement.

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

The following is a simple example of cursor usage:

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

☞ For details of the FETCH statement, see “[FETCH statement \[ESQL \[SP\]\]](#)” [*ASA SQL Reference*, page 482].

Cursor positioning

A cursor is positioned in one of three places:

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

Absolute row from start		Absolute row from end
0	Before first row	$-n - 1$
1		$-n$
2		$-n + 1$
3		$-n + 2$
$n - 2$		-3
$n - 1$		-2
n		-1
$n + 1$	After last row	0

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 **SQLC_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 **SQLC_NOTFOUND** error, which can be used as a convenient signal to complete sequential processing of the rows.

You can also reposition the cursor to an absolute position relative to the start or the end of the query results, or move it relative to the current cursor

position. There are special *positioned* versions of the UPDATE and DELETE statements that can be used to update or delete the row at the current position of the cursor. If the cursor is positioned before the first row or after the last row, an SQLE_NOTFOUND error is returned.

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

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

Cursor positioning after updates

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

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

☞ For more information about temporary tables, see “[Use of work tables in query processing](#)” [*ASA SQL User's Guide*, page 190].

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.

Authenticating users

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 connect as this initial user.

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

☞ For more information, see “[User authentication in UltraLite](#)” [*UltraLite Database User’s Guide*, page 40].

For the static interfaces, you must enable the user authentication feature before you can add or delete users or change passwords. On Palm OS, 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.

❖ To enable user authentication (embedded SQL)

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

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

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

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

User authentication
example

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*.

```

//Embedded SQL
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 373, and “[ULRevokeConnectFrom function](#)” on page 382.

Encrypting data

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

☞ For background information, see “[Encrypting UltraLite databases](#)” [*UltraLite Database User’s Guide*, page 36],

Encryption

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

❖ To strongly encrypt an UltraLite database

1. Load the encryption module.

Call **ULEnableStrongEncryption** before opening the database.

You open a database by calling `db_init`.

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, see .

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.

Changing the encryption key

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

❖ **To change the encryption key on an UltraLite database**

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

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

☞ For more information, see [“ULChangeEncryptionKey function” on page 364](#).

Obfuscation

❖ **To obfuscate an UltraLite database**

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

☞ For more information, see [“UL_ENABLE_OBFUSCATION macro” on page 221](#).

Adding synchronization to your application

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

The synchronization logic that keeps UltraLite applications up to date with the consolidated database is not held in the application itself.

Synchronization scripts stored in the consolidated database, together with the MobiLink synchronization server and the UltraLite runtime library, control how changes are processed when they are uploaded and determines which changes are to be downloaded.

Overview

The specifics of each synchronization is controlled by a set of synchronization parameters. These parameters are gathered into a structure, 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 that holds the synchronization parameters.

☞ For information, see [“Initializing the synchronization parameters” on page 89](#).

2. Assign the parameter values for your application.

☞ For information, see [“Network protocol options for UltraLite synchronization clients” \[MobiLink Clients, page 341\]](#).

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

☞ For information, see [“Invoking synchronization” on page 90](#).

You must ensure that there are no uncommitted changes when you synchronize. For more information, see [“Commit all changes before synchronizing” on page 53](#).

Synchronization parameters

Synchronization specifics are controlled through a set of synchronization parameters. For information on these parameters, see [“Network protocol options for UltraLite synchronization clients” \[MobiLink Clients, page 341\]](#).

Initializing the synchronization parameters

The synchronization parameters are stored in a structure.

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](#)” [*MobiLink Clients*, page 316].

❖ To initialize the synchronization parameters (embedded SQL)

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

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

Setting synchronization parameters

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

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

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

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

Invoking synchronization

The details of how to invoke synchronization depends on your target

platform and on the synchronization stream.

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

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

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

❖ **To invoke synchronization (HotSync)**

1. Call **ULInitSynchInfo** to initialize the synchronization parameters, and call **ULSetSynchInfo** to manage synchronization before exiting the application.

☞ For more information, see “[ULSetSynchInfo function](#)” on page 386.

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](#)” [*MobiLink Clients*, page 320].

Adding initial data to your application

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

Development tip

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

For more synchronization development tips, see [“Development tips”](#) [*MobiLink Administration Guide*, page 47].

Handling synchronization communications errors

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

```
if( psqlca->sqlcode == SQLE_COMMUNICATIONS_ERROR ) {
    printf( " Stream error information:\n"
           "      stream_id          = %d\t(ss_stream_id)\n"
           "      stream_context      = %d\t(ss_stream_context)\n"
           "      stream_error_code    = %ld\t(ss_error_code)\n"
           "      error_string         = \"%s\"\n"
           "      system_error_code    = %ld\n",
           (int)info.stream_error.stream_id,
           (int)info.stream_error.stream_context,
           (long)info.stream_error.stream_error_code,
           info.stream_error.error_string,
           (long)info.stream_error.system_error_code );
}
```

SQLE_COMMUNICATIONS_ERROR is the general error code for communications errors. More information about the specific error is supplied to your application in the members of the stream_error synchronization parameter.

To keep UltraLite small, the runtime reports numbers rather than messages. For information on what the numbers mean, see [“stream_error synchronization parameter”](#) on page 440.

Monitoring and canceling synchronization

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

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

- ◆ Specify the name of your callback function in the **observer** member of the synchronization structure (**ul_synch_info**).
- ◆ Call the synchronization function or method to start synchronization.
- ◆ UltraLite calls your callback function called whenever the synchronization state changes. The following section describes the synchronization state.

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

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

Handling synchronization status information

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

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

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

- ◆ **state** One of the following states:
 - **UL_SYNCH_STATE_STARTING** No synchronization actions have yet been taken.
 - **UL_SYNCH_STATE_CONNECTING** The synchronization stream has been built, but not yet opened.

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

☞ For a description of the synchronization process, see “[The synchronization process](#)” [*MobiLink Administration Guide*, page 15].

- ◆ **tableCount** Returns the number of tables being synchronized. For each table there is a sending and receiving phase, so this number may be more than the number of tables being synchronized.
- ◆ **tableIndex** The current table which is being uploaded or downloaded, starting at 0. This number may skip values when not all tables are being synchronized.
- ◆ **info** A pointer to the **ul_synch_info** structure.
- ◆ **sent.inserts** The number of inserted rows that have been uploaded so far.
- ◆ **sent.updates** The number of updated rows that have been uploaded so far.
- ◆ **sent.deletes** The number of deleted rows that have been uploaded so far.

- ◆ **sent.bytes** The number of bytes that have been uploaded so far.
- ◆ **received.inserts** The number of inserted rows that have been downloaded so far.
- ◆ **received.updates** The number of updated rows that have been downloaded so far.
- ◆ **received.deletes** The number of deleted rows that have been downloaded so far.
- ◆ **received.bytes** The number of bytes that have been downloaded so far.
- ◆ **stop** Set this member to true to interrupt the synchronization. The SQL exception `SQLE_INTERRUPTED` is set, and the synchronization stops as if a communications error had occurred. The observer is *always* called with either the `DONE` or `ERROR` state so that it can do proper cleanup.
- ◆ **getUserData** Returns the user data object.
- ◆ **getStatement** Returns the statement that called the synchronization. The statement is an internal UltraLite statement, and this method is unlikely to be of practical use, but is included for completion.
- ◆ **getErrorCode** When the synchronization state is set to `ERROR`, this method returns a diagnostic error code.
- ◆ **isOKToContinue** This is set to **false** when **cancelSynchronization** is called. Otherwise, it is **true**.

Example

The following code illustrates a very simple observer function:

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

This observer produces the following output when synchronizing two tables:

```
UL_SYNC_STATE is 0: Starting
UL_SYNC_STATE is 1: Connecting
UL_SYNC_STATE is 2: Sending Header
UL_SYNC_STATE is 3: Sending Table 1 of 2
UL_SYNC_STATE is 3: Sending Table 2 of 2
UL_SYNC_STATE is 4: Receiving Upload Ack
UL_SYNC_STATE is 5: Receiving Table 1 of 2
UL_SYNC_STATE is 5: Receiving Table 2 of 2
UL_SYNC_STATE is 6: Sending Download Ack
UL_SYNC_STATE is 7: Disconnecting
UL_SYNC_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.

Building embedded SQL applications

This section describes a general build procedure for UltraLite embedded SQL applications. You can use a simpler modification if your application uses only a single .sqc file. For more information, see “[Single-file build procedure](#)” on page 100.

☞ This section assumes a familiarity with the overall embedded SQL development model. For more information, see “[Using UltraLite Static Interfaces](#)” [*UltraLite Database User's Guide*, page 193].

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

General build procedure

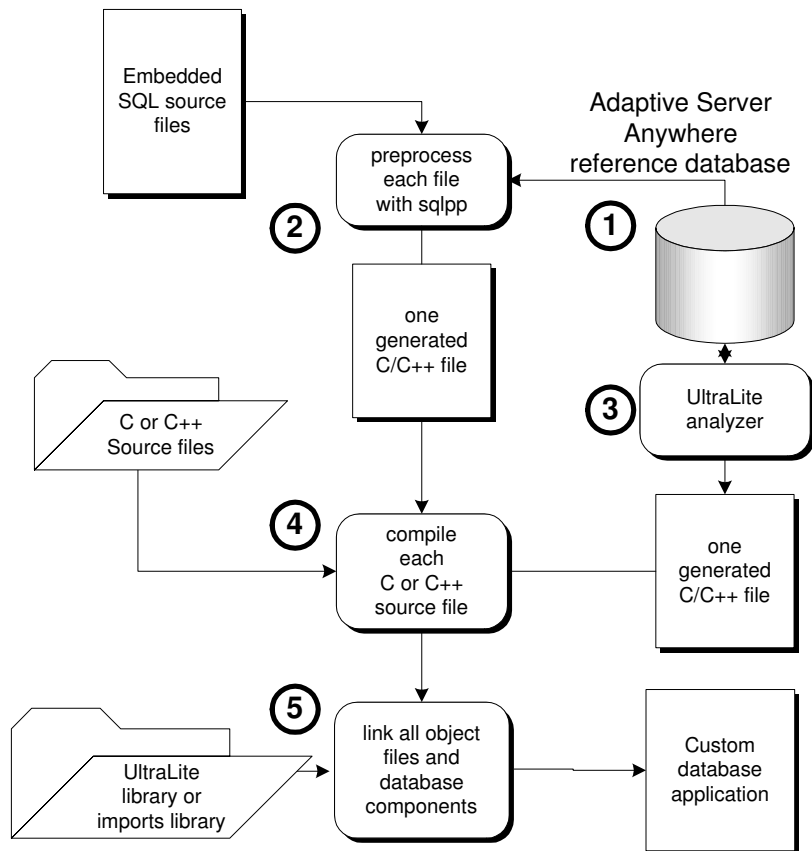
Sample code

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

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

Procedure

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



❖ **To build an UltraLite embedded SQL application**

1. Start the Adaptive Server Anywhere personal database server, specifying your reference database.
2. Run the SQL preprocessor on *each* embedded SQL source file.
 The SQL preprocessor is the `sqlpp` command-line utility. It carries out two functions in an UltraLite development project:
 - ◆ It preprocesses the embedded SQL files, producing C files to be compiled into your application.
 - ◆ It adds the SQL statements to the reference database, for use by the UltraLite generator.

Caution

sqlpp overwrites the output file without regard to its contents. Ensure that the output file name does not match the name of any of your source files. By default, sqlpp constructs the output file name by changing the suffix of your source file to .c. When in doubt, specify the output file name explicitly, following the name of the source file.

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

☞ For detailed information about the SQL preprocessor, see “The SQL Preprocessor” [*UltraLite Database User’s Guide*, page 95].

☞ For information about projects, see “Creating an UltraLite project” [*UltraLite Database User’s Guide*, page 202].

3. Run the UltraLite generator.

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

Enter the following command at a command-prompt:

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

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

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

- ◆ **-c** You must supply a connection string, to connect to the reference database.
 - ☞ For information on Adaptive Server Anywhere connection strings, see “Connection parameters” [*ASA Database Administration Guide*, page 176].
- ◆ **-f** Specify the output file name.
- ◆ **-j** Specify the UltraLite project name.
 - ☞ For more information on UltraLite generator options, see “The UltraLite Generator” [*UltraLite Database User’s Guide*, page 89].

4. Compile each C or C++ source file for the target platform of your choice. Include

- ◆ each C files generated by the SQL preprocessor,
- ◆ the C file made by the UltraLite generator,
- ◆ any additional C or C++ source files that comprise your application.

5. Link all these object files, together with the UltraLite runtime library.

Example

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

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

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

Single-file build procedure

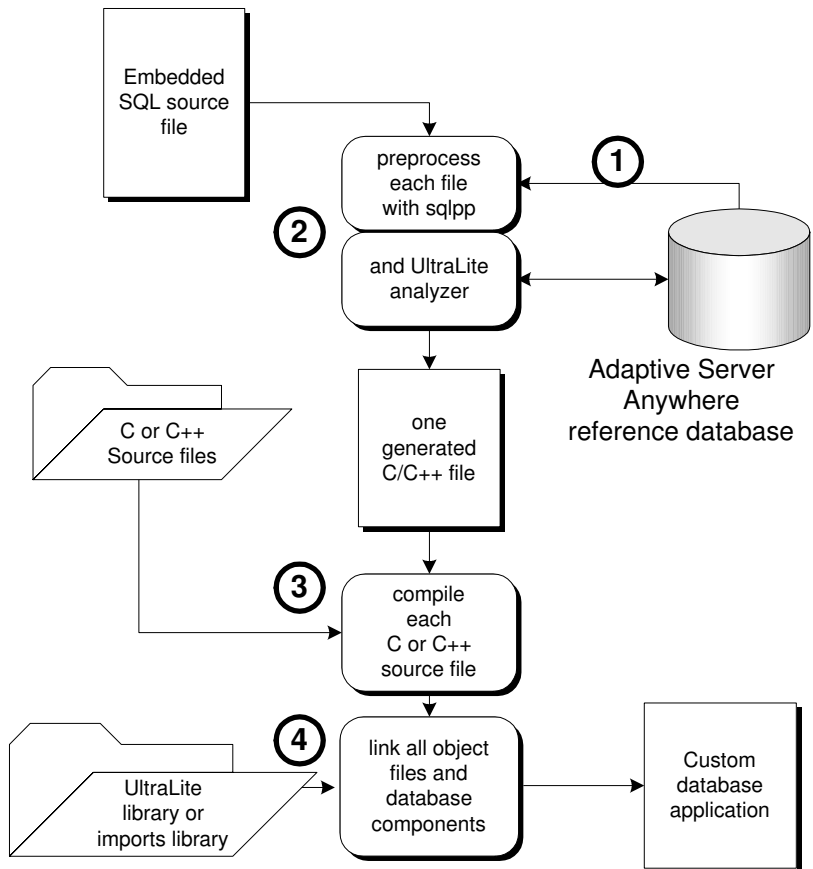
☞ This section assumes a familiarity with the overall embedded SQL development model. For more information, see [“Using UltraLite Static Interfaces”](#) [*UltraLite Database User’s Guide*, page 193].

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

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

If these requirements are not all met, you must use the general build process. For instructions, see [“Building embedded SQL applications”](#) on page 97 .

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



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

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

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

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

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

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

Example

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

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

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

Configuring development tools for embedded SQL development

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

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

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

☞ The UltraLite plug-in for Metrowerks CodeWarrior automatically provides Palm Computing platform developers with the techniques described here. For information on this plug-in, see [“Developing UltraLite applications with Metrowerks CodeWarrior” on page 115.](#)

☞ or a tutorial describing details for a very simple project, see [“Tutorial: Build an Application Using Embedded SQL” on page 177.](#)

SQL preprocessing

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

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

1. Add the `.sqlc` 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 `.sqlc` file.

- ◆ The custom build rule should run the SQL preprocessor. In Visual C++, the build rule should have the following command (entered on a single line):

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

where `asany9` is an environment variable that points to your SQL Anywhere installation directory, `connection-string` provides the connection to the reference database, and `project-name` is the name of your UltraLite project.

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

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

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

3. Compile the `.sqlc` files, and add the generated `.c` files to your development project.

You need to add the generated files to your project even though they are not source files, so that you can set up dependencies and build options.

4. For each generated `.c` file, set the preprocessor definitions.

- ◆ Under General or Preprocessor, add `UL_USE_DLL` to the Preprocessor definitions.
- ◆ Under Preprocessor, add `$(asany9)\h` and any other include folders you require to your include path, as a comma-separated list.

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

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

1. Add a dummy file to your development project.

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

-
2. Set the build rules for this file to be the UltraLite generator command line. In Visual C++, use a command of the following form (which should be all on one line):

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

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

3. Set the dummy file to depend on the output files from the preprocessor. In Visual C++, click Dependencies on the custom build page, and enter the names of the generated *.c* files produced by the SQL preprocessor. This instructs Visual C++ to run the UltraLite generator after all the necessary embedded SQL files have been preprocessed.

4. Compile your dummy file to generate the *.c* file that implements the UltraLite database.

5. Add the generated UltraLite database file to your project and change its C/C++ settings.

6. Add the UltraLite imports library to your object/libraries modules list.

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

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

For other targets, choose the appropriate import library.

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

CHAPTER 5

Common Features of UltraLite C/C++ Interfaces

About this chapter

This chapter addresses development issues that are common to the UltraLite C/C++ interfaces. It also describes how to mix the interfaces in the same application.

Contents

Topic:	page
Understanding the SQL Communications Area	106
Combining UltraLite C/C++ interfaces	108
Defragmenting UltraLite databases	110

Understanding the SQL Communications Area

All UltraLite C/C++ interfaces address the same UltraLite runtime engine. They each provide access to the same underlying functionality.

All UltraLite C/C++ interfaces also share the same basic data structure for marshaling data between the UltraLite runtime and your application. This data structure is the SQL Communications Area or SQLCA. Each SQLCA has a current connection, and separate threads cannot share a common SQLCA.

Your code must carry out the following tasks before connecting to a database:

- ◆ Initialize a SQLCA. This prepares your application for communication with the UltraLite runtime.
- ◆ Set any data store characteristics and register your error callback function. For a list of functions, see [“UltraLite C/C++ Common API Reference” on page 203](#).
- ◆ Start a database. This operation may be carried out as part of opening the connection.

The following functions are equivalent ways of carrying out these tasks.

Task	Interface	Function
Initialize SQLCA	Embedded SQL	db_init
	Static C++ API	ULData::Initialize
	C++ Component	ULSqlca::Initialize
Initialize SQLCA and start database	Embedded SQL	db_init db_start_database
	Static C++ API	ULData::Open
	C++ Component	The database is started as part of the connection function in UltraLite_ - DatabaseManager

Creating databases

To create a database, UltraLite requires a schema definition. In static interfaces that use generated code, the schema definition is held in the

generated code. In the UltraLite C++ component, the schema definition is held externally in a schema file.

The following connection primitives know about the generated code database schema, and create a database that matches that schema.

- ◆ `db_start_database` (embedded SQL)
- ◆ `ULData::Open` (static C++ API)
- ◆ `EXEC SQL CONNECT` statement (embedded SQL)

The C++ component does not require any generated code and does not reference the generated code database schema. If you add C++ component code to a static application, you must use one of the primitives that know about the generated code database schema to start the database. You cannot start a database without an external schema file using the C++ component `DatabaseManager::CreateAndOpenDatabase()` method.

Name the database explicitly

The different interfaces use different default filenames for the database. If you are mixing the interfaces, it is therefore best to always name the database explicitly when creating or connecting. You can do this using a `DBN` connection parameter.

Combining UltraLite C/C++ interfaces

You can combine the use of separate C/C++ interfaces in a single application. There are several reasons you may wish to do this. For example:

- ◆ You may wish to add features from the new UltraLite C++ Component to an existing static interface application.
- ◆ You may wish to include dynamic SQL features together with SQL statements that are supported only in static interfaces.

Use a generated database schema

When combining interfaces, the database schema must be defined by the generated code of a static interface because the static code has no way of accessing a database schema held in an external file. This means that if you are mixing C++ Component features with static interface features, you must start the database using the static interface. That is, you must start the database using `ULData::StartDatabase` (static C++ API) or `db_start_database` (embedded SQL).

Simple mixing

The simplest way to add dynamic SQL or other C++ Component features to a static interface application is to use a separate SQLCA and a separate connection for the two feature sets. One SQLCA and connection can use C++ Component code and execute dynamic SQL queries, while the other SQLCA and connection can use static interface features. In this approach, no coordination between the static interface code and the C++ Component is required. The overhead for the separate connection is minimal.

The only restriction to this method of mixing interfaces is that the dynamic SQL and static statements cannot be part of the same transaction.

Sharing a SQLCA

If you wish statements from static and dynamic SQL to be part of the same transaction, you must share a single connection among the interfaces. Sharing a SQLCA is a prerequisite to sharing connections. It is also possible to write your application to share a SQLCA but have different connections.

❖ To share a single SQLCA

1. Declare the SQLCA using the UltraLite C++ Component `ULSqlca` object.

```
//Declare a SQLCA using C++ Component
ULSqlca    MySqlca;
```

2. Set the static interface to use this object.

For embedded SQL:

```
//Set the embedded SQL SQLCA
EXEC SQL SET SQLCA "MySqlca.GetCA()";
```

For the static C++ API:


```
ULData MyData
MyData.Initialize( MySqlca.GetCA() );
```

3. Initialize the SQLCA once only.

Use either the UltraLite C++ Component `ULSqlca::Initialize()` method or `db_init`. The following example uses C++ component initialization.

```
//Initialize the SQLCA
MySqlca.Initialize();
```

4. Start the database.

You must start the database from the static component, so that the internal generated schema can be used to create the database on the first connection.

For embedded SQL:

```
db_start_database( );
```

For the static C++ API:

```
MyData.StartDatabase( );
```

You can now use separate connections from the C++ Component and the static interface, and these connections will share the same SQLCA. If you wish to share a single connection, there is another step to carry out.

❖ **To share a single connection**

1. Ensure that your application is sharing a single SQLCA, as described in the previous procedure.

2. Manage the connection using the UltraLite C++ Component.

Use the `DatabaseManager::OpenConnection` and `Connection::Release` methods to open and close connections. Do not use the static interface connection mechanisms.

Once `OpenConnection` returns, the connection is current for the SQLCA and is used for any subsequent embedded SQL or static C++ API statements.

Defragmenting UltraLite databases

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

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

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

☞ For more information, see [“ULStoreDefragFini function” on page 218](#), [“ULStoreDefragInit function” on page 219](#), and [“ULStoreDefragStep function” on page 220](#).

❖ To defragment an UltraLite database

1. Obtain a `p_ul_store_defrag_info` information block. For example,

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

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

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

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

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

Example

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

```
p_ul_store_defrag_info    DefragInfo;

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

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

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

CHAPTER 6

Developing UltraLite Applications for the Palm Computing Platform

About this chapter

This chapter describes details of development, deployment and synchronization that are specific to developing applications for the Palm Computing Platform.

Contents

Topic:	page
Introduction	114
Developing UltraLite applications with Metrowerks CodeWarrior	115
Saving state in UltraLite Palm applications	120
Building multi-segment applications	122
Adding HotSync synchronization to Palm applications	125
Adding TCP/IP, HTTP, or HTTPS synchronization to Palm applications	127
Deploying Palm applications	128

Introduction

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

Development environments

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

- ◆ Metrowerks CodeWarrior, version 8 or 9, using embedded SQL, the static C++ API, or the UltraLite C++ component.

☞ See “[Developing UltraLite applications with Metrowerks CodeWarrior](#)” on page 115.

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.

☞ For a list of supported platforms, see “[UltraLite development platforms](#)” [*Introducing SQL Anywhere Studio*, page 99] and “[UltraLite target platforms](#)” [*Introducing SQL Anywhere Studio*, page 109].

- ◆ AppForge MobileVB, using the UltraLite MobileVB component.

☞ See *UltraLite for MobileVB User’s Guide*.

Target platforms

☞ For a list of supported target operating systems, see “[UltraLite target platforms](#)” [*Introducing SQL Anywhere Studio*, page 109].

See also

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

For information on supported development environments, see “[UltraLite development platforms](#)” [*Introducing SQL Anywhere Studio*, page 99].

Developing UltraLite applications with Metrowerks CodeWarrior

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

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

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

Installing the UltraLite plug-in for CodeWarrior

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

❖ To install the UltraLite plug-in for CodeWarrior

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

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

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

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

Uninstalling the
CodeWarrior plug-in

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

Creating UltraLite projects in CodeWarrior

❖ To create an UltraLite project in CodeWarrior

1. Start CodeWarrior.
2. Create a new project.
 - a. From the CodeWarrior menu, choose File ► New. A tabbed dialog appears.
 - b. On the Projects tab, choose Palm OS Application Stationery.
 - c. Also on the Projects tab, choose a name and location for the project. Click OK.

3. Choose an UltraLite stationery.

The UltraLite plug-in adds the following choices to the stationery list:

- ◆ Palm OS UltraLite C++ API App
- ◆ Palm OS UltraLite C++ Component App
- ◆ Palm OS UltraLite ESQL App

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

The stationery is standard C stationery for embedded SQL, and standard C++ stationery for the static C++ API and C++ component. It contains almost-empty source files.

4. If you are using embedded SQL or the static C++ API, configure the target settings for your project.

If you are using the C++ component, these settings are ignored.

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

Preprocessing

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

When you build a C++ API project, the UltraLite plugin calls *ulgen* to generate the UltraLite API files and the database code.

There is no preprocessing required for C++ component, which is why the target settings are ignored.

For embedded SQL and the static C++ API, plugin also adds paths to required UltraLite files, such as headers and runtime library, to the search paths.

Converting an existing CodeWarrior project to an UltraLite application

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

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

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

2. For embedded SQL applications, remove all `.cpp` files generated by the SQL preprocessor from the Files view. These files are automatically generated and re-added when the `.sqc` files are built.
3. For static C++ API applications, mark the `.ulg` dummy file dirty and remove the UltraLite Files folder.

Using the UltraLite plug-in for CodeWarrior

For embedded SQL and the static C++ API, 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 static C++ API UltraLite project, or if you alter a publication used in a static C++ API project, you must manually touch the dummy `.ulg` file to prompt the UltraLite generator to run.

☞ For an overview of the tasks the plug-in carries out, see “[Configuring development tools for static UltraLite development](#)” [*UltraLite Database User’s Guide*, page 208].

Using prefix files

A **prefix file** is a header file that all source files in a Metrowerks

CodeWarrior project must include. You should use *ulpalmos.h* from the *h* subdirectory of your SQL Anywhere Studio installation as your prefix file. The CodeWarrior plug-in sets this for you automatically.

If you have your own prefix file, it must include *ulpalmos.h*. The *ulpalmos.h* file defines macros required by UltraLite Palm applications and also sets CodeWarrior compiler options required by UltraLite.

Notes

Although the UltraLite plug-in does not configure your development environment for expanded mode, you can build expanded mode applications. For more information, see [“Building Expanded Mode applications” on page 119](#).

If you are using either the *ULSecureCerticomTLSSStream* or *ULSecureRSATLSSStream* functions to implement encrypted synchronization, you must add *ulrsa.lib* or *ulecc.lib* to your CodeWarrior projects.

Building the CustDB sample application from CodeWarrior

CustDB is a simple sales-status application.

☞ For a diagram of the sample database schema, see [“The CustDB sample database” on page xvii](#).

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

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

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

❖ To build the CustDB sample application using CodeWarrior

1. Start the CodeWarrior IDE.
2. Open the CustDB project file:
 - ◆ Choose File ► Open.
 - ◆ Open the project file *Samples\UltraLite\custdb\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 115.

Building Expanded Mode applications

CodeWarrior supports a code generation mode called **expanded mode**, which improves memory use for global data. If you are using CodeWarrior version 9 you can now use expanded mode with an A5-based jump table. To do so, you must specify a specific version of the UltraLite runtime library. For expanded mode, specify the following runtime library:

```
ultralite\palm\68k\lib\cw9_a4a5jt\ulrt.lib
```

Expanded mode may be helpful for large applications that would otherwise exceed the 64 KB global data limit. A limitation of expanded mode is that encrypted synchronization can be used only via HotSync, as the synchronization security libraries for UltraLite do not use expanded mode.

Using the UltraLite plug-in

The UltraLite plug-in for CodeWarrior does not add the expanded mode library to applications.

The easiest way to build expanded mode applications from the UltraLite plug-in is to copy the expanded mode runtime library over the normal runtime library.

❖ To use the expanded mode runtime library from the UltraLite plug-in

1. Back up the standard runtime library.

Copy the file `ultralite\palm\68k\lib\cw\ulrt.lib` to a safe location.

2. Copy the expanded mode runtime library over the normal runtime library.

Copy `ultralite\palm\68k\lib\cw9_a4a5jt\ulrt.lib` over `ultralite\palm\68k\lib\cw\ulrt.lib`.

The UltraLite plug-in should then add the required paths, headers, and runtime library to the CW project.

3. Repeat the process if you install updated UltraLite software.

Installing updates to the UltraLite software will copy the new standard mode library over your expanded mode runtime library. Be sure to repeat the earlier steps in the procedure to use the expanded mode runtime library.

Saving state in UltraLite Palm applications

You can save the state of tables and cursors when an application is closed by suspending the connection instead of closing it.

The current state is only stored for tables that are not closed, and when the connection object is not closed. This means that, to maintain state, the application should terminate without ever closing the connection object; or exiting the routine that defined the connection object; or assigning the variable for the connection object to nothing or null.

Whenever your UltraLite application is closed, and the user switches to another application, UltraLite saves the state of any open cursors and tables.

1. When the user returns to the application, call the appropriate open methods:
 - ◆ For the Static C++ API, call the following functions:
 - ◆ `ULData::Open`,
 - ◆ `UEnablePalmRecordDB` or `UEnableFileDB`,
 - ◆ `ULConnection::Open`.
 - ◆ For embedded SQL, call the following functions:
 - ◆ `db_init`,
 - ◆ `UEnablePalmRecordDB` or `UEnableFileDB`,
 - ◆ `EXEC SQL CONNECT`.
 - ◆ For the C++ Component, call the following functions:
 - ◆ `ULSqlca.Initialize`,
 - ◆ `ULInitDatabaseManager`,
 - ◆ `UEnablePalmRecordDB` or `UEnableFileDB`,
 - ◆ `OpenDatabase`.
2. Confirm that the connection was restored properly by checking that the `SQLCODE` is `SQLE_CONNECTION_RESTORED`.
3. 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.

4. 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.

Closing a connection rolls back any uncommitted transactions. By not closing connection objects, any outstanding transactions are saved (not committed), so that when the application restarts, those transactions will appear and can be committed or rolled back. Also, uncommitted changes are not synchronized.

Restoring state in UltraLite Palm applications

When an application restarts on the Palm OS, UltraLite restores the state of any cursors or tables that were not explicitly closed when the application was most recently shut down.

Building multi-segment applications

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

☞ UltraLite applications include the following types of code:

- ◆ **User-defined code** Application code, including the *.cpp* file generated by the SQL Preprocessor.
- ◆ **Generated code for SQL statements** Code generated by the UltraLite Analyzer to execute SQL statements.
- ◆ **Generated code for the database schema** Code generated by the UltraLite Analyzer to represent the database tables.
- ◆ **Runtime library** The UltraLite runtime library is compiled as multi-segment code. Segment names of the form *ULRTn* and *ULRTnn* 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 124](#).

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 122](#).
- ◆ Enable multi-segment code-generation and explicitly assign segments yourself.
 - ☞ For more information, see [“Explicitly assigning segments” on page 123](#).

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 123](#).

You enable generated code segments by defining macros.

❖ To enable multi-segment code generation

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

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

☞ For more information, see [“UL_ENABLE_SEGMENTS macro” on page 222](#).

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 `.sql` file. All the code for a single `.sql` file goes into a single segment.

Explicitly assigning segments

This section describes how to explicitly assign the generated code for SQL statements to segments. You must first enable multi-segment code generation as described in [“Enabling multi-segment code generation” on page 122](#).

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

❖ To explicitly assign generated statement code to segments (embedded SQL)

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

❖ To explicitly assign generated statement code to segments (static C++ API)

1. 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](#)” [*UltraLite Database User's Guide*, page 212].

- ◆ 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)

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

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

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

The first segment

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

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

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

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

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

Adding HotSync synchronization to Palm applications

HotSync synchronization takes place when your UltraLite application is closed. It is initiated by the HotSync.

If you use HotSync, then you synchronize by calling `ULSetSynchInfo` before closing the application. Do not use **ULSynchronize** or **ULConnection.Synchronize** for HotSync synchronization.

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

1. Prepare a **ul_synch_info** structure.
2. Call the `ULSetSynchInfo` 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 **db_fini** or **ULData.Close**. The **ul_synch_info.stream** parameter is ignored, and so does not need to be set.

For example:

```
//Static C++ API
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" );

ULSetSynchInfo( &sqlca, &info );

if( !db.Close( ) ) {
    return( false );
}
```

3. Call `db_fini` or `ULData.Close`.

☞ For more information, see “Saving state in UltraLite Palm applications” on page 120, and “Synchronization parameters” [*MobiLink Clients*, page 316].

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

Specifying the stream parameters

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

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

- ◆ Supply the required information in the `stream_parms` member of `ul_synch_info` passed to `ULSetSynchInfo`.
 - ☞ For a list of available values, see “[Network protocol options for UltraLite synchronization clients](#)” [*MobiLink Clients*, page 341].
- ◆ Supply a null value for the `stream_parms` member. The MobiLink conduit then searches in the *ClientParams* registry entry on the machine where it is running for information on how to connect to the MobiLink synchronization server.

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

☞ For more information, see “[HotSync configuration overview](#)” [*MobiLink Clients*, page 298].

See also

☞ For information about configuring HotSync, including a description of how to set up your MobiLink HotSync conduit, see “[Configuring the MobiLink HotSync conduit](#)” [*MobiLink Clients*, page 301].

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 [“Synchronizing data” on page 51](#).

Transport layer security on the Palm Computing Platform

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

☞ For information on transport-layer security, see [“MobiLink Transport-Layer Security” \[MobiLink Administration Guide, page 165\]](#).

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** or **ULConnection.Synchronize** to carry out the synchronization.

When using TCP/IP, HTTP, or HTTPS synchronization, `db_init` or `ULData.Initialize` and `db_fini` or `ULData.Close` save and restore the state of the application on exiting and activating the application, but do not participate in synchronization.

Before closing the application, set the synchronization information using `ULSetSynchInfo`, providing **ul_synch_info** structure as an argument.

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 [“Network protocol options for UltraLite synchronization clients” \[MobiLink Clients, page 341\]](#).

Deploying Palm applications

This section describes the following aspects of deploying Palm applications:

- ◆ Deploying the application.
 - ☞ See [“Deploying the application” on page 128](#)
- ◆ Deploying the MobiLink synchronization conduit for HotSync.
 - ☞ See [“Deploying the MobiLink synchronization conduit” on page 128](#).
- ◆ Deploying an initial copy of the UltraLite database.
 - ☞ See [“Deploying UltraLite databases” on page 128](#).

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

Deploying the application

❖ To install an application on a Palm device

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

Deploying the MobiLink synchronization conduit

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

☞ For more information about installing the MobiLink synchronization conduit, see [“Deploying the MobiLink HotSync conduit”](#) [*MobiLink Clients*, page 302].

Deploying UltraLite databases

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

☞ For more information, see [“The ULUtil utility”](#) [*UltraLite Database User’s Guide*, page 123].

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

☞ For information on installing the synchronization conduit, see [“Configuring the MobiLink HotSync conduit”](#) [*MobiLink Clients*, page 301].

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 222.

CHAPTER 7

Developing UltraLite Applications for Windows CE

About this chapter

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

Contents

Topic:	page
Introduction	132
Building the CustDB sample application	134
Storing persistent data	136
Deploying Windows CE applications	137
Synchronization on Windows CE	140

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 “UltraLite development platforms” [*Introducing SQL Anywhere Studio*, page 99], and “UltraLite target platforms” [*Introducing SQL Anywhere Studio*, page 109].

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://msdn.microsoft.com/>.

A first application

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

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

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

```
$(ASANY9)\win32\sqlpp" -q -c "dsn=UltraLite 9.0 Sample"
$(InputPath) ul_database.cpp
```
 - Set the output file to *ul_database.cpp*.

- Disable the use of precompiled headers for *ul_database.cpp*.

Choosing how to link the runtime library

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

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

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

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

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

Building the CustDB sample application

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

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

☞ For a diagram of the sample database schema, see [“The CustDB sample database”](#) on page xvii.

❖ To build the CustDB sample application

1. Start eMbedded Visual C++.
2. Open the project file that corresponds to your version of eMbedded Visual C++:
 - ◆ *Samples\UltraLite\CustDB\EVC\EVCCustDB.vcp* for eVC 3.0.
 - ◆ *Samples\UltraLite\CustDB\EVC40\EVCCustDB.vcp* for eVC 4.0.
3. Choose Build ► Set Active Platform to set the target platform.
 - ◆ Set a platform of your choice.
4. Choose Build->Set Active Configuration to select the configuration.
 - ◆ Set an active configuration of your choice.
5. If you are building CustDB for the Pocket PC x86em emulator platform only:
 - ◆ Choose Project ► Settings. The Project Settings dialog appears.
 - ◆ On the Link tab, in the Object/library modules box, change the UltraLite runtime library entry to the *emulator30* directory rather than the *emulator* directory.
6. Build the application:
 - ◆ Press F7 or select Build ► Build EVCCustDB.exe to build CustDB. When eMbedded Visual C++ has finished building the application, it automatically attempts to upload it to the remote device.
7. Start the synchronization server:
 - ◆ To start the MobiLink synchronization server, select Programs ► Sybase SQL Anywhere 9 ► MobiLink ► Synchronization Server Sample.
8. Run the CustDB application:
 - ◆ Press CTRL+F5 or select Build ► Execute CustDB.exe

Folder locations and environment variables

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

For embedded SQL, 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 [“Building embedded SQL applications” on page 97](#).

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 path name of the file-based persistent store.

The UltraLite runtime carries out no substitutions on the **file_name** parameter. If a directory has to be created in order for the file name to be valid, the application must ensure that any directories are created before calling **db_init**.

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

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

Example

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

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

Deploying Windows CE applications

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

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

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

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

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

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

Deploying applications that use ActiveSync

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

☞ For more information, see “Deploying applications that use ActiveSync” [*MobiLink Clients*, page 312], “Installing the MobiLink provider for ActiveSync” [*MobiLink Clients*, page 310] and “Registering applications for use with ActiveSync” [*MobiLink Clients*, page 311].

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 141](#).

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 140
- ◆ TCP/IP See “TCP/IP, HTTP, or HTTPS synchronization from Windows CE” on page 143.
- ◆ HTTP See “TCP/IP, HTTP, or HTTPS synchronization from Windows CE” on page 143.

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 “Synchronizing data” on page 51. For detailed information on synchronization parameters, see “Synchronization parameters” [*MobiLink Clients*, page 316].

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 137.

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 376](#).

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 137](#).

❖ To add ActiveSync synchronization in the main dialog class

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

Find the message map in the source file for your main dialog (the name is of the same form as *CMyAppDlg.cpp*). Add a registered message

using the **static** and declare a message handler using `ON_REGISTERED_MESSAGE` as in the following example:

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

2. Implement the message handler.

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

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

The return value of this function should be 0.

☞ For information on handling the synchronization message, see [“ULIsSynchronizeMessage function” on page 376](#).

❖ 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 376](#).

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

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

☞ For more information, see [“Synchronizing data” on page 51](#).



PART III

TUTORIALS

This part provides tutorials that walk you through the development of a simple UltraLite application.

CHAPTER 8

Tutorial: Build an Application Using the C++ Component

About this chapter

This chapter provides a tutorial to guide you through the process of building a simple UltraLite C++ Component application.

Contents

Topic:	page
Introduction	148
Lesson 1: Connect to the database	149
Lesson 2: Insert data into the database	156
Lesson 3: Select the rows from the table	158
Lesson 4: Add synchronization to your application	160
Lesson 5: Deploy to a Windows CE device	162

Introduction

This tutorial guides you through the process of building an UltraLite C++ application. The application is built for Windows operating systems, and runs at a command prompt.

This tutorial uses a text editor to edit the C++ files. You can also use any C++ development environment, such as Microsoft Visual Studio.

Timing

The tutorial takes about 30 minutes if you copy and paste the code. If you enter the code yourself, it takes significantly longer.

Competencies and experience

This tutorial assumes:

- ◆ you are familiar with the C++ programming language
- ◆ you have a C++ compiler installed on your computer
- ◆ you know how to create an UltraLite schema using the UltraLite Schema Painter

☞ For more information, see “[Lesson 1: Create an UltraLite database schema](#)” [*UltraLite Database User's Guide*, page 130].

Note

The synchronization portion of this tutorial requires SQL Anywhere Studio.

Goals

The goals for the tutorial are to gain competence and familiarity with the process of developing an UltraLite C++ Component application.

Lesson 1: Connect to the database

In the first procedure, you create a database schema. You then write, compile, and run a C++ application that creates a database using the schema you have created.

❖ To create a database schema

1. Create a directory to hold the files you create in this tutorial.

The remainder of this tutorial assumes that this directory is `c:\tutorial\cpp`. If you create a directory with a different name, use that directory instead of `c:\tutorial\cpp` throughout the tutorial.

2. Using the UltraLite Schema Painter, create a database schema in your new directory with the following characteristics.

☞ For more information about using the UltraLite Schema painter, see “Lesson 1: Create an UltraLite database schema” [*UltraLite Database User’s Guide*, page 130].

Schema file name: **tutcustomer.usm**

Table name: **customer**

Columns in customer:

Column Name	Data Type (Size)	Column allows NULL values?	Default value
id	integer	No	autoincrement
fname	char(15)	No	None
lname	char(20)	No	None
city	char(20)	Yes	None
phone	char(12)	Yes	555-1234

Primary key: ascending **id**

❖ To connect to an UltraLite database

1. In Microsoft Visual C++, choose File ► New.
2. On the Files tab, choose C++ Source File.
3. Save the file as `customer.cpp` in your tutorial directory.
4. Import the UltraLite libraries and use the UltraLite namespace.
Copy the code below into `customer.cpp`.

```
#include "uliface.h"
#include <stdio.h>
#include <tchar.h>
#include <assert.h>
using namespace UltraLite;
#define MAX_NAME_LEN 100
ULSqlca Sqlca;
```

☞ For information about how using the UltraLite namespace makes class declarations simpler, see [“Using the UltraLite namespace” on page 14](#).

5. Define connection parameters to connect to the database. In this example, the parameters are the location of the database and schema files.

In the following code, these locations are hard coded. In a real application, the locations would be specified at runtime. In addition, these connection parameters are sufficient only for connections in the development environment; additional parameters are needed for the application to run on a Windows CE device.

Copy the code below into *customer.cpp*.

```
static ul_char const * ConnectionParms =
    UL_TEXT( "UID=DBA;PWD=SQL" )
    UL_TEXT( ";DBF=tutcustomer.udb" )
    UL_TEXT( ";schema_file=tutcustomer.usm" );
```

☞ For more information about connection parameters, see [“Class UltraLite_Connection_iface” on page 236](#).

6. Define a method for error handling.

UltraLite provides a callback mechanism to notify the application of errors.

This is a sample callback function.

```
ul_error_action UL_GENNED_FN_MOD MyErrorCallBack(
    SQLCA *    sqlca,
    ul_error_kind kind,
    ul_void *  user_data,
    ul_char *  message_parameter )
{
    ul_error_action rc;

    (void) user_data;
    switch( kind ) {
case UL_ERROR_KIND_MEDIA_REMOVED:
    // Not handled in this sample: Prompt user to re-
    // insert media
    // (message_parameter contains the filename).
    rc = UL_ERROR_ACTION_ABORT;
    // use UL_ERROR_ACTION_RETRY to retry
    break;

case UL_ERROR_KIND_SQLCODE:
    switch( sqlca->sqlcode ){
    // The following errors are used for flow control,
    // and we don't want to
    // report them here:
    case SQLE_NOTFOUND:
    case SQLE_ULTRALITE_DATABASE_NOT_FOUND:
    break;
    case SQLE_CANNOT_ACCESS_SCHEMA_FILE:
    _tprintf( _TEXT(
"Error %ld: UltraLite schema file %s not found\n" ),
        sqlca->sqlcode, message_parameter );
    break;
    case SQLE_COMMUNICATIONS_ERROR:
    _tprintf( _TEXT(
"Error %ld: Communications error\n" ), sqlca->sqlcode
        );
    break;
    default:
    _tprintf( _TEXT(
```

```

        "Error %ld: %s\n" ), sqlca->sqlcode, message_parameter
    );
    break;
}
rc = UL_ERROR_ACTION_ABORT; // N/A for SQLCODE kind,
return ABORT
break;

default:
    // future error kinds
    assert( 0 );
    rc = UL_ERROR_ACTION_ABORT; // default action
    break;
}
return rc;
}

```

In UltraLite, two errors are used to control application flow. The `SQLE_ULTRALITE_DATABASE_NOT_FOUND` error is signaled on the first connection attempt (when only the schema file is present), and is used to prompt the application to create a database from the schema file. The `SQLE_NOTFOUND` error marks the end of a loop over a result set.

☞ For more information about error handling, see [“Handling errors” on page 34](#).

7. Define a method to open a connection to a database.

If the database file does not exist, a `SQLException` is thrown. The schema file is used to create a new database and establish a connection to it.

If the database file exists, a connection is established.

```
Connection * open_conn( DatabaseManager * dm ) {
    Connection * conn;
    conn = dm->OpenConnection( Sqlca, ConnectionParms );
    if( conn == NULL ) {
        if( Sqlca.GetSQLCode() == SQLE_ULTRALITE_DATABASE_NOT_
            FOUND ) {
            // The database doesn't exist yet -- create it using the
            // schema file.
            conn = dm->CreateAndOpenDatabase( Sqlca,
                ConnectionParms );
            if( conn == NULL ) {
                handle_error( _TEXT("create database") );
            } else {
                _tprintf( _TEXT("Connected to a new database.\n")
                    );
            }
        } else {
            handle_error( _TEXT("open database") );
        }
    } else {
        _tprintf( _TEXT("Connected to an existing database.\n")
            );
    }
    return conn;
}
```

8. Implement the main() method.

The main method carries out the following tasks.

- ◆ Instantiates a DatabaseManager object. All UltraLite objects are created from the DatabaseManager object.
- ◆ Registers the error handling function.
- ◆ Opens a connection to the database.
- ◆ Closes the connection and shuts down the database manager.

```

int main() {
    ul_char buffer[ 100 ];
    DatabaseManager * dm;
    Connection * conn;
    Sqlca.Initialize();
    ULRegisterErrorCallback(
        Sqlca.GetCA(), MyErrorCallBack,
        UL_NULL, buffer, 100 );
    dm = ULInitDatabaseManager( Sqlca );
    if( dm == UL_NULL ){
        // You may have mismatched UNICODE vs. ANSI
        runtimes.
        Sqlca.Finalize();
        return 1;
    }
    conn = open_conn( dm );
    if( conn == UL_NULL ){
        dm->Shutdown( Sqlca );
        Sqlca.Finalize();
        return 1;
    }
    dm->Shutdown( Sqlca );
    Sqlca.Finalize();
    return 0;
}

```

9. Compile and link the Customer class.

The method you use to compile the class depends on your compiler. The following instructions are for the Microsoft Visual C++ command line compiler using a makefile.

- ◆ From a command prompt, browse to your tutorial directory.
- ◆ Create a makefile named *makefile*.
- ◆ In the makefile, add directories to your include path as follows.

```

IncludeFolders= \
/I"$(ASANY9)\h"

```

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

```

LibraryFolders= \
/LIBPATH:"$(ASANY9)\ultralite\win32\386\lib"

```

- ◆ In the makefile, add libraries to your linker command line options as follows.

```

Libraries= \
ulimp.lib

```

The UltraLite runtime library, *ulimp.lib*, is an ASCII version of the library. If you choose the Unicode version, *ulimpw.lib*, you should add `/DUNICODE` to the compiler options.

- ◆ In the makefile, set the following compiler options all on one line.

```
CompileOptions=/c /nologo /W3 /Od /Zi /DWIN32 /DUL_USE_
DLL
```

- ◆ In the makefile, add an instruction for linking the application.

```
customer.exe: customer.obj
link /NOLOGO /DEBUG customer.obj $(LibraryFolders)
$(Libraries)
```

- ◆ In the makefile, add an instruction for compiling the application.

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

- ◆ Add an instruction to create a preprocessed version of the file. This step is included for debugging purposes.

```
customer.i: customer.cpp
cl $(CompileOptions) $(IncludeFolders) customer.cpp -
P
```

- ◆ Run the makefile as follows:

```
nmake
```

An executable named *customer.exe* is created.

10. Run the application.

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

Lesson 2: Insert data into the database

The following procedures demonstrate how to add data to a database.

❖ To add rows to your database

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

This procedure carries out the following tasks.

- ◆ Opens the table using the `connection->OpenTable()` method. You must open a `Table` object to carry out operations on the table.
- ◆ Obtains identifiers for the required columns of the table. The other columns in the table can accept NULL values or have a default value.
- ◆ If the table is empty, adds two rows. To insert each row, the code changes to insert mode using the `InsertBegin` method, sets values for each required column, and executes an insert to add the rows to the database.

The `commit` method is only required when you turn off `autocommit`. By default, `autocommit` is enabled but it may be disabled for better performance, or for multi-operation transactions.

- ◆ If the table is not empty, reports the number of rows in the table.
- ◆ Closes the `Table` object.
- ◆ Returns a boolean indicating whether the operation was completed.


```
bool do_insert( Connection * conn ) {
    Table * table = conn->OpenTable( UL_TEXT("customer") );
    if( table == NULL ) {
        return false;
    }
    if( table->GetRowCount() == 0 ) {
        _tprintf( UL_TEXT("Inserting two rows.\n") );
        table->InsertBegin();
        table->Set( UL_TEXT("fname"), UL_TEXT("Penny") );
        table->Set( UL_TEXT("lname"), UL_TEXT("Stamp") );
        table->Insert();
        CHECK_ERROR();
        table->InsertBegin();
        table->Set( UL_TEXT("fname"), UL_TEXT("Gene") );
        table->Set( UL_TEXT("lname"), UL_TEXT("Poole") );
        table->Insert();
        CHECK_ERROR();
        conn->Commit();
        CHECK_ERROR();
    } else {
        _tprintf( UL_TEXT("The table has %lu rows\n"),
            table->GetRowCount() );
    }
    table->Release();
    return true;
}
```

2. Call the `do_insert` method you have created.

Add the following line to the `main()` method, immediately after the call to `open_conn`.

```
do_insert(conn);
```

3. Compile your application by running `nmake`.
4. Run your application by typing `customer` at the command prompt.

Lesson 3: Select the rows from the table

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

❖ To list the rows in the table

1. Add the method below to *customer.cpp*. This method carries out the following tasks.

- ◆ Opens the Table object.
- ◆ Retrieves the column identifiers.
- ◆ Sets the current position before the first row of the table.
Any operations on a table are carried out at the current position. The position may be before the first row, on one of the rows of the table, or after the last row. By default, as in this case, the rows are ordered by their primary key value (id). To order rows in a different way, you can add an index to an UltraLite database and open a table using that index.
- ◆ For each row, the id and name are written out. The loop carries on until the Next method returns false, which occurs after the final row.
- ◆ Closes the Table object.

```
bool do_select( Connection * conn ) {
    Table * table = conn->OpenTable( UL_TEXT("customer") );
    if( table == NULL ) {
        return false;
    }
    TableSchema * schema = table->GetSchema();
    if( schema == NULL ) {
        table->Release();
        return false;
    }
    ul_column_num id_cid = schema->GetColumnID( UL_TEXT("id")
        );
    ul_column_num fname_cid = schema->GetColumnID( UL_
        TEXT("fname") );
    ul_column_num lname_cid = schema->GetColumnID( UL_
        TEXT("lname") );
    schema->Release();
    while( table->Next() ) {
        ul_char fname[ MAX_NAME_LEN ];
        ul_char lname[ MAX_NAME_LEN ];
        table->Get( fname_cid ).GetString( fname, MAX_NAME_LEN
            );
        table->Get( lname_cid ).GetString( lname, MAX_NAME_LEN
            );
        _tprintf( "id=%d, name=%s %s\n", (int)table->Get( id_cid
            ), fname, lname );
    }
    table->Release();
    return true;
}
```

2. Add the following line to the `main()` method, immediately after the call to the insert method:

```
do_select(conn);
```

3. Compile your application by running `nmake`.
4. Run your application by typing `customer` at the command prompt.

Lesson 4: Add synchronization to your application

This lesson synchronizes your application with a consolidated database running on your computer.

The following procedures add synchronization code to your application, start the MobiLink synchronization server, and run your application to synchronize.

Note

This lesson uses MobiLink synchronization, which is part of SQL Anywhere Studio. You must have SQL Anywhere Studio installed on your computer to carry out this lesson.

The UltraLite database you created in the previous lessons synchronizes with the UltraLite 9.0 Sample database. The UltraLite 9.0 sample database has a ULCustomer table whose columns match those in the customer table of your UltraLite database.

This lesson assumes that you are familiar with MobiLink synchronization.

❖ To add synchronization to your application

1. Add the method below to *customer.cpp*. This method carries out the following tasks.
 - ◆ Sets the synchronization stream to TCP/IP. Synchronization can also be carried out over HTTP, ActiveSync, or HTTPS. For more information, see “UltraLite Clients” [*MobiLink Clients*, page 277].
 - ◆ Sets the script version. MobiLink synchronization is controlled by scripts stored in the consolidated database. The script version identifies which set of scripts to use.
 - ◆ Sets `sendColumnNames` to true so the MobiLink synchronization server can generate synchronization scripts automatically.
☞ For more information, see “Generating scripts automatically” [*MobiLink Administration Guide*, page 230].
 - ◆ Sets the MobiLink user name. This value is used for authentication at the MobiLink synchronization server. It is distinct from the UltraLite database user ID, although in some applications you may wish to give them the same value.
 - ◆ Sets the `download_only` parameter to true. By default, MobiLink synchronization is two-way. This application uses download-only synchronization so that the rows in your table do not get uploaded to the sample database.

```
bool do_sync( Connection * conn ) {
    ul_synch_info info;
    conn->InitSynchInfo( &info );
    info.stream = ULStream();
    info.version = UL_TEXT( "ul_default" );
    info.user_name = UL_TEXT( "sample" );
    info.send_column_names = true;
    info.download_only = true;
    if( !conn->Synchronize( &info ) ) {
        handle_error( _TEXT("synchronize" ) );
        return false;
    }
    return true;
}
```

2. Add the following line to the `main()` method, immediately after the call to the insert method and before the call to the select method.

```
do_sync(conn);
```

3. Compile your application by running `nmake`.

❖ To synchronize your data

1. Start the MobiLink synchronization server.

From a command prompt, run the following command.

```
dbmlsrv9 -c "dsn=UltraLite 9.0 Sample" -v+ -zu+ -za
```

The `-zu+` and `-za` command line options provide automatic addition of users and generation of synchronization scripts.

☞ For more information about these options, see the “[MobiLink Synchronization Server Options](#)” [*MobiLink Administration Guide*, page 189].

2. Run your application by typing `customer` at the command prompt.

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

Lesson 5: Deploy to a Windows CE device

The following procedure demonstrates how to deploy an UltraLite C++ Component application to a Windows CE device.

❖ To deploy to a Windows CE device

1. Ensure that your device is connected to your computer.
2. Start File Explorer on your device.
Choose Start ► Programs ► File Explorer.
3. Create directories to hold the UltraLite runtime and application.
 - ◆ Navigate to the root of the device. The root may be named My Device or My Pocket PC.
 - ◆ Create a directory named *UltraLite*.
 - ◆ Open the *UltraLite* directory and create subdirectories named *lib* and *CustDB*.
\UltraLite\lib is the location for the UltraLite runtime files, and
\UltraLite\CustDB is the location for the application.
4. Copy the UltraLite runtime files to the Windows CE device.
You can now run the application on your Windows CE device. This completes the tutorial.

CHAPTER 9

Tutorial: Build an Application Using the Static C++ API

About this chapter

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

Contents

Topic:	page
Introduction to the UltraLite static C++ API	164
Lesson 1: Getting started	166
Lesson 2: Create an UltraLite database template	167
Lesson 3: Run the UltraLite generator	168
Lesson 4: Write the application source code	169
Lesson 5: Build and run your application	172
Lesson 6: Add synchronization to your application	174
Restore the sample database	176

Introduction to the UltraLite static C++ API

You can use the UltraLite static 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 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 simple application using the UltraLite static C++ API. The application is a Windows console application, developed using Microsoft Visual C++, which queries data in the ULProduct table of the UltraLite 9.0 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 *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 create 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

1. Create a directory to hold the files you will create. The remainder of the tutorial assumes that this directory is *c:\APITutorial*.

❖ To copy the sample database

1. Make a backup copy of the UltraLite 9.0 Sample database into the tutorial directory. The UltraLite 9.0 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 9.0 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 9.0 Sample database (CustDB) as a reference database. The copy you placed in the *APITutorial* directory serves as a backup.

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 publication in the reference database. The publication is simply a convenient device for assembling tables and column-based subsets of tables: there is no direct connection to SQL Remote or MobiLink synchronization.

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 tutorial uses SQL statements to define the UltraLite database.

☞ For an example of how to add SQL statements to a database, see [“Lesson 1: Getting started” on page 166](#).

❖ To create the UltraLite database template

1. Start Sybase Central.
2. Connect to the UltraLite 9.0 Sample database.
 - ◆ Choose Tools ► Connect.
 - ◆ If a list of plugins is displayed, choose Adaptive Server Anywhere 9.
 - ◆ In the Connect dialog, choose the UltraLite 9.0 Sample ODBC data source.
 - ◆ Click OK to connect.
3. Create a publication that describes the data you wish to include in your UltraLite database.
 - ◆ In Sybase Central, open the custdb database.
 - ◆ Open the Publications folder. Choose File ► New ► Publication. The Publication Creation wizard appears.
 - ◆ Name the publication ProductPub.
 - ◆ 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 9.0 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

The following procedure creates a C++ source file containing the application source code. This 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.

You can find the source code in *Samples\UltraLite\APITutorial\sample.cpp*, although you may have to edit the file to uncomment the inserts.

❖ To write the application source code

1. In Microsoft Visual C++, choose File ► New.
2. On the Files tab, choose C++ Source File. Click OK.
3. Copy and paste the following source code into a file named *sample.cpp* in your *APITutorial* directory.

```

// (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 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 174.

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 9\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 9\ultralite\win32\386\lib
```

Click OK to submit the changes.

3. Create a project named **APITutorial** (it should have 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 its location.
 - ◆ Select Create a 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 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 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 the Source Files folder and select Add Files to Folder. Locate *sample.cpp* 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 the Source Files folder and choose Add Files to Folder. Browse to *ulapi.cpp*, located in the *src* subdirectory of your SQL Anywhere installation. 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. 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 insert 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 instead 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 9.0" );
synch_info.stream = ULStream();
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 dbmlsrv9 -c "dsn=UltraLite 9.0 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
```

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 9.0 Sample database in [“Lesson 1: Getting started” on page 166](#). 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 10

Tutorial: Build an Application Using Embedded SQL

About this chapter

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

☞ For an overview of the development process and background information on the UltraLite database, see [“Developing embedded SQL applications” on page 7](#).

☞ For information on developing embedded SQL UltraLite Applications, see [“Developing Applications Using Embedded SQL” on page 61](#).

☞ For a description of embedded SQL, see [“Embedded SQL API Reference” on page 357](#).

Contents

Topic:	page
Introduction	178
Lesson 1: Configure eMbedded Visual C++	179
Lesson 2: Write an embedded SQL source file	180
Lesson 3: Build the sample embedded SQL UltraLite application	186
Lesson 4: Add synchronization to your application	187

Introduction

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

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

❖ To prepare for the tutorial

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

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

Lesson 1: Configure eMbedded Visual C++

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

❖ To configure eMbedded Visual C++ for UltraLite development

1. Start Microsoft eMbedded Visual C++ 3.0.

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

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

- a. Select Tools ► Options.

The Options dialog is displayed.

- b. Click the Directories tab

- c. For each target platform and CPU combination,

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

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

If you have installed SQL Anywhere to a directory other than the default, substitute the `h` subdirectory of your installation.

- ◆ Choose Library Files under the Show Directories For dropdown menu. Include the UltraLite `lib` directory, located in a platform-specific directory. For example, for the Pocket PC emulator, choose the following:

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

- d. Click OK.

Lesson 2: Write an embedded SQL source file

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

❖ To build the sample embedded SQL UltraLite application

1. Start Microsoft eMbedded Visual C++.
Choose Start ► Programs ► Microsoft eMbedded Visual Tools ► eMbedded Visual C++.
2. Create a new workspace named **UltraLite**:
 - ◆ Select File ► New.
 - ◆ Click the Workspaces tab.
 - ◆ Choose Blank Workspace. Specify a workspace name **UltraLite** and specify *C:\tutorial* as the location to save this workspace. Click OK. The **UltraLite** workspace is added to the Workspace window.
3. Create a new project named **esql** and add it to the **UltraLite** workspace.
 - ◆ Select File ► New.
 - ◆ Click the Projects tab.
 - ◆ Choose WCE Pocket PC 2002 Application. Specify a project name **esql** and select Add To Current Workspace. Select the applicable CPUs. Click OK.
 - ◆ Choose Create An Empty Project and click Finish. The project is saved in the *c:\tutorial\esql* folder.
4. Create the *sample.sqc* source file.
 - ◆ Choose File ► New.
 - ◆ Click the Files tab.
 - ◆ Select C++ Source File.
 - ◆ Select Add to Project and select esql from the dropdown list.
 - ◆ Name the file *sample.sqc*. Click OK.
 - ◆ Copy the following source code into the file:


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

```

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

```

- ◆ Save the file.

5. Configure the *sample.sqc* source file settings to invoke the SQL preprocessor to preprocess the source file:

- ◆ Right-click *sample.sqc* in the Workspace window and select Settings. The Project Settings dialog appears.
- ◆ From the Settings For drop down menu, choose All Configurations.
- ◆ In the Custom Build tab, enter the following statement in the Commands box. Ensure that the statement is entered all on one line. The following statement runs the SQL preprocessor *sqlpp* on the *sample.sqc* file, and writes the processed output in a file named *sample.cpp*. The SQL preprocessor translates SQL statements in the source file into C/C++.

```

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

```

☞ For more information about the SQL preprocessor, see “The SQL preprocessor” [ASA Programming Guide, page 203].

- ◆ Specify *sample.cpp* in the Outputs box.
- ◆ Click OK to submit the changes.

6. Start the Adaptive Server Anywhere personal database server.

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

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

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

```
dbeng9 custdb.db
```

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

7. Preprocess the *sample.sqc* file.

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

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

8. Add *sample.cpp* to the project:

- ◆ Right-click the Source Files folder in the Workspace window and select Add Files to Folder.
- ◆ Browse to *c:\tutorial\esql\sample.cpp* and click OK.
The *sample.cpp* file appears inside the Source Files folder.

Explanation of the sample program

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

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

1. Include the appropriate header files.

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

2. Define the SQL communications area, *sqlca*.

Use the following command:

```
EXEC SQL INCLUDE SQLCA;
```

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

Prefix SQL statements

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

3. Define host variables by creating a declaration section.

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

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

☞ For information about host variables, see [“Using host variables” on page 68](#).

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

Call this function as follows:

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

5. Connect to the database using the CONNECT statement.

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

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

6. Insert data into database tables.

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

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

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

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

function to synchronize the remote database with the consolidated database.

☞ For information on adding synchronization to an UltraLite application, see [“Lesson 4: Add synchronization to your application” on page 187](#).

7. Execute your SQL query.

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

8. Perform error handling.

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

9. Disconnect from the database.

You should rollback or commit any outstanding changes before disconnecting.

To disconnect, use the `DISCONNECT` statement as follows:

```
EXEC SQL DISCONNECT;
```

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

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

Lesson 3: Build the sample embedded SQL UltraLite application

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

❖ To build the sample embedded SQL UltraLite application

1. Ensure that the Adaptive Server Anywhere personal database server is still running.

2. Configure the project settings:

◆ Right-click **esql** and select Settings.

The Project Settings dialog appears.

◆ Select All Configurations under the Settings For drop down menu.

◆ Click the Link tab and add the following runtime library to the Object/Library Modules box.

```
ulimp.lib
```

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

```
__NT__
```

◆ Click OK to close the dialog.

3. Build the executable:

◆ Select Build ► Build esql.exe.

The **esql** executable is created. Depending on your settings, the executable may be created in a Debug directory within your tutorial directory.

4. Run the application:

◆ Select Build ► Execute esql.exe.

A screen appears and displays the first row of the product table.

Lesson 4: Add synchronization to your application

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

Synchronization via TCP/IP

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

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

☞ For more information about the `ULSynchronize` function, see [“ULSynchronize function” on page 388](#).

Running the sample application with synchronization

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

❖ To synchronize your application

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

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

3. Preprocess *sample.sqc*.

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

4. Build the executable.

Select Build ► Build *esql.exe* to build the sample executable.

5. Start the MobiLink synchronization server.

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

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

6. Run the application:

- ◆ Select Build ► Execute *esql.exe* to run the sample application.

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

CHAPTER 11

Tutorial: Build an Application Using ODBC

About this chapter

This chapter walks you through the creation of an UltraLite ODBC application.

Contents

Topic:	page
Introduction to UltraLite ODBC	190
Lesson 1: Getting started	191
Lesson 2: Create an UltraLite database schema file	193
Lesson 3: Connect to the database	194
Lesson 4: Insert data into the database	197
Lesson 5: Query the database	198

Introduction to UltraLite ODBC

ODBC is a standard database programming interface. UltraLite supports a subset of the ODBC interface, together with extensions to permit synchronization. For a listing of the functions UltraLite supports, see [“UltraLite ODBC API Reference” on page 389](#).

This section walks you through the creation of a simple UltraLite ODBC application. It does not provide an extensive guide to ODBC programming, as the main reference for ODBC is the Microsoft [ODBC SDK documentation](#).

UltraLite ODBC does not share some features with other C/C++ interfaces. In particular, the functions listed in [“UltraLite C/C++ Common API Reference” on page 203](#) cannot be used from UltraLite ODBC.

Lesson 1: Getting started

In this tutorial, you will create a set of files, including source files and executable files. You should make a directory to hold these files. In the remainder of the tutorial, it is assumed the directory is `c:\tutorial\ulodbc`. If you choose a different name, use that name throughout.

The ODBC interface does not depend on any particular C/C++ compiler or development environment. The tutorial uses a makefile with Microsoft's `nmake` syntax. If you are using a different development environment, make the appropriate substitutions.

❖ Create and test your build environment

1. Add the following code to a file called *makefile* in your tutorial directory:

```

IncludeFolders= /I"${ASANY9}\h"

CompileOptions=/c /nologo /W3 /Od /Zi /DWIN32 /DUL_USE_DLL

LibraryFolders= \
/LIBPATH:"${ASANY9}\ultralite\win32\386\lib"

Libraries= ulimp.lib

LinkOptions=/NOLOGO /DEBUG

sample.exe: sample.obj
    link $(LinkOptions) sample.obj $(LibraryFolders)
        $(Libraries)

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

```

These options compile a source file called *sample.cpp* into an executable *sample.exe*, using the UltraLite import library for Windows (*ulimp.lib*). They rely on the environment variable `ASANY9`, which is defined as your SQL Anywhere installation directory.

2. Add the following code to a file called *sample.cpp* in your tutorial directory:

```

#include "ulodbc.h"
#include <stdio.h>
#include <tchar.h>
int main() {
    return 0;
}

```

This application simply returns 0 to the calling environment.

3. Compile and link *sample.cpp*.

If you are using the Microsoft compiler, type **nmake** at a command prompt to compile and link your application. Otherwise, use the command appropriate for your development environment.

Compiling and linking the application confirms that your build environment is set up properly. You are now ready for the rest of the tutorial.

Lesson 2: Create an UltraLite database schema file

The database schema holds the table definitions. This tutorial uses a simple one-table database. It is the same schema as used in other UltraLite component tutorials.

This section assumes you can use the UltraLite Schema Painter to create a schema file.

☞ For more information about creating a database schema, see the “[Lesson 1: Create an UltraLite database schema](#)” [*UltraLite Database User’s Guide*, page 130].

❖ Create a database schema

1. Create a database schema using the UltraLite Schema Painter.

To start the UltraLite Schema Painter, choose Start ► Programs ► SQL Anywhere Studio 9 ► UltraLite ► UltraLite Schema Painter.

Create your schema as follows:

◆ **Schema filename** `c:\tutorial\ulodbc\customer.usm`

◆ **Table name** customer

◆ **Columns in customer**

Column Name	Data Type (Size)	Column allows NULL values?	Default value
id	integer	No	autoincrement
fname	char(15)	No	None
lname	char(20)	No	None
city	char(20)	Yes	None
phone	char(12)	Yes	555-1234

◆ **Primary key** ascending id

Lesson 3: Connect to the database

UltraLite uses standard ODBC programming methods to connect to a database. Each application requires an environment handle to manage the communication with UltraLite and a connection handle for a specific connection.

❖ Write code to allocate an environment handle

1. Add functions **opendb** and **closedb** to *sample.cpp*:

```
static SQLHANDLE opendb( void ){
    SQLRETURN retn;
    SQLHANDLE henv;
    retn = SQLAllocHandle( SQL_HANDLE_ENV, SQL_NULL_HANDLE,
        &henv );
    if( retn == SQL_SUCCESS ){
        _tprintf( "success in opendb: %d.\n", retn );
        return henv;
    } else {
        _tprintf( "error in opendb: %d.\n", retn );
        return henv;
    }
}

static void closedb( SQLHANDLE henv ){
    SQLRETURN retn;
    retn = SQLFreeHandle( SQL_HANDLE_ENV, henv );
}
```

These functions do not connect to the database, they simply allocate the environment handle `henv` that manages UltraLite features.

2. Call `opendb` and `closedb` from the `main()` function.

Alter your `main()` function in *sample.cpp* so that it reads as follows:

```
int main() {
    SQLHANDLE henv;
    henv = opendb();
    closedb( henv );
    return 0;
}
```

3. Compile, link, and run your application to confirm that the application builds properly.

☞ For more information about the functions called in this procedure, see [“SQLAllocHandle function” on page 391](#), and [“SQLFreeHandle function” on page 402](#).

The next step is to connect to the UltraLite database

❖ Write code to connect to your database

1. Add functions **connect** and **disconnect** to *sample.cpp*:

```

static SQLHANDLE connect ( SQLHANDLE henv ){
    SQLRETURN retn;
    SQLHANDLE hcon;
    retn = SQLAllocHandle( SQL_HANDLE_DBC, henv, &hcon );
    retn = SQLConnect( hcon
        , (SQLTCHAR*)UL_TEXT(
            "schema_file=customer.usm;dbf=customer.udb" )
        , SQL_NTS
        , (SQLTCHAR*)UL_TEXT( "dba" )
        , SQL_NTS
        , (SQLTCHAR*)UL_TEXT( "sql" )
        , SQL_NTS );
    if( retn == SQL_SUCCESS ){
        _tprintf( "success in connect: %d.\n", retn );
        return hcon;
    } else {
        _tprintf( "error in connect: %d.\n", retn );
        return hcon;
    }
}

static void disconnect( SQLHANDLE hcon, SQLHANDLE henv ){
    SQLRETURN retn;
    retn = SQLDisconnect( hcon );
    retn = SQLFreeHandle( SQL_HANDLE_DBC, hcon );
}

```

2. Call **connect** and **disconnect** from the **main()** function.

Alter your **main()** function in *sample.cpp* so that it reads as follows:

```

int main() {
    SQLHANDLE henv;
    SQLHANDLE hcon;
    henv = opendb();
    hcon = connect( henv );
    disconnect( hcon, henv );
    closedb( henv );
    return 0;
}

```

3. Compile, link, and run your application to confirm that the application builds properly.

You should see that the application creates an UltraLite database file *customer.udb* together with a temporary file *customer.~db*.

☞ For more information about the functions called in this procedure, see “SQLConnect function” on page 394, and “SQLDisconnect function” on page 396.

You now have an application that connects to and disconnects from a database. The next step is to add some data to the database.

Lesson 4: Insert data into the database

ODBC provides a set of functions to carry out operations on the database. In this lesson we use the simplest statement, `SQLExecDirect`.

❖ Write code to insert data into the database

1. Add a function `insert` to `sample.cpp`:

```
static ul_bool insert( SQLHANDLE hcon )
{
    SQLRETURN retn;
    SQLHANDLE hstmt;
    retn = SQLAllocHandle( SQL_HANDLE_STMT, hcon, &hstmt );
    static const ul_char * sql = UL_TEXT(
        "INSERT customer( id, fname, lname ) VALUES ( 42,
        'jane', 'doe' )" );
    retn = SQLExecDirect( hstmt, (SQLTCHAR*)sql, SQL_NTS );
    if( retn == SQL_SUCCESS ) {
        _tprintf( "success in insert.\n" );
    } else {
        _tprintf( "error in insert: %d.\n", retn );
    }
    retn = SQLFreeHandle( SQL_HANDLE_STMT, hstmt );
    hstmt = 0;
}
return retn == SQL_SUCCESS;
}
```

2. Call `insert` from the `main()` function.

Alter your `main()` function in `sample.cpp` so that it reads as follows:

```
int main() {
    SQLHANDLE henv;
    SQLHANDLE hcon;
    henv = opendb();
    hcon = connect( henv );
    insert( hcon );
    disconnect( hcon, henv );
    closedb( henv );
    return 0;
}
```

3. Compile, link, and run your application to confirm that the application builds properly.

You should see that the application reports success on inserting the data.

☞ For more information about the function called in this procedure, see [“SQLExecDirect function” on page 398](#).

Lesson 5: Query the database

In order to process query result sets, ODBC requires that statements be prepared before they are executed. In this lesson you prepare and execute a statement, and print out the results.

❖ Write code to query the database

1. Add functions **prepare**, **execute**, and **fetch** to *sample.cpp*:

```
static SQLHANDLE prepare( SQLHANDLE hcon ){
    SQLRETURN retn;
    SQLHANDLE hstmt;
    static const ul_char * sql =
        UL_TEXT( "SELECT id, fname, lname FROM customer" );
    retn = SQLAllocHandle( SQL_HANDLE_STMT, hcon, &hstmt );
    retn = SQLPrepare( hstmt, (SQLTCHAR*)sql, SQL_NTS );
    if( retn == SQL_SUCCESS ) {
        _tprintf( "success in prepare.\n" );
    } else {
        _tprintf( "error in prepare: %d.\n", retn );
        retn = SQLFreeHandle( SQL_HANDLE_STMT, hstmt );
        hstmt = 0;
    }
    return hstmt;
}
```

The prepare function does not execute the SQL statement.

```
static ul_bool execute( SQLHANDLE hstmt )
{
    SQLRETURN retn;
    retn = SQLExecute( hstmt );
    if( retn == SQL_SUCCESS ) {
        _tprintf( "success in execute.\n" );
    } else {
        _tprintf( "error in execute: %d.\n", retn );
        retn = SQLFreeHandle( SQL_HANDLE_STMT, hstmt );
        hstmt = 0;
    }
    return retn == SQL_SUCCESS;
}
```

The execute function executes the query, but does not make the result set directly available to the client application. Your application must explicitly fetch the rows it needs from the result set.

```

static ul_bool fetch( SQLHANDLE hstmt )
{
#define NAME_LEN 20
    SQLCHAR      fName[NAME_LEN], lName[NAME_LEN];
    SQLINTEGER    id;
    SQLINTEGER    cbID = 0, cbFName = SQL_NTS, cbLName =
        SQL_NTS;
    SQLRETURN     retn;

    SQLBindCol( hstmt, 1, SQL_C_ULONG, &id, 0, &cbID );
    SQLBindCol( hstmt, 2, SQL_C_CHAR,
        fName, sizeof(fName), &cbFName );
    SQLBindCol( hstmt, 3, SQL_C_CHAR,
        lName, sizeof(lName), &cbLName );
    while ( ( retn = SQLFetch( hstmt ) ) != SQL_NO_DATA ) {
        if (retn == SQL_SUCCESS || retn == SQL_SUCCESS_WITH_
            INFO) {
            fName[ cbFName ] = '\0';
            lName[ cbLName ] = '\0';
            _tprintf( "%20s %d %20s\n", fName, id, lName );
        } else {
            _tprintf ( "error while fetching: %d.\n", retn
                );
            break;
        }
    }
    return retn == SQL_SUCCESS;
}

```

The values are fetched into variables that have been bound to the column. String variables are not returned with a null terminator, and so the null terminator is added for printing purposes. The length of the actual string that was returned is available in the final parameter of `SQLBindCol`.

2. Call prepare, execute, and fetch from the main() function.

Alter your `main()` function in `sample.cpp` so that it reads as follows:

```

int main() {
    SQLHANDLE henv;
    SQLHANDLE hcon;
    SQLHANDLE hstmt;

    henv = opendb();
    hcon = connect( henv );
    insert( hcon );
    hstmt = prepare( hcon );
    execute( hstmt );
    fetch( hstmt );
    SQLFreeHandle( SQL_HANDLE_STMT, hstmt );
    disconnect( hcon );
    closedb( henv );
    return 0;
}

```

-
3. Compile, link, and run your application to confirm that the application builds properly.

You should see that the application prints out the row you inserted.

☞ For more information about the functions called in this procedure, see [“SQLPrepare function” on page 408](#), [“SQLExecute function” on page 399](#), [“SQLBindCol function” on page 392](#), and [“SQLFetch function” on page 400](#).

This completes the tutorial.

PART IV

API REFERENCE

This part provides API reference material for UltraLite C/C++ programmers.

CHAPTER 12

UltraLite C/C++ Common API Reference

About this chapter

This chapter lists functions and macros that are for use from any of the embedded SQL, static C++ API, or C++ Component interfaces. They cannot be used from ODBC.

The functions in this chapter all require a SQL Communications Area. For more information, see [“Common Features of UltraLite C/C++ Interfaces”](#) on page 105.

Contents

Topic:	page
Callback function for ULRegisterErrorCallback	204
Callback function for ULRegisterSchemaUpgradeObserver	206
ULEnableFileDB function	208
ULEnableGenericSchema function (deprecated)	209
ULEnablePalmRecordDB function	210
ULEnableStrongEncryption function	211
ULEnableUserAuthentication function	212
ULRegisterErrorCallback function	213
ULRegisterSchemaUpgradeObserver function	216
ULStoreDefragFini function	218
ULStoreDefragInit function	219
ULStoreDefragStep function	220
Macros and compiler directives for UltraLite C/C++ applications	221

Callback function for ULRegisterErrorCallback

Handle errors that the UltraLite runtime signals to your application.

Prototype

```
ul_error_action UL_GENNED_FN_MOD error-callback-function(  
SQLCA *                sqlca,  
ul_error_kind          kind,  
ul_void *              user_data,  
ul_char *              buffer  
)
```

Parameters

◆ **error-callback-function** The name of your function. You must supply the name to ULRegisterErrorCallback. See [“ULRegisterErrorCallback function” on page 213](#).

◆ **sqlca** A pointer to the SQL communications area (SQLCA).

The SQLCA contains the SQL code in `sqlca->sqlcode`. Any error parameters have already been retrieved from the SQLCA and stored in *buffer*.

This `sqlca` pointer does not necessarily point to the SQLCA in your application, and cannot be used to call back to UltraLite. It is used only to communicate the SQL code to the callback.

In the static C++ API the SQLCA is declared in the header file as **sqlca**. In the C++ Component use the `Sqlca.GetCA()` method.

◆ **kind** The kind of error that UltraLite is signaling. One of the following constants:

• **UL_ERROR_KIND_MEDIA_REMOVED** The media card containing the database has been removed and must be re-inserted for the current I/O operation to succeed.

A common action to take for this kind of error is to prompt the user to insert the media card.

For this kind of error, the buffer holds the database filename.

• **UL_ERROR_KIND_SQLCODE** A SQL error. The SQLCA contains the SQL code in `sqlca->sqlcode`. SQL codes and their meanings are listed in [“Error messages indexed by Adaptive Server Anywhere SQLCODE” \[ASA Error Messages, page 2\]](#). The returned error action is ignored for this kind of error; the operation is always canceled.

◆ **user_data** The user data supplied to ULRegisterErrorCallback.

UltraLite does not change this data in any way. As the callback function may be signaled anywhere in your application, the `user_data` argument is an alternative to creating a global variable.

	<ul style="list-style-type: none"> ◆ buffer The buffer supplied when the callback function was registered. UltraLite fills the buffer with a string holding any substitution parameters for the error message. To keep UltraLite as small as possible, UltraLite does not supply error messages themselves. The substitution parameters depend on the specific error. You can look up the error parameters for SQL errors in “Database Error Messages” [ASA Error Messages, page 1].
Return value	<p>Returns one of the following actions:</p> <ul style="list-style-type: none"> ◆ UL_ERROR_ACTION_CANCEL Cancel the operation that raised the error. ◆ UL_ERROR_ACTION_CONTINUE Continue execution, ignoring the operation that raised the error. ◆ UL_ERROR_ACTION_DEFAULT Behave as if there is no error callback. This value is particularly appropriate for <code>UL_ERROR_KIND_SQLCODE</code>. ◆ UL_ERROR_ACTION_TRY_AGAIN Try the operation again that raised the error.
Description	<p>☞ For a description of error handling using this technique, see “ULRegisterErrorCallback function” on page 213.</p>
See also	<ul style="list-style-type: none"> ◆ “ULRegisterErrorCallback function” on page 213 ◆ “Error messages indexed by Adaptive Server Anywhere SQLCODE” [ASA Error Messages, page 2]

Callback function for ULRegisterSchemaUpgradeObserver

Enables applications to display progress during UltraLite database schema upgrades.

Prototype

```
ul_ret_void UL_GENNED_FN_MOD upgrade-callback-function(  
p_ul_schema_upgrade_status status  
)
```

Parameters

- ◆ **upgrade-callback-function** The name of your function. You must supply the name to ULRegisterSchemaUpgradeObserver.
- ◆ **status** A pointer to a schema upgrade structure, which has the following definition:

```
typedef struct {  
    ul_schema_upgrade_state state;  
    ul_u_long                progress_counter;  
    ul_u_long                final_progress_count;  
    ul_u_long                upgrade_operations;  
    ul_bool                  stop;  
    ul_void *                user_data;  
}
```

The status fields are as follows:

- ◆ **state** One of the following:
 - **UL_UPGRADE_STATE_STARTING** The upgrade is starting. The user can safely cancel the operation at this stage.
 - **UL_UPGRADE_STATE_UPGRADING** The upgrade is in progress.
 - **UL_UPGRADE_STATE_ABORT** The schema upgrade is canceled as a result of a recoverable error or as a result of user action. The old database is preserved.
 - **UL_UPGRADE_STATE_ERROR** A critical error occurred. The database is unusable.
 - **UL_UPGRADE_STATE_DONE** The upgrade completed successfully.
- ◆ **progress_counter** An approximation of the progress so far. The value is a number between zero and `final_progress_count`, enabling you to display a percentage complete value in a dialog box.
- ◆ **final_progress_count** The value of the `progress_counter` when the upgrade completes successfully.

- ◆ **upgrade_operations** An approximation of the amount of work done during the upgrade. The value starts at zero and increases as the upgrade proceeds. It is updated more frequently than `progress_counter`. It can be used as a relative measure to compare against other schema upgrades.
- ◆ **stop** To cancel the schema upgrade, set this value to `ul_true` when the state is `UL_UPGRADE_STATE_STARTING`. Your application will receive a second callback with `UL_UPGRADE_STATUS_ABORT`.

In embedded SQL and the Static C++ API, the connect operation fails with `SQLCODE SQLE_SCHEMA_UPGRADE_NOT_ALLOWED`. In the C++ Component, the `UpgradeSchemaFromFile` call returns false and the `SQLCODE` is set to `SQLCODE_SCHEMA_UPGRADE_NOT_ALLOWED`.

The stop value is ignored if you set stop to `ul_true` when the state is not `UL_UPGRADE_STATE_STARTING`.
- ◆ **user_data** This is identical to the `user_data` parameter passed to `ULRegisterSchemaUpgradeObserver` or `UpgradeSchemaFromFile`.

ULEnableFileDB function

Use a file-based data store on a device operating the Palm Computing Platform version 4.0 or later.

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

Parameters **sqlca** A pointer to the SQLCA. This argument is supplied even in C++ Component and static C++ API applications.

In the static C++ API the SQLCA is declared in the header file as **sqlca**. In the C++ Component use the `Sqlca.GetCA()` method.

Description 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 connecting to the database.

Examples The following embedded SQL code illustrates the use of ULEnableFileDB.

```
db_init( & sqlca );
ULEnablePalmRecordDB( &sqlca );
// connection code here
if( SQLCODE == SQLE_CONNECTION_RESTORED ){
    // connection was restored
    // cursor is already open
} else {
    // open cursor
}
```

See also [“ULEnablePalmRecordDB function” on page 210](#)

UEnableGenericSchema function (deprecated)

Upgrade the database schema when deploying a new version of an application.

Deprecated feature

This statement is deprecated. Use `ULRegisterSchemaUpgradeObserver` instead.

Prototype

```
void UEnableGenericSchema( SQLCA * sqlca );
```

Parameters

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

In the static C++ API the SQLCA is declared in the header file as **sqlca**. In the C++ Component use the `Sqlca.GetCA()` method.

Description

When a new UltraLite application is deployed to a device, UltraLite by default re-creates an empty database, losing any data that was in the database before the new application was deployed. If you call `UEnableGenericSchema`, 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 `db_init` or `ULData.Open()`. An exception is the Palm Computing Platform, where there is no need to close all cursors before upgrading. Immediately following an upgrade on the Palm Computing Platform the `LAUNCH_SUCCESS_FIRST` launch code is returned.

Backup before upgrading

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

For more information about the schema upgrade process, see [“How schema upgrade works”](#) [*UltraLite Database User’s Guide*, page 56].

UEnablePalmRecordDB function

Use a standard record-based data store on a device operating the Palm Computing Platform.

Prototype

```
void UEnablePalmRecordDB( SQLCA * sqlca );
```

Parameters

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

In the static C++ API the SQLCA is declared in the header file as **sqlca**. In the C++ Component use the `Sqlca.GetCA()` method.

Description

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

Examples

The following embedded SQL code illustrates the use of `UEnablePalmRecordDB`.

```
db_init( & sqlca );
UEnablePalmRecordDB( &sqlca );
// connection code here
if( SQLCODE == SQLE_CONNECTION_RESTORED ){
    // connection was restored
    // cursor is already open
} else {
    // open cursor
}
```

See also

[“UEnableFileDB function” on page 208](#)

ULEnableStrongEncryption function

Strongly encrypt an UltraLite database.

Prototype void **ULEnableStrongEncryption**(SQLCA * *sqlca*)

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

In the static C++ API the SQLCA is declared in the header file as **sqlca**. In the C++ Component use the `Sqlca.GetCA()` method.

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

See also [“Encrypting data” on page 49](#)

ULEnableUserAuthentication function

Enable user authentication in the UltraLite application.

Prototype

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

Parameters

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

In the static C++ API the SQLCA is declared in the header file as **sqlca**. In the C++ Component use the `Sqlca.GetCA()` method.

Description

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 a connection is opened.

See also

[“User authentication in UltraLite”](#) [*UltraLite Database User's Guide*, page 40]

ULRegisterErrorCallback function

Register a callback function that handles errors.

Prototype

```
void ULRegisterErrorCallback (
    SQLCA *          sqlca,
    ul_error_callback_fn callback,
    ul_void *        user_data,
    ul_char *        buffer,
    size_t           len
)
```

Parameters

- ◆ **sqlca** A pointer to the SQL Communications Area.
In the static C++ API the SQLCA is declared in the header file as **sqlca**. In the C++ Component use the `Sqlca.GetCA()` method.
- ◆ **callback** The name of your callback function. For information about the prototype of the function, see [“Callback function for ULRegisterErrorCallback” on page 204](#).
A callback value of `UL_NULL` disables any previously registered callback function.
- ◆ **user_data** As the callback function may be called from any location in your application, any context information you wish it to access would have to be globally accessible. As an alternative to global variables, use this field to supply any context information you wish your function to have access to.
UltraLite does not modify the supplied data, it simply passes it to your callback function when it is invoked.
You can declare any data type here and cast it into the correct type in your callback function. For example, you could include a line of the following form in your callback function:

```
MyContextType * context = (MyContextType *)user_data;
```
- ◆ **buffer** A character array holding the substitution parameters for the error message, including a null terminator. To keep UltraLite as small as possible, UltraLite does not supply error messages. The substitution parameters depend on the specific error. You can look up the error parameters for SQL errors in [“Database Error Messages” \[ASA Error Messages, page 1\]](#).
The buffer must exist as long as UltraLite is active. Supply `UL_NULL` if you do not want to receive parameter information.

-
- ◆ **len** The length of the buffer, in `ul_char` characters. A value of 100 is large enough to hold most error parameters. If the buffer is too small, the parameters are truncated safely.

Description

Once this function has been called, the user-supplied callback function is called whenever UltraLite signals an error. You should therefore call `ULRegisterErrorCallback` immediately after initializing the SQL Communications Area.

Error handling using this callback technique is particularly helpful during development, as it ensures that your application is notified of any and all errors that occur. However, the callback function does not control execution flow, so the callback function does not replace other error handling code.

Example

The following code registers a callback function for an UltraLite C++ Component application:

```
int main(){
    ul_char buffer[100];
    DatabaseManager * dm;
    Connection * conn;
    Sqlca.Initialize();
    ULRegisterErrorCallback(
        Sqlca.GetCA(),
        MyErrorCallBack,
        UL_NULL,
        buffer,
        100 );
    dm = ULInitDatabaseManager( Sqlca );
    ...
}
```

The following is a sample callback function:

```
ul_error_action UL_GENNED_FN_MOD MyErrorCallBack(
    SQLCA * sqlca,
    ul_error_kind kind,
    ul_void * user_data,
    ul_char * message_parameter ){
    ul_error_action rc;
    (void) user_data;

    switch( kind ) {
    case UL_ERROR_KIND_MEDIA_REMOVED:
        // TODO: Prompt user to re-insert media
        // (message_parameter contains the filename).
        rc = UL_ERROR_ACTION_ABORT; // use UL_ERROR_ACTION_RETRY
        to retry
        break;
    }
```

```

case UL_ERROR_KIND_SQLCODE:
    switch( sqlca->sqlcode ){
        // The following errors are used for flow control, and
        // we don't want to
        // report them here:
        case SQLE_NOTFOUND:
        case SQLE_ULTRALITE_DATABASE_NOT_FOUND:
            break;
        case SQLE_CANNOT_ACCESS_SCHEMA_FILE:
            _tprintf(
                _TEXT( "Error %ld: UltraLite schema file %s not
                found\n" ),
                sqlca->sqlcode,
                message_parameter );
            break;
        case SQLE_COMMUNICATIONS_ERROR:
            _tprintf(
                _TEXT( "Error %ld: Communications error\n" ),
                sqlca->sqlcode );
            break;
        default:
            _tprintf(
                _TEXT( "Error %ld: %s\n" ),
                sqlca->sqlcode,
                message_parameter );
            break;
    }
    rc = UL_ERROR_ACTION_ABORT; // ignored for SQL kind
    break;
default:
    // future error kinds
    assert( 0 );
    rc = UL_ERROR_ACTION_ABORT; // default action
    break;
}
return rc;
}

```

See also

- ◆ [“Error messages indexed by Adaptive Server Anywhere SQLCODE”](#)
[*ASA Error Messages*, page 2]
- ◆ [“Callback function for ULRegisterErrorCallback”](#) on page 204

ULRegisterSchemaUpgradeObserver function

Prototype

```
ul_ret_void ULRegisterSchemaUpgradeObserver (  
SQLCA * sqlca,  
ul_schema_upgrade_observer_fn callback,  
ul_void * user_data  
);
```

Parameters

- ◆ **sqlca** A pointer to the SQL Communications Area.
In the static C++ API the SQLCA is declared in the header file as **sqlca**.
In the C++ Component use the `Sqlca.GetCA()` method.
- ◆ **callback** The name of your callback function. For information about the prototype of the function, see [“Callback function for ULRegisterErrorCallback” on page 204](#).
A callback value of `UL_NULL` disables any previously registered callback function.
- ◆ **user_data** As the callback function may be called from any location in your application, any context information you wish it to access would have to be globally accessible. As an alternative to global variables, use this field to supply any context information you wish your function to have access to.
UltraLite does not modify the supplied data, it simply passes it to your callback function when it is invoked.
You can declare any data type here and cast it into the correct type in your callback function. For example, you could include a line of the following form in your callback function:

```
MyContextType * context = (MyContextType *)user_data;
```

Description

The schema upgrade process can be time consuming. You can use an observer callback to monitor the upgrade process and provide feedback to your end user. For example, you can use this feature to provide a dialog box that displays percentage progress to the user.

Call `ULRegisterSchemaUpgradeObserver` after the call to `db_init` (embedded SQL) or `ULData::Open` (Static C++ API), but before connecting to the database. The upgrade takes place during the connect operation, at which time callbacks are made to your registered callback function.

Although you can call `ULRegisterSchemaUpgradeObserver` from the C++ Component, you can also use the `Connection.UpgradeSchemaFromFile` method to carry out the same action. If your application calls both, `ULRegisterSchemaUpgradeObserver` is used.

You must call `ULRegisterSchemaUpgradeObserver` in applications that upgrade the database schema. If you do not wish to provide feedback to the user, pass `UL_NULL` as the callback function. If your application does not upgrade a database schema, you may wish to omit the call to `ULRegisterSchemaUpgradeObserver`, as it can add significantly to application size.

See also

[“Upgrading UltraLite database schemas”](#) [*UltraLite Database User’s Guide*, page 54]

ULStoreDefragFini function

Prototype	<code>ul_ret_void ULStoreDefragFini(SQLCA * <i>sqlca</i>, p_ul_store_defrag_info <i>dfi</i>);</code>
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 110 “ULStoreDefragInit function” on page 219

ULStoreDefragInit function

Prototype	<code>p_ul_store_defrag_info ULStoreDefragInit(SQLCA * <i>sqlca</i>);</code>
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 110 “ULStoreDefragFini function” on page 218

ULStoreDefragStep function

Prototype	<code>ul_bool ULStoreDefragStep(SQLCA * <i>sqlca</i> p_ul_store_defrag_info <i>dfi</i>);</code>
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 110 “ULStoreDefragFini function” on page 218 “ULStoreDefragInit function” on page 219

Macros and compiler directives for UltraLite C/C++ applications

This section describes compiler directives to supply for UltraLite C/C++ applications. Unless stated otherwise, directives apply to both embedded SQL and C++ API applications.

Compiler directives can be supplied on your compiler command line or in the compiler settings dialog box of your user interface. Alternatively, they can be defined in source code.

On the compiler command line, a compiler directive is commonly set by using the /D command-line option. For example, to compile an UltraLite application with user authentication, a makefile for the Microsoft Visual C++ compiler may look as follows:

```
CompileOptions=/c /DPRWIN32 /Od /Zi /DWIN32
/D_NT_ /DUL_USE_DLL /DULB_USE_BIGINT_TYPES
/DULB_USE_FLOAT_TYPES /DUL_ENABLE_USER_AUTH

IncludeFolders= \
/I"$(VCDIR)\include" \
/I"$(ASANY9)\h"

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

where *VCDIR* is your Visual C++ directory and *ASANY9* is your SQL Anywhere directory.

In source code, directives are supplied using the #define statement.

UL_AS_SYNCHRONIZE macro

Function	Provides the name of the callback message used to indicate an ActiveSync synchronization.
Applies to	Windows CE applications using ActiveSync only.
See also	“Adding ActiveSync synchronization to your application” on page 140 “Adding ActiveSync synchronization to your application” on page 140

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 data” on page 49](#)

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 [“Authenticating users” on page 47](#)

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 122](#)

[“Enabling multi-segment code generation” on page 122](#)

UL_OMIT_COLUMN_INFO macro

Function Reduce the number of Palm OS data segments required by the generated code.

Usage Define the preprocessor symbol `UL_OMIT_COLUMN_INFO` before compiling generated files. In CodeWarrior, you can do this by adding the `#define` to your prefix file.

The macro decreases the number of segments required by the generated code. To accomplish the reduction, it omits extended schema information that the UltraLite generator would otherwise write. As a result, when this macro is defined, you cannot upgrade a database schema, and you cannot use the `send_column_names` synchronization parameter.

UL_STORE_PARMS macro

Function Supply a set of keyword-value pairs to configure database storage.

Syntax **`#define UL_STORE_PARMS UL_TEXT("keyword=value;... ")`**

All spaces in the keyword-value list are significant, except spaces at the start of the string and any spaces that immediately follow a semicolon.

Usage Define the `UL_STORE_PARMS` macro in the header of your application

source code so that it is visible to all `db_init()` calls.

Parameters

Keywords are case insensitive. The case sensitivity of the values depends on the application interpreting it. For example, the case sensitivity of the filename depends on the operating system.

☞ For a list of available parameters, see [“Database Schema parameters” \[UltraLite Database User’s Guide, page 78\]](#), and [“Additional connection parameters” \[UltraLite Database User’s Guide, page 82\]](#).

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 `_sprintf` 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. */
. . .
_sprintf( store_parms, UL_TEXT("cache_size=%lu"),
          cache_size );
db_init( &sqlca );
```

Here is a similar example that uses the password as an encryption key, and which sets the location and file name of the UltraLite database file. The code assumes that the variable `filename` already contains the desired path and filename. The code must be executed before calling `db_init`:

```
TCHAR store_parms[100];
#undef  UL_STORE_PARMS
#define UL_STORE_PARMS  store_parms
//put the password in UL_STORE_PARMS to be used as encryption
key
my_sprintf( store_parms, UL_TEXT("key=%s"), app_pword );

#ifdef UNDER_CE
if( filename != NULL ) {
    my_strcat( store_parms, _T(";file_name=") );
    my_strcat( store_parms, filename );
}
#endif
#endif
```

See also

[“Database Schema parameters” \[UltraLite Database User’s Guide, page 78\]](#)

[“Additional connection parameters” \[UltraLite Database User’s Guide, page 82\]](#)

[“Creating UltraLite database schema files” on page 15](#)

[“Encrypting data” on page 49](#)

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 432](#)

[“ULGetLastDownloadTime function” on page 369](#)

[“ULCountUploadRows function” on page 366](#)

[“UL_SYNC_ALL_PUBS macro” on page 224](#)

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 432](#)

[“ULGetLastDownloadTime function” on page 369](#)

[“ULCountUploadRows function” on page 366](#)

[“UL_SYNC_ALL macro” on page 224](#)

UL_TEXT macro

Function Prepares constant strings to be compiled as single-byte strings or wide-character strings. In embedded SQL and C++ API applications, use this macro to enclose all constant strings so that the compiler handles these parameters correctly.

UL_USE_DLL macro

Function For Windows CE and Windows applications only, define this directive to use the runtime library DLL, rather than a static runtime library.

Applies to The generated database code.

UNDER_CE macro

Function Use this macro when compiling UltraLite applications for Windows CE only. By default, this macro is defined in all new eMbedded Visual C++ projects.

See also [“Developing UltraLite Applications for Windows CE” on page 131.](#)

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 117.](#)

See also [“Developing UltraLite Applications for the Palm Computing Platform” on page 113.](#)

CHAPTER 13

UltraLite C++ Component API Reference

About this chapter

This chapter describes the UltraLite C++ Component API.

Contents

Topic:	page
Class ULSqlca	229
Class ULSqlcaBase	230
Class ULSqlcaWrap	233
Class UltraLite_Connection	234
Class UltraLite_Connection_iface	236
Class UltraLite_Cursor_iface	245
Class UltraLite_DatabaseManager	249
Class UltraLite_DatabaseManager_iface	250
Class UltraLite_DatabaseSchema	252
Class UltraLite_DatabaseSchema_iface	253
Class UltraLite_IndexSchema	256
Class UltraLite_IndexSchema_iface	257
Class UltraLite_PreparedStatement	260
Class UltraLite_PreparedStatement_iface	261
Class UltraLite_ResultSet	263
Class UltraLite_ResultSet_iface	264
Class UltraLite_ResultSetSchema	265
Class UltraLite_RowSchema_iface	266
Class UltraLite_SQLObject_iface	269
Class UltraLite_StreamReader	271
Class UltraLite_StreamReader_iface	272
Class UltraLite_StreamWriter	275

Topic:	page
Class UltraLite_Table	276
Class UltraLite_Table_iface	278
Class UltraLite_TableSchema	285
Class UltraLite_TableSchema_iface	286
Class ULValue	291

Class ULSqlca

Synopsis	public ULSqlca
Base classes	◆ “Class ULSqlcaBase” on page 230
Remarks	<p>Class ULSqlcaBase subclass which contains a SQLCA structure so an external one is not required.</p> <p>This is used in most C++ Component applications. The communication area must be initialized before any other functions are called. Each thread requires its own communication area.</p>
Members	<p>All members of ULSqlca, including all inherited members.</p> <ul style="list-style-type: none"> ◆ “Finalize Function” on page 230 ◆ “GetCA Function” on page 230 ◆ “GetParameter Function” on page 231 ◆ “GetParameterCount Function” on page 231 ◆ “GetSQLCode Function” on page 231 ◆ “GetSQLCount Function” on page 231 ◆ “GetSQLErrorOffset Function” on page 232 ◆ “Initialize Function” on page 232 ◆ “LastCodeOK Function” on page 232 ◆ “LastFetchOK Function” on page 232 ◆ “ULSqlca Function” on page 229 ◆ “~ULSqlca Function” on page 229

ULSqlca Function

Synopsis	ULSqlca::ULSqlca()
Remarks	Constructor.

~ULSqlca Function

Synopsis	ULSqlca::~~ULSqlca()
Remarks	Destructor.

Class `ULSqlcaBase`

Synopsis	<code>public ULSqlcaBase</code>
Derived classes	<ul style="list-style-type: none">◆ “Class <code>ULSqlca</code>” on page 229◆ “Class <code>ULSqlcaWrap</code>” on page 233
Remarks	<p>Provides a communication area.</p> <p><code>ULSqlcaBase</code> defines the communication area between the interface library and the application. Use a subclass of this class (typically Class <code>ULSqlca</code>) to create your communication area. There is always an underlying <code>SQLCA</code> object required. The communication area must be initialized before any other functions are called. Each thread requires its own communication area.</p>
Members	<p>All members of <code>ULSqlcaBase</code>, including all inherited members.</p> <ul style="list-style-type: none">◆ “Finalize Function” on page 230◆ “GetCA Function” on page 230◆ “GetParameter Function” on page 231◆ “GetParameterCount Function” on page 231◆ “GetSQLCode Function” on page 231◆ “GetSQLCount Function” on page 231◆ “GetSQLErrorOffset Function” on page 232◆ “Initialize Function” on page 232◆ “LastCodeOK Function” on page 232◆ “LastFetchOK Function” on page 232

Finalize Function

Synopsis	<code>void ULSqlcaBase::Finalize()</code>
Remarks	<p>Finalizes this communication area.</p> <p>Until the communication area is initialized again, it cannot be used.</p>

GetCA Function

Synopsis	<code>SQLCA * ULSqlcaBase::GetCA()</code>
Remarks	Gets the <code>SQLCA</code> structure for direct access to additional fields.
Returns	Raw <code>sqlca</code> structure.

GetParameter Function

Synopsis	<pre> size_t ULSqlcaBase::GetParameter(ul_u_long <i>parm_num</i> ul_char * <i>buffer</i> size_t <i>size</i>) </pre>
Parameters	<ul style="list-style-type: none"> ◆ parm_num A 1-based parameter number. ◆ buffer The buffer to receive parameter string. ◆ size The size, in ul_chars, of the buffer.
Remarks	<p>Gets the error parameter string.</p> <p>The output parameter string is always null-terminated, even if the buffer is too small and the parameter is truncated. The parameter number is 1-based.</p>
Returns	<p>If the function succeeds, the return value is the buffer size required to hold the entire parameter string (number of ul_chars, including the null terminator). If the function fails, the return value is zero. The function fails if an invalid (out of range) parameter number is given.</p>

GetParameterCount Function

Synopsis	<pre> ul_u_long ULSqlcaBase::GetParameterCount() </pre>
Remarks	Gets the error parameter count for last operation.
Returns	The number of parameters for the current error.

GetSQLCode Function

Synopsis	<pre> an_sql_code ULSqlcaBase::GetSQLCode() </pre>
Remarks	Gets the error code (SQLCODE) for last operation.
Returns	The sqlcode value

GetSQLCount Function

Synopsis	<pre> an_sql_code ULSqlcaBase::GetSQLCount() </pre>
Remarks	<p>Gets the sql count variable (SQLCOUNT) for the last operation.</p> <p>This indicates the number of rows affected by an INSERT, DELETE, or UPDATE operation, and is zero otherwise.</p>

GetSQLErrorOffset Function

Synopsis	ul_s_long ULSqlcaBase::GetSQLErrorOffset()
Remarks	Gets the error offset in dynamic SQL statement.
Returns	When applicable, the return value is the offset into the associated dynamic SQL statement (passed to the PrepareStatement function) corresponding to the current error. When not applicable, the return value is -1.

Initialize Function

Synopsis	bool ULSqlcaBase::Initialize()
Remarks	Initializes this communication area. You must initialize the communications area before any other operations occur.
Returns	On success, true, otherwise false. This method can fail if basic interface library initialization fails, which could occur if system resources are depleted.

LastCodeOK Function

Synopsis	bool ULSqlcaBase::LastCodeOK()
Remarks	Tests the error code for the last operation.
Returns	true if the sqlcode is SQLE_NOERROR or a warning; false if it indicates an error.

LastFetchOK Function

Synopsis	bool ULSqlcaBase::LastFetchOK()
Remarks	Tests the error code for last fetch operation. Use this function only immediately after performing a fetch operation.
Returns	true if the sqlcode indicates that a row was fetched successfully by the last operation.

Class ULSqlcaWrap

Synopsis	<code>public ULSqlcaWrap</code>
Base classes	◆ “Class ULSqlcaBase” on page 230
Remarks	<p>Class ULSqlcaBase subclass which attaches to an existing SQLCA object.</p> <p>This can be used with a previously-initialized SQLCA object (in which case, you would NOT call Initialize again). The communication area must be initialized before any other functions are called. Each thread requires its own communication area.</p>
Members	<p>All members of ULSqlcaWrap, including all inherited members.</p> <ul style="list-style-type: none"> ◆ “Finalize Function” on page 230 ◆ “GetCA Function” on page 230 ◆ “GetParameter Function” on page 231 ◆ “GetParameterCount Function” on page 231 ◆ “GetSQLCode Function” on page 231 ◆ “GetSQLCount Function” on page 231 ◆ “GetSQLErrorOffset Function” on page 232 ◆ “Initialize Function” on page 232 ◆ “LastCodeOK Function” on page 232 ◆ “LastFetchOK Function” on page 232 ◆ “ULSqlcaWrap Function” on page 233 ◆ “~ULSqlcaWrap Function” on page 233

ULSqlcaWrap Function

Synopsis	<pre>ULSqlcaWrap::ULSqlcaWrap(SQLCA * sqlca)</pre>
Parameters	◆ sqlca The SQLCA object to use.
Remarks	<p>Constructor.</p> <p>You may initialize the given SQLCA object before creating this object. In this case, don’t call Initialize Function again.</p>

~ULSqlcaWrap Function

Synopsis	<code>ULSqlcaWrap::~~ULSqlcaWrap()</code>
Remarks	Destructor.

Class UltraLite_Connection

Synopsis	public UltraLite_Connection
Base classes	<ul style="list-style-type: none">◆ “Class UltraLite_SQLObject_iface” on page 269◆ “Class UltraLite_Connection_iface” on page 236
Remarks	Connection class.
Members	<p>All members of UltraLite_Connection, including all inherited members.</p> <ul style="list-style-type: none">◆ “AddRef Function” on page 269◆ “ChangeEncryptionKey Function” on page 237◆ “Commit Function” on page 237◆ “CountUploadRows Function” on page 237◆ “GetConnection Function” on page 269◆ “GetConnectionNum Function” on page 237◆ “GetDatabaseID Function” on page 237◆ “GetDatabaseProperty Function” on page 237◆ “GetIFace Function” on page 269◆ “GetLastDownloadTime Function” on page 238◆ “GetLastIdentity Function” on page 238◆ “GetNewUUID Function” on page 238◆ “GetPublicationMask Function” on page 238◆ “GetSchema Function” on page 238◆ “GetSqlca Function” on page 239◆ “GetSuspend Function” on page 239◆ “GetSynchResult Function” on page 239◆ “GetUtilityULValue Function” on page 239◆ “GlobalAutoincUsage Function” on page 239◆ “GrantConnectTo Function” on page 239◆ “InitSynchInfo Function” on page 240◆ “OpenTable Function” on page 240◆ “OpenTableWithIndex Function” on page 240◆ “PrepareStatement Function” on page 241◆ “Release Function” on page 270◆ “ResetLastDownloadTime Function” on page 241◆ “RevokeConnectFrom Function” on page 241◆ “Rollback Function” on page 241◆ “RollbackPartialDownload Function” on page 241◆ “SetDatabaseID Function” on page 241◆ “SetSuspend Function” on page 242◆ “SetSynchInfo Function” on page 242◆ “Shutdown Function” on page 242◆ “StartSynchronizationDelete Function” on page 242

- ◆ “StopSynchronizationDelete Function” on page 242
- ◆ “StrToUUID Function” on page 243
- ◆ “Synchronize Function” on page 243
- ◆ “UpgradeSchemaFromFile Function” on page 244
- ◆ “UpgradeSchemaFromFile Function” on page 244
- ◆ “UUIDToStr Function” on page 243
- ◆ “UUIDToStr Function” on page 244

Class UltraLite_Connection_iface

Synopsis	public UltraLite_Connection_iface
Derived classes	◆ “Class UltraLite_Connection” on page 234
Remarks	Connection interface.
Members	<p>All members of UltraLite_Connection_iface, including all inherited members.</p> <ul style="list-style-type: none">◆ “ChangeEncryptionKey Function” on page 237◆ “Commit Function” on page 237◆ “CountUploadRows Function” on page 237◆ “GetConnectionNum Function” on page 237◆ “GetDatabaseID Function” on page 237◆ “GetDatabaseProperty Function” on page 237◆ “GetLastDownloadTime Function” on page 238◆ “GetLastIdentity Function” on page 238◆ “GetNewUUID Function” on page 238◆ “GetPublicationMask Function” on page 238◆ “GetSchema Function” on page 238◆ “GetSqlca Function” on page 239◆ “GetSuspend Function” on page 239◆ “GetSynchResult Function” on page 239◆ “GetUtilityULValue Function” on page 239◆ “GlobalAutoincUsage Function” on page 239◆ “GrantConnectTo Function” on page 239◆ “InitSynchInfo Function” on page 240◆ “OpenTable Function” on page 240◆ “OpenTableWithIndex Function” on page 240◆ “PrepareStatement Function” on page 241◆ “ResetLastDownloadTime Function” on page 241◆ “RevokeConnectFrom Function” on page 241◆ “Rollback Function” on page 241◆ “RollbackPartialDownload Function” on page 241◆ “SetDatabaseID Function” on page 241◆ “SetSuspend Function” on page 242◆ “SetSynchInfo Function” on page 242◆ “Shutdown Function” on page 242◆ “StartSynchronizationDelete Function” on page 242◆ “StopSynchronizationDelete Function” on page 242◆ “StrToUUID Function” on page 243◆ “Synchronize Function” on page 243◆ “UpgradeSchemaFromFile Function” on page 244

- ◆ “UpgradeSchemaFromFile Function” on page 244
- ◆ “UUIDToStr Function” on page 243
- ◆ “UUIDToStr Function” on page 244

ChangeEncryptionKey Function

Synopsis virtual bool **UltraLite_Connection_iface::ChangeEncryptionKey**(
 const ULValue & *new_key*
)

Parameters ◆ **new_key** The new encryption key value for the database.

Remarks Changes the encryption key.

Commit Function

Synopsis virtual bool **UltraLite_Connection_iface::Commit**()

Remarks Commits the current transaction.

CountUploadRows Function

Synopsis virtual ul_u_long **UltraLite_Connection_iface::CountUploadRows**(
 ul_publication_mask *mask*
 ul_u_long *threshold*
)

Parameters ◆ **mask** The set of publications to consider.

◆ **threshold** The limit on the number of rows to count.

Remarks Determines the number of rows that need to be uploaded.

GetConnectionNum Function

Synopsis virtual ul_connection_num **UltraLite_Connection_iface::GetConnectionNum**()

Remarks Gets the connection number.

GetDatabaseID Function

Synopsis virtual ul_u_long **UltraLite_Connection_iface::GetDatabaseID**()

Remarks Gets the database ID used for global autoincrement columns.

GetDatabaseProperty Function

Synopsis virtual ULValue **UltraLite_Connection_iface::GetDatabaseProperty**(
 ul_database_property_id *id*
)

Parameters	◆ id The ID of the property being requested.
Remarks	Gets the Database Property.
Returns	The value of the requested property.

GetLastDownloadTime Function

Synopsis virtual bool **UltraLite_Connection_iface::GetLastDownloadTime**(
 ul_publication_mask *mask*
 DECL_DATETIME * *value*
)

Parameters	◆ mask The publication mask.
	◆ value output: the last download time.

Remarks Gets the time of the last download.

GetLastIdentity Function

Synopsis virtual ul_u_big **UltraLite_Connection_iface::GetLastIdentity**()

Remarks Gets the @@identity value.

GetNewUUID Function

Synopsis virtual bool **UltraLite_Connection_iface::GetNewUUID**(
 p_ul_binary *uuid*
)

Parameters ◆ **uuid** The new UUID value.

Remarks Creates a new UUID.

GetPublicationMask Function

Synopsis virtual ul_publication_mask **UltraLite_Connection_iface::GetPublicationMask**(
 const ULValue & *pub_id*
)

Parameters ◆ **pub_id** The publication name or ordinal.

Remarks Gets the publication mask for a given publication name.

Publication masks are not publication IDs. 0 is returned if the publication is not found.

GetSchema Function

Synopsis virtual UltraLite_DatabaseSchema * **Ultra-**
 Lite_Connection_iface::GetSchema()

Remarks Gets the database schema.

GetSqlca Function

Synopsis virtual ULSqlcaBase const & **UltraLite_Connection_iface::GetSqlca()**

Remarks Gets the communication area associated with this connection.

GetSuspend Function

Synopsis virtual bool **UltraLite_Connection_iface::GetSuspend()**

Remarks Gets the Suspend property.

Returns true if this connection is suspended, false otherwise.

GetSynchResult Function

Synopsis virtual bool **UltraLite_Connection_iface::GetSynchResult**(
 p_ul_synch_result *synch_result*
)

Parameters ♦ **synch_result** A pointer to the ul_synch_result structure that holds the synchronization results.

Remarks Gets the result of the last synchronization.

GetUtilityULValue Function

Synopsis virtual ULValue **UltraLite_Connection_iface::GetUtilityULValue()**

Remarks Gets a new [Class ULValue](#).

A [Class ULValue](#) object must be bound to a connection in order for many of its methods to succeed.

GlobalAutoincUsage Function

Synopsis virtual ul_u_short **UltraLite_Connection_iface::GlobalAutoincUsage()**

Remarks Gets the percent usage of the global autoincrement counter.

GrantConnectTo Function

Synopsis virtual bool **UltraLite_Connection_iface::GrantConnectTo**(
 const ULValue & *uid*
 const ULValue & *pwd*
)

Parameters	<ul style="list-style-type: none"> ◆ uid The user ID being granted authority to connect. ◆ pwd The password the user ID must specify to connect.
Remarks	Adds a new user or changes an existing user's password.

InitSynchInfo Function

Synopsis	<pre>virtual void UltraLite_Connection_iface::InitSynchInfo(p_ul_synch_info <i>info</i>)</pre>
Parameters	<ul style="list-style-type: none"> ◆ info A pointer to the <code>ul_synch_info</code> structure that holds the synchronization parameters.
Remarks	Initializes the synchronization information structure.

OpenTable Function

Synopsis	<pre>virtual UltraLite_Table * UltraLite_Connection_iface::OpenTable(const ULValue & <i>table_id</i> const ULValue & <i>persistent_name</i>)</pre>
Parameters	<ul style="list-style-type: none"> ◆ table_id The table name or ordinal. ◆ persistent_name The instance name used for suspending.
Remarks	<p>Opens a table.</p> <p>When a table is first opened, the cursor position is BeforeFirst()</p>

OpenTableWithIndex Function

Synopsis	<pre>virtual UltraLite_Table * UltraLite_Connection_iface::OpenTableWithIndex(const ULValue & <i>table_id</i> const ULValue & <i>index_id</i> const ULValue & <i>persistent_name</i>)</pre>
Parameters	<ul style="list-style-type: none"> ◆ table_id The table name or ordinal. ◆ index_id The index name or ordinal. ◆ persistent_name The instance name used for suspending.
Remarks	<p>Opens a table, using a specified index to order the rows.</p> <p>When a table is first opened, the cursor position is BeforeFirst()</p>

Parameters	◆ value The database ID, which determines the starting value for global autoincrement columns.
Remarks	Sets the database ID used for global autoincrement columns.

SetSuspend Function

Synopsis	virtual void UltraLite_Connection_iface::SetSuspend (bool <i>suspend</i>)
Parameters	◆ suspend Set to true to suspend the connection so that its state can be restored when the database is reopened.
Remarks	Sets the Suspend property. If true, this connection is suspended and restored when the database is reopened. The connection name (or lack of one) is used to identify suspended connections.

SetSynchInfo Function

Synopsis	virtual bool UltraLite_Connection_iface::SetSynchInfo (p_ul_synch_info <i>info</i>)
Parameters	◆ info A pointer to the ul_synch_info structure that holds the synchronization parameters.
Remarks	Attaches a ul_synch_info struct to the current database.

Shutdown Function

Synopsis	virtual void UltraLite_Connection_iface::Shutdown ()
Remarks	Destroys this connection and any remaining associated objects. If not set to suspend, this connection is rolled back.

StartSynchronizationDelete Function

Synopsis	virtual bool UltraLite_Connection_iface::StartSynchronizationDelete ()
Remarks	START SYNCHRONIZATION DELETE for this connection.

StopSynchronizationDelete Function

Synopsis	virtual bool UltraLite_Connection_iface::StopSynchronizationDelete ()
Remarks	STOP SYNCHRONIZATION DELETE for this connection.

StrToUUID Function

Synopsis	virtual bool UltraLite_Connection_iface::StrToUUID (p_ul_binary <i>dst</i> size_t <i>len</i> const ULValue & <i>src</i>)
Parameters	<ul style="list-style-type: none"> ◆ dst The UUID value being returned. ◆ len The length of the ul_binary array. ◆ src A string holding the UUID value to be converted.
Remarks	Converts a string to a UUID.

Synchronize Function

Synopsis	virtual bool UltraLite_Connection_iface::Synchronize (p_ul_synch_info <i>info</i>)
Parameters	◆ info A pointer to the ul_synch_info structure that holds the synchronization parameters.
Remarks	Synchronizes the database.

Example:

```
ul_synch_info info;
conn.InitSynchInfo( &info );
info.user_name = UL_TEXT( "user_name" );
info.version = UL_TEXT( "test" );
conn.Synchronize( &info );
```

Or

```
ul_synch_info info;
conn.InitSynchInfo( &info );
info.user_name = UL_TEXT( "user_name" );
info.version = UL_TEXT( "test" );
conn.SetSynchInfo( &info );
conn.Synchronize();
```

UUIDToStr Function

Synopsis	virtual bool UltraLite_Connection_iface::UUIDToStr (char * <i>dst</i> size_t <i>len</i> p_ul_binary <i>src</i>)
----------	--

Parameters	<ul style="list-style-type: none"> ◆ dst The string being returned. ◆ len The length of the <code>ul_binary</code> array. ◆ src The UUID value to be converted to a string.
Remarks	Converts a UUID to an ANSI string.

UUIDToStr Function

Synopsis	<pre>virtual bool UltraLite_Connection_iface::UUIDToStr(ul_wchar * <i>dst</i> size_t <i>len</i> p_ul_binary <i>src</i>)</pre>
----------	--

Parameters	<ul style="list-style-type: none"> ◆ dst The Unicode string being returned. ◆ len The length of the <code>ul_binary</code> array. ◆ src The UUID value to be converted to a string.
Remarks	Converts a UUID to a Unicode string.

UpgradeSchemaFromFile Function

Synopsis	<pre>virtual bool UltraLite_Connection_iface::UpgradeSchemaFromFile(const ULValue & <i>options</i>)</pre>
----------	--

Parameters	◆ options A semicolon-delimited list of upgrade options.
Remarks	Upgrades to a new schema using an external schema definition file. <code>options</code> is a semicolon-delimited list of upgrade options.

UpgradeSchemaFromFile Function

Synopsis	<pre>virtual bool UltraLite_Connection_iface::UpgradeSchemaFromFile(const ULValue & <i>options</i> ul_schema_upgrade_observer_fn <i>callback</i> ul_void * <i>user_data</i>)</pre>
----------	---

Parameters	<ul style="list-style-type: none"> ◆ options A list of options. ◆ callback The upgrade observer callback function. ◆ user_data User data passed to the callback function.
Remarks	Upgrades to a new schema using an external schema definition file. <code>options</code> is a semi-colon delimited list of upgrade options.

Class UltraLite_Cursor_iface

Synopsis	public UltraLite_Cursor_iface
Derived classes	<ul style="list-style-type: none"> ◆ “Class UltraLite_ResultSet” on page 263 ◆ “Class UltraLite_Table” on page 276
Remarks	Cursor interface.
Members	<p>All members of UltraLite_Cursor_iface, including all inherited members.</p> <ul style="list-style-type: none"> ◆ “AfterLast Function” on page 245 ◆ “BeforeFirst Function” on page 245 ◆ “First Function” on page 245 ◆ “Get Function” on page 246 ◆ “GetRowCount Function” on page 246 ◆ “GetState Function” on page 246 ◆ “GetStreamReader Function” on page 246 ◆ “GetSuspend Function” on page 246 ◆ “IsNull Function” on page 247 ◆ “Last Function” on page 247 ◆ “Next Function” on page 247 ◆ “Previous Function” on page 247 ◆ “Relative Function” on page 247 ◆ “SetSuspend Function” on page 247

AfterLast Function

Synopsis	virtual bool UltraLite_Cursor_iface::AfterLast()
Remarks	Moves the cursor after the last row.

BeforeFirst Function

Synopsis	virtual bool UltraLite_Cursor_iface::BeforeFirst()
Remarks	Moves the cursor before the first row.

First Function

Synopsis	virtual bool UltraLite_Cursor_iface::First()
Remarks	Moves the cursor to the first row.

Get Function

Synopsis virtual ULValue **UltraLite_Cursor_iface::Get**(
 const ULValue & *column_id*
)

Parameters ♦ **column_id** The name or ordinal of the column.

Remarks Fetches a value from a column.

GetRowCount Function

Synopsis virtual ul_u_long **UltraLite_Cursor_iface::GetRowCount()**

Remarks Gets the number of rows in the table.

Calling this method is equivalent to executing “select count(*) from table”

GetState Function

Synopsis virtual UL_RS_STATE **UltraLite_Cursor_iface::GetState()**

Remarks Gets the internal state of the cursor.

See enum UL_RS_STATE in ulglobal.h

GetStreamReader Function

Synopsis virtual UltraLite_StreamReader * **UltraLite_Cursor_iface::GetStreamReader**(
 const ULValue & *id*
)

Parameters ♦ **id** A column identifier, which may be either a 1-based ordinal number
 or a column name.

Remarks Gets a stream reader object for reading string or binary column data in
 chunks.

GetSuspend Function

Synopsis virtual bool **UltraLite_Cursor_iface::GetSuspend()**

Remarks Gets the value of the Suspend property.

Returns true if this cursor is suspended, false otherwise.

IsNull Function

Synopsis	virtual bool UltraLite_Cursor_iface::IsNull (const ULValue & <i>column_id</i>)
Parameters	◆ column_id The name or ordinal of the column.
Remarks	Checks if a column is NULL.

Last Function

Synopsis	virtual bool UltraLite_Cursor_iface::Last ()
Remarks	Moves the cursor to the last row.

Next Function

Synopsis	virtual bool UltraLite_Cursor_iface::Next ()
Remarks	Moves the cursor forward one row. On failure there is no next row, the resulting cursor position is AfterLast Function

Previous Function

Synopsis	virtual bool UltraLite_Cursor_iface::Previous ()
Remarks	Moves the cursor back one row. On failure, the resulting cursor position is BeforeFirst Function

Relative Function

Synopsis	virtual bool UltraLite_Cursor_iface::Relative (ul_fetch_offset <i>offset</i>)
Parameters	◆ offset The number of rows to move.
Remarks	Moves the cursor by <i>offset</i> rows from the current cursor position.

SetSuspend Function

Synopsis	virtual void UltraLite_Cursor_iface::SetSuspend (bool <i>suspend</i>)
Parameters	

-
- ◆ **suspend** Set to true to suspend the connection so that its state can be restored when the database is reopened.

Remarks

Sets the value of the Suspend property.

If true, this cursor is suspended and restored when the database is reopened. Use the persistent name parameter when opening the associated object to identify suspended cursors. If a persistent name parameter was not supplied for this cursor, it cannot be suspended.

Class UltraLite_DatabaseManager

Synopsis	public UltraLite_DatabaseManager
Base classes	◆ “Class UltraLite_DatabaseManager_iface” on page 250
Remarks	DatabaseManager class.
Members	All members of UltraLite_DatabaseManager, including all inherited members. <ul style="list-style-type: none">◆ “CreateAndOpenDatabase Function” on page 250◆ “DropDatabase Function” on page 250◆ “OpenConnection Function” on page 251◆ “Shutdown Function” on page 251

Class UltraLite_DatabaseManager_iface

Synopsis	public UltraLite_DatabaseManager_iface
Derived classes	◆ “ Class UltraLite_DatabaseManager ” on page 249
Remarks	DatabaseManager interface.
Members	All members of UltraLite_DatabaseManager_iface, including all inherited members. ◆ “ CreateAndOpenDatabase Function ” on page 250 ◆ “ DropDatabase Function ” on page 250 ◆ “ OpenConnection Function ” on page 251 ◆ “ Shutdown Function ” on page 251

CreateAndOpenDatabase Function

Synopsis	virtual UltraLite_Connection * UltraLite_DatabaseManager_iface::CreateAndOpenDatabase (ULSqlcaBase & <i>sqlca</i> ULValue const & <i>parms_string</i>)
Parameters	◆ sqlca The initialized sqlca to associate with new connection. ◆ parms_string The creation and connection parameters.
Remarks	Creates a new database and connects to it. The given sqlca is associated with the new connection. If the database already exists, this function fails.
Returns	If the function succeeds, a new connection object is returned. If the function fails, NULL is returned.

DropDatabase Function

Synopsis	virtual bool UltraLite_DatabaseManager_iface::DropDatabase (ULSqlcaBase & <i>sqlca</i> const ULValue & <i>parms_string</i>)
Parameters	◆ sqlca The initialized sqlca. ◆ parms_string The database identification parameters.
Remarks	Erases an existing database. In order to be erased, the database must be stopped.

OpenConnection Function

Synopsis	<pre>virtual UltraLite_Connection * UltraLite_DatabaseManager_ iface::OpenConnection(ULSqlcaBase & <i>sqlca</i> ULValue const & <i>parms_string</i>)</pre>
Parameters	<ul style="list-style-type: none"> ◆ sqlca The initialized sqlca to associate with the new connection. ◆ parms_string The connection string.
Remarks	<p>Opens a new connection to an existing database.</p> <p>The given sqlca is associated with the new connection.</p> <ul style="list-style-type: none"> ◆ SQLE_CONNECTION_ALREADY_EXISTS - You are already connected using these parameters (specifically the sqlca and connection name, CON). ◆ SQLE_INVALID_LOGON - The userid you supplied does not exist or the password is incorrect. User authentication is only enabled if you have called the ULEnableUserAuthentication function before connecting. ◆ SQLE_INVALID_SQL_IDENTIFIER - The userid or password you supplied was not legal, or no userid was supplied at all (use UID=user;PWD=password). ◆ SQLE_TOO_MANY_CONNECTIONS - You have opened too many connections and attempted to exceed the (concurrent) connection limit. <p>To get error information, use the associated Class ULSqlca object. Possible errors include:</p>
Returns	If the function succeeds, a new connection object is returned. If the function fails, NULL is returned.

Shutdown Function

Synopsis	<pre>virtual void UltraLite_DatabaseManager_iface::Shutdown(ULSqlcaBase & <i>sqlca</i>)</pre>
Parameters	<ul style="list-style-type: none"> ◆ sqlca The initialized sqlca.
Remarks	<p>Closes all databases and releases the database manager.</p> <p>Any remaining associated objects are destroyed. After calling this function, the database manager can no longer be used (nor can any other previously obtained objects).</p>

Class UltraLite_DatabaseSchema

Synopsis	public UltraLite_DatabaseSchema
Base classes	<ul style="list-style-type: none">◆ “Class UltraLite_SQLObject_iface” on page 269◆ “Class UltraLite_DatabaseSchema_iface” on page 253
Remarks	DatabaseSchema class.
Members	<p>All members of UltraLite_DatabaseSchema, including all inherited members.</p> <ul style="list-style-type: none">◆ “AddRef Function” on page 269◆ “GetCollationName Function” on page 253◆ “GetConnection Function” on page 269◆ “GetIFace Function” on page 269◆ “GetPublicationCount Function” on page 253◆ “GetPublicationID Function” on page 253◆ “GetPublicationMask Function” on page 254◆ “GetPublicationName Function” on page 254◆ “GetSignature Function” on page 254◆ “GetTableCount Function” on page 254◆ “GetTableName Function” on page 254◆ “GetTableSchema Function” on page 255◆ “IsCaseSensitive Function” on page 255◆ “Release Function” on page 270

Class UltraLite_DatabaseSchema_iface

Synopsis	public UltraLite_DatabaseSchema_iface
Derived classes	◆ “ Class UltraLite_DatabaseSchema ” on page 252
Remarks	DatabaseSchema interface.
Members	All members of UltraLite_DatabaseSchema_iface, including all inherited members. <ul style="list-style-type: none"> ◆ “GetCollationName Function” on page 253 ◆ “GetPublicationCount Function” on page 253 ◆ “GetPublicationID Function” on page 253 ◆ “GetPublicationMask Function” on page 254 ◆ “GetPublicationName Function” on page 254 ◆ “GetSignature Function” on page 254 ◆ “GetTableCount Function” on page 254 ◆ “GetTableName Function” on page 254 ◆ “GetTableSchema Function” on page 255 ◆ “IsCaseSensitive Function” on page 255

GetCollationName Function

Synopsis	virtual ULValue UltraLite_DatabaseSchema_iface::GetCollationName()
Remarks	Gets the name of the current collation sequence.
Returns	A Class ULValue containing a string is returned.

GetPublicationCount Function

Synopsis	virtual ul_publication_count UltraLite_DatabaseSchema_iface::GetPublicationCount()
Remarks	Gets the number of publications in the database. Publication IDs range from 1 to GetPublicationCount Function

GetPublicationID Function

Synopsis	virtual ul_u_short UltraLite_DatabaseSchema_iface::GetPublicationID(const ULValue & pub_id)
Parameters	◆ pub_id A 1-based ordinal number.
Remarks	Gets a 1-based id for the publication given its name.

GetPublicationMask Function

Synopsis virtual ul_publication_mask **UltraLite_DatabaseSchema_iface::GetPublicationMask**(
 const ULValue & *pub_id*
)
Parameters ♦ **pub_id** A 1-based ordinal number.
Remarks Gets the publication mask for a given publication name Publication masks
 are not publication IDs.
 0 is returned if the publication is not found

GetPublicationName Function

Synopsis virtual ULValue **UltraLite_DatabaseSchema_iface::GetPublicationName**(
 const ULValue & *pub_id*
)
Parameters ♦ **pub_id** A 1-based ordinal number.
Remarks Gets the name of a publication given its 1-based index ID.
 Publication masks are not publication IDs.

GetSignature Function

Synopsis virtual ULValue **UltraLite_DatabaseSchema_iface::GetSignature**()
Remarks Gets the database signature.

GetTableCount Function

Synopsis virtual ul_table_num **UltraLite_DatabaseSchema_iface::GetTableCount**()
Remarks Gets the number of tables.

GetTableName Function

Synopsis virtual ULValue **UltraLite_DatabaseSchema_iface::GetTableName**(
 ul_table_num *tableID*
)
Parameters ♦ **tableID** A 1-based ordinal number.
Remarks Gets the name of a table given its 1-based table ID.
 The [Class ULValue](#) object returned is empty if the table does not exist.

GetTableSchema Function

Synopsis	<pre>virtual UltraLite_TableSchema * UltraLite_DatabaseSchema_ iface::GetTableSchema(const ULValue & <i>table_id</i>)</pre>
Parameters	◆ table_id A 1-based ordinal number.
Remarks	Gets a TableSchema object given a 1-based table ID or name UL_NULL is returned if the table does not exist.

IsCaseSensitive Function

Synopsis	<pre>virtual bool UltraLite_DatabaseSchema_iface::IsCaseSensitive()</pre>
Remarks	Gets the case sensitivity of the database.
Returns	true is returned if the database is case sensitive.

Class UltraLite_IndexSchema

Synopsis	public UltraLite_IndexSchema
Base classes	<ul style="list-style-type: none">◆ “Class UltraLite_SQLObject_iface” on page 269◆ “Class UltraLite_IndexSchema_iface” on page 257
Remarks	IndexSchema class.
Members	<p>All members of UltraLite_IndexSchema, including all inherited members.</p> <ul style="list-style-type: none">◆ “AddRef Function” on page 269◆ “GetColumnCount Function” on page 257◆ “GetColumnName Function” on page 257◆ “GetConnection Function” on page 269◆ “GetID Function” on page 258◆ “GetIFace Function” on page 269◆ “GetName Function” on page 258◆ “GetReferencedIndexName Function” on page 258◆ “GetReferencedTableName Function” on page 258◆ “GetTableName Function” on page 258◆ “IsColumnDescending Function” on page 258◆ “IsForeignKey Function” on page 259◆ “IsForeignKeyCheckOnCommit Function” on page 259◆ “IsForeignKeyNullable Function” on page 259◆ “IsPrimaryKey Function” on page 259◆ “IsUniqueIndex Function” on page 259◆ “IsUniqueKey Function” on page 259◆ “Release Function” on page 270

Class UltraLite_IndexSchema_iface

Synopsis	public UltraLite_IndexSchema_iface
Derived classes	◆ “Class UltraLite_IndexSchema ” on page 256
Remarks	IndexSchema interface.
Members	All members of <code>UltraLite_IndexSchema_iface</code> , including all inherited members. <ul style="list-style-type: none"> ◆ “GetColumnCount Function” on page 257 ◆ “GetColumnName Function” on page 257 ◆ “GetID Function” on page 258 ◆ “GetName Function” on page 258 ◆ “GetReferencedIndexName Function” on page 258 ◆ “GetReferencedTableName Function” on page 258 ◆ “GetTableName Function” on page 258 ◆ “IsColumnDescending Function” on page 258 ◆ “IsForeignKey Function” on page 259 ◆ “IsForeignKeyCheckOnCommit Function” on page 259 ◆ “IsForeignKeyNullable Function” on page 259 ◆ “IsPrimaryKey Function” on page 259 ◆ “IsUniqueIndex Function” on page 259 ◆ “IsUniqueKey Function” on page 259

GetColumnCount Function

Synopsis	virtual <code>ul_column_num</code> UltraLite_IndexSchema_iface::GetColumnCount()
Remarks	Gets the number of columns in the index.

GetColumnName Function

Synopsis	virtual <code>ULValue</code> UltraLite_IndexSchema_iface::GetColumnName(ul_column_num <i>col_id_in_index</i>)
Parameters	◆ col_id_in_index The 1-based ordinal number indicating the position of the column in the index.
Remarks	Gets the name of the column given the position of the column in the index. <code>Class ULValue</code> object returned is empty if the column does not exist. <code>SQL_C_COLUMN_NOT_FOUND</code> is returned if the column name does not exist.

GetID Function

Synopsis	virtual ul_index_num UltraLite_IndexSchema_iface::GetID()
Remarks	Gets the index Id.
Returns	The id of this index.

GetName Function

Synopsis	virtual ULValue UltraLite_IndexSchema_iface::GetName()
Remarks	Gets the name of the index.

GetReferencedIndexName Function

Synopsis	virtual ULValue UltraLite_IndexSchema_iface::GetReferencedIndexName()
Remarks	Gets the associated primary index name. This method is for foreign keys only. The Class ULValue object returned is empty if the index is not a foreign key.

GetReferencedTableName Function

Synopsis	virtual ULValue UltraLite_IndexSchema_iface::GetReferencedTableName()
Remarks	Gets the associated primary table name. This method is for foreign keys The Class ULValue object returned is empty if the index is not a foreign key

GetTableName Function

Synopsis	virtual ULValue UltraLite_IndexSchema_iface::GetTableName()
Remarks	Gets the name of the table containing the index.

IsColumnDescending Function

Synopsis	virtual bool UltraLite_IndexSchema_iface::IsColumnDescending(const ULValue & <i>column_name</i>)
Parameters	◆ column_name The column name.
Remarks	Returns true if the column is set to descending order. SQLE_COLUMN_NOT_FOUND is set if the column name does not exist.

IsForeignKey Function

Synopsis virtual bool **UltraLite_IndexSchema_iface::IsForeignKey()**

Remarks Returns true if the index is a foreign key.

IsForeignKeyCheckOnCommit Function

Synopsis virtual bool **UltraLite_IndexSchema_iface::IsForeignKeyCheckOnCommit()**

Remarks Returns true if this foreign key checks referential integrity on commit.
Otherwise RI is checked on insert

IsForeignKeyNullable Function

Synopsis virtual bool **UltraLite_IndexSchema_iface::IsForeignKeyNullable()**

Remarks Returns true if the index is a unique foreign key constraint.

IsPrimaryKey Function

Synopsis virtual bool **UltraLite_IndexSchema_iface::IsPrimaryKey()**

Remarks Returns true if the index is the primary key.

IsUniqueIndex Function

Synopsis virtual bool **UltraLite_IndexSchema_iface::IsUniqueIndex()**

Remarks Returns true if the index is a unique index.

IsUniqueKey Function

Synopsis virtual bool **UltraLite_IndexSchema_iface::IsUniqueKey()**

Remarks Returns true if the index is a primary key or a unique constraint.

Class UltraLite_PreparedStatement

Synopsis	public UltraLite_PreparedStatement
Base classes	<ul style="list-style-type: none">◆ “Class UltraLite_SQLObject_iface” on page 269◆ “Class UltraLite_PreparedStatement_iface” on page 261
Remarks	PreparedStatement class.
Members	<p>All members of UltraLite_PreparedStatement, including all inherited members.</p> <ul style="list-style-type: none">◆ “AddRef Function” on page 269◆ “ExecuteQuery Function” on page 261◆ “ExecuteStatement Function” on page 261◆ “GetConnection Function” on page 269◆ “GetIFace Function” on page 269◆ “GetPlan Function” on page 261◆ “GetSchema Function” on page 262◆ “GetStreamWriter Function” on page 262◆ “HasResultSet Function” on page 262◆ “Release Function” on page 270◆ “SetParameter Function” on page 262◆ “SetParameterNull Function” on page 262

Class UltraLite_PreparedStatement_iface

Synopsis	public UltraLite_PreparedStatement_iface
Derived classes	◆ “Class UltraLite_PreparedStatement ” on page 260
Remarks	PreparedStatement interface.
Members	All members of <code>UltraLite_PreparedStatement_iface</code> , including all inherited members. <ul style="list-style-type: none"> ◆ “ExecuteQuery Function” on page 261 ◆ “ExecuteStatement Function” on page 261 ◆ “GetPlan Function” on page 261 ◆ “GetSchema Function” on page 262 ◆ “GetStreamWriter Function” on page 262 ◆ “HasResultSet Function” on page 262 ◆ “SetParameter Function” on page 262 ◆ “SetParameterNull Function” on page 262

ExecuteQuery Function

Synopsis	virtual <code>UltraLite_ResultSet * UltraLite_PreparedStatement_iface::ExecuteQuery()</code>
Remarks	Executes a SQL query. A <code>ResultSet</code> object is returned.

ExecuteStatement Function

Synopsis	virtual <code>ul_s_long UltraLite_PreparedStatement_iface::ExecuteStatement()</code>
Remarks	Executes a SQL statement.

GetPlan Function

Synopsis	virtual <code>size_t UltraLite_PreparedStatement_iface::GetPlan(</code> <code>ul_char * <i>buffer</i></code> <code>size_t <i>size</i></code> <code>)</code>
Parameters	◆ buffer The buffer to receive the plan description. ◆ size The size, in <code>ul_chars</code> , of the buffer.
Remarks	Gets a text-based description of query execution plan.

GetSchema Function

Synopsis virtual UltraLite_ResultSetSchema * **UltraLite_PreparedStatement_iface::GetSchema()**

Remarks Gets the schema for the result set.

GetStreamWriter Function

Synopsis virtual UltraLite_StreamWriter * **UltraLite_PreparedStatement_iface::GetStreamWriter(**
 ul_column_num *parameter_id*
)

Parameters ♦ **parameter_id** A column identifier, which may be either a 1-based ordinal number or a column name.

Remarks Gets a stream writer for streaming string/binary data into a parameter.

HasResultSet Function

Synopsis virtual bool **UltraLite_PreparedStatement_iface::HasResultSet()**

Remarks Determines if this SQL statement has a result set.

SetParameter Function

Synopsis virtual void **UltraLite_PreparedStatement_iface::SetParameter(**
 ul_column_num *parameter_id*
 ULValue const & *value*
)

Parameters ♦ **parameter_id** The 1-based ordinal of the parameter.

 ♦ **value** The value to set the parameter.

Remarks Sets a parameter for the SQL statement.

SetParameterNull Function

Synopsis virtual void **UltraLite_PreparedStatement_iface::SetParameterNull(**
 ul_column_num *parameter_id*
)

Parameters ♦ **parameter_id** The 1-based ordinal of the parameter.

Remarks Sets a parameter to null.

Class UltraLite_ResultSet

Synopsis	public UltraLite_ResultSet
Base classes	<ul style="list-style-type: none">◆ “Class UltraLite_SQLObject_iface” on page 269◆ “Class UltraLite_ResultSet_iface” on page 264◆ “Class UltraLite_Cursor_iface” on page 245
Remarks	ResultSet class.
Members	All members of UltraLite_ResultSet, including all inherited members. <ul style="list-style-type: none">◆ “AddRef Function” on page 269◆ “AfterLast Function” on page 245◆ “BeforeFirst Function” on page 245◆ “First Function” on page 245◆ “Get Function” on page 246◆ “GetConnection Function” on page 269◆ “GetIFace Function” on page 269◆ “GetRowCount Function” on page 246◆ “GetSchema Function” on page 264◆ “GetState Function” on page 246◆ “GetStreamReader Function” on page 246◆ “GetSuspend Function” on page 246◆ “IsNull Function” on page 247◆ “Last Function” on page 247◆ “Next Function” on page 247◆ “Previous Function” on page 247◆ “Relative Function” on page 247◆ “Release Function” on page 270◆ “SetSuspend Function” on page 247

Class UltraLite_ResultSet_iface

Synopsis	public UltraLite_ResultSet_iface
Derived classes	◆ “Class UltraLite_ResultSet” on page 263
Remarks	ResultSet interface.
Members	All members of UltraLite_ResultSet_iface, including all inherited members. ◆ “GetSchema Function” on page 264

GetSchema Function

Synopsis	virtual UltraLite_ResultSetSchema * UltraLite_ResultSet_iface::GetSchema()
Remarks	Gets the schema for this result set.

Class UltraLite_ResultSetSchema

Synopsis	public UltraLite_ResultSetSchema
Base classes	<ul style="list-style-type: none">◆ “Class UltraLite_SQLObject_iface” on page 269◆ “Class UltraLite_RowSchema_iface” on page 266
Remarks	ResultSetSchema class.
Members	<p>All members of UltraLite_ResultSetSchema, including all inherited members.</p> <ul style="list-style-type: none">◆ “AddRef Function” on page 269◆ “GetColumnCount Function” on page 266◆ “GetColumnID Function” on page 266◆ “GetColumnName Function” on page 266◆ “GetColumnPrecision Function” on page 267◆ “GetColumnScale Function” on page 267◆ “GetColumnSize Function” on page 267◆ “GetColumnSQLType Function” on page 267◆ “GetColumnType Function” on page 268◆ “GetConnection Function” on page 269◆ “GetIFace Function” on page 269◆ “Release Function” on page 270

Class UltraLite_RowSchema_iface

Synopsis	public UltraLite_RowSchema_iface
Derived classes	<ul style="list-style-type: none">◆ “Class UltraLite_ResultSetSchema” on page 265◆ “Class UltraLite_TableSchema” on page 285
Remarks	RowSchema interface.
Members	All members of UltraLite_RowSchema_iface, including all inherited members. <ul style="list-style-type: none">◆ “GetColumnCount Function” on page 266◆ “GetColumnID Function” on page 266◆ “GetColumnName Function” on page 266◆ “GetColumnPrecision Function” on page 267◆ “GetColumnScale Function” on page 267◆ “GetColumnSize Function” on page 267◆ “GetColumnSQLType Function” on page 267◆ “GetColumnType Function” on page 268

GetColumnCount Function

Synopsis	virtual ul_column_num UltraLite_RowSchema_iface::GetColumnCount()
Remarks	Gets the number of columns in the table.

GetColumnID Function

Synopsis	virtual ul_column_num UltraLite_RowSchema_iface::GetColumnID(const ULValue & <i>column_name</i>)
Parameters	◆ column_name The column name.
Remarks	Gets the 1-based column ID 0 is returned if the column does not exist SQLE_COLUMN_NOT_FOUND is set if the column name does not exist.

GetColumnName Function

Synopsis	virtual ULValue UltraLite_RowSchema_iface::GetColumnName(ul_column_num <i>column_id</i>)
Parameters	◆ column_id A 1-based ordinal number.
Remarks	Gets the name of a column given its 1-based ID The Class ULValue object returned is empty if the column does not exist. SQLE_COLUMN_NOT_FOUND is set if the column name does not exist

GetColumnPrecision Function

Synopsis	virtual size_t UltraLite_RowSchema_iface::GetColumnPrecision (const ULValue & <i>column_id</i>)
Parameters	◆ column_id A 1-based ordinal number.
Remarks	Gets the precision of a numeric column Same errors as GetColumnScale.

GetColumnSQLType Function

Synopsis	virtual ul_column_sql_type UltraLite_RowSchema_iface::GetColumnSQLType (const ULValue & <i>column_id</i>)
Parameters	◆ column_id A 1-based ordinal number.
Remarks	Gets the SQL type of a column. See the ul_column_sql_type in ulprotos.h. UL_SQLTYPE_BAD_INDEX is returned if the column does not exist No SQL error is set

GetColumnScale Function

Synopsis	virtual size_t UltraLite_RowSchema_iface::GetColumnScale (const ULValue & <i>column_id</i>)
Parameters	◆ column_id A 1-based ordinal number.
Remarks	Gets the scale of a numeric column. 0 is returned if the column is not a numeric type or if the column does not exist. SQLE_COLUMN_NOT_FOUND is set if the column name does not exist SQLE_DATATYPE_NOT_ALLOWED is set if the column type is not a numeric.

GetColumnSize Function

Synopsis	virtual size_t UltraLite_RowSchema_iface::GetColumnSize (const ULValue & <i>column_id</i>)
Parameters	◆ column_id A 1-based ordinal number.
Remarks	Gets the size of the column. 0 is returned if the column does not exist or if the column type does not have a variable length. SQLE_COLUMN_NOT_FOUND is set if the column

name does not exist `SQL_DATATYPE_NOT_ALLOWED` is set if the column type is not one of the following: `UL_SQLTYPE_CHAR`
`UL_SQLTYPE_BINARY`

GetColumnType Function

Synopsis virtual ul_column_storage_type **UltraLite_RowSchema_iface::GetColumnType**(
 const ULValue & *column_id*
)

Parameters ♦ **column_id** A 1-based ordinal number.

Remarks Gets the type of a column.

See the `ul_column_storage_type` enum in `ulprotos.h`
`UL_TYPE_BAD_INDEX` is returned if the column does not exist No SQL error is set

Class UltraLite_SQLObject_iface

Synopsis	public UltraLite_SQLObject_iface
Derived classes	<ul style="list-style-type: none"> ◆ “Class UltraLite_Connection” on page 234 ◆ “Class UltraLite_DatabaseSchema” on page 252 ◆ “Class UltraLite_IndexSchema” on page 256 ◆ “Class UltraLite_PreparedStatement” on page 260 ◆ “Class UltraLite_ResultSet” on page 263 ◆ “Class UltraLite_ResultSetSchema” on page 265 ◆ “Class UltraLite_StreamReader” on page 271 ◆ “Class UltraLite_StreamWriter” on page 275 ◆ “Class UltraLite_Table” on page 276 ◆ “Class UltraLite_TableSchema” on page 285
Remarks	SQLObject interface.
Members	<p>All members of UltraLite_SQLObject_iface, including all inherited members.</p> <ul style="list-style-type: none"> ◆ “AddRef Function” on page 269 ◆ “GetConnection Function” on page 269 ◆ “GetIFace Function” on page 269 ◆ “Release Function” on page 270

AddRef Function

Synopsis	virtual ul_ret_void UltraLite_SQLObject_iface::AddRef()
Remarks	<p>Increase the internal reference count for an object.</p> <p>Note that you must match each call to AddRef Function with a call to Release Function in order to free the object.</p>

GetConnection Function

Synopsis	virtual UltraLite_Connection * UltraLite_SQLObject_iface::GetConnection()
Remarks	Gets the Connection object.
Returns	The connection associated with this object.

GetIFace Function

Synopsis	virtual ul_void * UltraLite_SQLObject_iface::GetIFace(ul_iface_id <i>iface</i>)
----------	--

Parameters ◆ **iface** Reserved for future use.

Remarks Reserved for future use.

Release Function

Synopsis virtual ul_u_long **UltraLite_SQLObject_iface::Release()**

Remarks Release a reference to an object.

The object will be freed once all references have been removed. You must call [Release Function](#) at least once and if you use [AddRef Function](#) you also need a matching call from each [AddRef Function](#).

Class UltraLite_StreamReader

Synopsis	public UltraLite_StreamReader
Base classes	<ul style="list-style-type: none">◆ “Class UltraLite_SQLObject_iface” on page 269◆ “Class UltraLite_StreamReader_iface” on page 272
Remarks	StreamReader class.
Members	<p>All members of UltraLite_StreamReader, including all inherited members.</p> <ul style="list-style-type: none">◆ “AddRef Function” on page 269◆ “GetByteChunk Function” on page 272◆ “GetConnection Function” on page 269◆ “GetIFace Function” on page 269◆ “GetLength Function” on page 272◆ “GetStringChunk Function” on page 273◆ “GetStringChunk Function” on page 273◆ “Release Function” on page 270◆ “SetReadPosition Function” on page 274

Class UltraLite_StreamReader_iface

Synopsis	public UltraLite_StreamReader_iface
Derived classes	◆ “Class UltraLite_StreamReader” on page 271
Remarks	StreamReader interface. This interface supports reading/retrieving string (varchar) and binary columns.
Members	All members of UltraLite_StreamReader_iface, including all inherited members. ◆ “GetByteChunk Function” on page 272 ◆ “GetLength Function” on page 272 ◆ “GetStringChunk Function” on page 273 ◆ “GetStringChunk Function” on page 273 ◆ “SetReadPosition Function” on page 274

GetByteChunk Function

Synopsis	virtual bool UltraLite_StreamReader_iface::GetByteChunk (ul_byte * <i>data</i> size_t <i>buffer_len</i> size_t * <i>len_retn</i> bool * <i>morebytes</i>)
Parameters	◆ data A buffer of bytes. ◆ buffer_len The length of buffer. ◆ len_retn output: the length returned. ◆ morebytes output: true if there are more bytes to read.
Remarks	Gets a byte chunk from current StreamReader offset. Copy <i>buffer_len</i> bytes in to buffer <i>data</i> . bytes are read from where the last read left off unless SetReadPosition Function is used.

GetLength Function

Synopsis	virtual size_t UltraLite_StreamReader_iface::GetLength (bool <i>fetching_as_wide</i>)
Parameters	◆ fetching_as_wide The indicated fetch string type. Use false for binary.

Remarks Gets the length of a string/binary value.

For binary and for strings fetched as ANSI (`fetch_is_wide = false`) the number of bytes is returned. When fetching UNICODE (`fetch_is_wide = true`) the number of unicode characters is returned.

GetStringChunk Function

Synopsis

```
virtual bool UltraLite_StreamReader_iface::GetStringChunk(
    ul_wchar * str
    size_t buffer_len
    size_t * len_retn
    bool * morebytes
)
```

Parameters

- ◆ **str** A buffer of wide characters.
- ◆ **buffer_len** The length of the buffer.
- ◆ **len_retn** output: the length returned.
- ◆ **morebytes** output: true if there are more characters to read.

Remarks Gets a string chunk from current StreamReader offset.

Copy `buffer_len` wide characters in to buffer `str`. Characters are read from where the last read left off unless [SetReadPosition Function](#) is used.

GetStringChunk Function

Synopsis

```
virtual bool UltraLite_StreamReader_iface::GetStringChunk(
    char * str
    size_t buffer_len
    size_t * len_retn
    bool * morebytes
)
```

Parameters

- ◆ **str** A buffer of characters.
- ◆ **buffer_len** The length of the buffer.
- ◆ **len_retn** output: the length returned.
- ◆ **morebytes** output: true if there are more characters to read.

Remarks Gets a string chunk from current StreamReader offset.

Copy `buffer_len` bytes in to buffer `str`. Characters are read from where the last read left off unless [SetReadPosition Function](#) is used.

SetReadPosition Function

Synopsis	virtual bool UltraLite_StreamReader_iface::SetReadPosition (size_t <i>offset</i>)
Parameters	◆ offset The offset. For strings, the offset is in characters.
Remarks	Sets the offset in the data for the next read.

Class UltraLite_StreamWriter

Synopsis	public UltraLite_StreamWriter
Base classes	◆ “Class UltraLite_SQLObject_iface” on page 269
Remarks	StreamWriter class.
Members	All members of UltraLite_StreamWriter, including all inherited members. <ul style="list-style-type: none">◆ “AddRef Function” on page 269◆ “GetConnection Function” on page 269◆ “GetIFace Function” on page 269◆ “Release Function” on page 270

Class UltraLite_Table

Synopsis	public UltraLite_Table
Base classes	<ul style="list-style-type: none">◆ “Class UltraLite_SQLObject_iface” on page 269◆ “Class UltraLite_Table_iface” on page 278◆ “Class UltraLite_Cursor_iface” on page 245
Remarks	Table class.
Members	All members of UltraLite_Table, including all inherited members. <ul style="list-style-type: none">◆ “AddRef Function” on page 269◆ “AfterLast Function” on page 245◆ “BeforeFirst Function” on page 245◆ “Delete Function” on page 278◆ “DeleteAllRows Function” on page 278◆ “Find Function” on page 279◆ “FindBegin Function” on page 279◆ “FindFirst Function” on page 279◆ “FindLast Function” on page 280◆ “FindNext Function” on page 280◆ “FindPrevious Function” on page 280◆ “First Function” on page 245◆ “Get Function” on page 246◆ “GetConnection Function” on page 269◆ “GetIFace Function” on page 269◆ “GetRowCount Function” on page 246◆ “GetSchema Function” on page 281◆ “GetState Function” on page 246◆ “GetStreamReader Function” on page 246◆ “GetStreamWriter Function” on page 281◆ “GetSuspend Function” on page 246◆ “Insert Function” on page 281◆ “InsertBegin Function” on page 281◆ “IsNull Function” on page 247◆ “Last Function” on page 247◆ “Lookup Function” on page 281◆ “LookupBackward Function” on page 282◆ “LookupBegin Function” on page 282◆ “LookupForward Function” on page 282◆ “Next Function” on page 247◆ “Previous Function” on page 247◆ “Relative Function” on page 247◆ “Release Function” on page 270

- ◆ “Set Function” on page 283
- ◆ “SetDefault Function” on page 283
- ◆ “SetNull Function” on page 283
- ◆ “SetSuspend Function” on page 247
- ◆ “TruncateTable Function” on page 283
- ◆ “Update Function” on page 283
- ◆ “UpdateBegin Function” on page 284

Class UltraLite_Table_iface

Synopsis	public UltraLite_Table_iface
Derived classes	◆ “Class UltraLite_Table” on page 276
Remarks	Table interface.
Members	All members of UltraLite_Table_iface, including all inherited members. <ul style="list-style-type: none">◆ “Delete Function” on page 278◆ “DeleteAllRows Function” on page 278◆ “Find Function” on page 279◆ “FindBegin Function” on page 279◆ “FindFirst Function” on page 279◆ “FindLast Function” on page 280◆ “FindNext Function” on page 280◆ “FindPrevious Function” on page 280◆ “GetSchema Function” on page 281◆ “GetStreamWriter Function” on page 281◆ “Insert Function” on page 281◆ “InsertBegin Function” on page 281◆ “Lookup Function” on page 281◆ “LookupBackward Function” on page 282◆ “LookupBegin Function” on page 282◆ “LookupForward Function” on page 282◆ “Set Function” on page 283◆ “SetDefault Function” on page 283◆ “SetNull Function” on page 283◆ “TruncateTable Function” on page 283◆ “Update Function” on page 283◆ “UpdateBegin Function” on page 284

Delete Function

Synopsis	virtual bool UltraLite_Table_iface::Delete()
Remarks	Deletes the current row. The cursor position is left on the next valid row.

DeleteAllRows Function

Synopsis	virtual bool UltraLite_Table_iface::DeleteAllRows()
Remarks	Deletes all rows from table.

If the stop sync property is set on the connection, the deleted rows are not synchronized.

Note: Uncommitted inserts from other connections are not deleted, any uncommitted deletes from other connections will not be deleted if the other connection does a rollback after [DeleteAllRows Function](#) has been called.

Returns true on success, false otherwise (table not open, SQL error, etc.).

Find Function

Synopsis `virtual bool UltraLite_Table_iface::Find(
ul_column_num ncols
)`

Parameters ♦ **ncols** For composite indexes, the number of columns to use in the lookup.

Remarks Equivalent to FindFirst.

To specify the value to search for, Set the column value for each column in the index

The cursor is left positioned on the first row that exactly matches the index value

If no row matches the index value, the cursor position is AfterLast() and the function returns false

FindBegin Function

Synopsis `virtual bool UltraLite_Table_iface::FindBegin()`

Remarks Selects find mode for setting columns.

Only columns in the index that the table was opened with may be set.

FindFirst Function

Synopsis `virtual bool UltraLite_Table_iface::FindFirst(
ul_column_num ncols
)`

Parameters ♦ **ncols** For composite indexes, the number of columns to use in the lookup.

Remarks Does an exact match lookup based on the current index scanning forward through the table.

♦ To specify the value to search for, Set the column value for each column in the index

-
- ◆ The cursor is left positioned on the first row that exactly matches the index value
 - ◆ If no row matches the index value, the cursor position is AfterLast() and the function returns false

FindLast Function

Synopsis

```
virtual bool UltraLite_Table_iface::FindLast(  
    ul_column_num ncols  
)
```

Parameters

- ◆ **ncols** For composite indexes, the number of columns to use in the lookup.

Remarks

Does an exact match lookup based on the current index scanning backward through the table.

- ◆ To specify the value to search for, Set the column value for each column in the index
- ◆ The cursor is left positioned on the first row that exactly matches the index value
- ◆ If no row matches the index value, the cursor position is BeforeFirst() and the function returns false

FindNext Function

Synopsis

```
virtual bool UltraLite_Table_iface::FindNext(  
    ul_column_num ncols  
)
```

Parameters

- ◆ **ncols** For composite indexes, the number of columns to use in the lookup.

Remarks

Gets the next row (forward) that exactly matches the index.

Returns false if no more rows match the index. In this case the cursor is positioned after the last row.

FindPrevious Function

Synopsis

```
virtual bool UltraLite_Table_iface::FindPrevious(  
    ul_column_num ncols  
)
```

Parameters

- ◆ **ncols** For composite indexes, the number of columns to use in the lookup.

Remarks Gets the previous row (backward) that exactly matches the index.
Returns false if no more rows match the index. In this case the cursor is positioned before the first row.

GetSchema Function

Synopsis `virtual UltraLite_TableSchema * UltraLite_Table_iface::GetSchema()`

Remarks Gets a schema object for this table.

GetStreamWriter Function

Synopsis `virtual UltraLite_StreamWriter * UltraLite_Table_iface::GetStreamWriter(const ULValue & column_id)`

Parameters ♦ **column_id** A column identifier, which may be either a 1-based ordinal number or a column name.

Remarks Gets a stream writer for streaming string/binary data into a column.

Insert Function

Synopsis `virtual bool UltraLite_Table_iface::Insert()`

Remarks Inserts a new row into the table.

The table must be in insert mode for this operation to succeed. Use [InsertBegin Function](#) to switch to insert mode.

InsertBegin Function

Synopsis `virtual bool UltraLite_Table_iface::InsertBegin()`

Remarks Selects insert mode for setting columns.

All columns may be modified in this mode.

Lookup Function

Synopsis `virtual bool UltraLite_Table_iface::Lookup(ul_column_num ncols)`

Parameters ♦ **ncols** For composite indexes, the number of columns to use in the lookup.

Remarks Equivalent to `LookupForward`.

To specify the value to search for, Set the column value for each column in the index

The cursor is left positioned on the first row that matches or is greater than the index value

If the resulting cursor position is AfterLast() the return value is false

LookupBackward Function

Synopsis	virtual bool UltraLite_Table_iface::LookupBackward (ul_column_num <i>ncols</i>)
Parameters	◆ ncols For composite indexes, the number of columns to use in the lookup.
Remarks	Does a lookup based on the current index scanning backward through the table. <ul style="list-style-type: none">◆ To specify the value to search for, Set the column value for each column in the index◆ The cursor is left positioned on the last row that matches or is less than the index value◆ If resulting cursor position is BeforeFirst() the return value is false◆ For composite indexes, ncols specifies the number of columns to use in the lookup

LookupBegin Function

Synopsis	virtual bool UltraLite_Table_iface::LookupBegin ()
Remarks	Selects lookup mode for setting columns. Only columns in the index that the table was opened with may be set.

LookupForward Function

Synopsis	virtual bool UltraLite_Table_iface::LookupForward (ul_column_num <i>ncols</i>)
Parameters	◆ ncols For composite indexes, the number of columns to use in the lookup.
Remarks	Does a lookup based on the current index scanning forward through the table.

- ◆ To specify the value to search for, Set the column value for each column in the index
- ◆ The cursor is left positioned on the first row that matches or is greater than the index value
- ◆ If the resulting cursor position is AfterLast() the return value is false

Set Function

Synopsis	virtual bool UltraLite_Table_iface::Set (const ULValue & <i>column_id</i> const ULValue & <i>value</i>)
Parameters	◆ column_id A 1-based ordinal number identifying the column. ◆ value The value to which the column is set.
Remarks	Sets a column value.

SetDefault Function

Synopsis	virtual bool UltraLite_Table_iface::SetDefault (const ULValue & <i>column_id</i>)
Parameters	◆ column_id A 1-based ordinal number identifying the column.
Remarks	Sets column(s) to their default value.

SetNull Function

Synopsis	virtual bool UltraLite_Table_iface::SetNull (const ULValue & <i>column_id</i>)
Parameters	◆ column_id A 1-based ordinal number identifying the column.
Remarks	Sets a column to null.

TruncateTable Function

Synopsis	virtual bool UltraLite_Table_iface::TruncateTable ()
Remarks	Truncates the table. Temporarily activates stop synchronization delete.

Update Function

Synopsis	virtual bool UltraLite_Table_iface::Update ()
----------	--

Remarks Updates the current row.
The table must be in update mode for this operation to succeed. Use [UpdateBegin Function](#) to switch to update mode;

UpdateBegin Function

Synopsis virtual bool **UltraLite_Table_iface::UpdateBegin()**

Remarks Selects update mode for setting columns.
Columns in the primary key may not be modified when in update mode.

Class UltraLite_TableSchema

Synopsis	public UltraLite_TableSchema
Base classes	<ul style="list-style-type: none"> ◆ “Class UltraLite_SQLObject_iface” on page 269 ◆ “Class UltraLite_TableSchema_iface” on page 286 ◆ “Class UltraLite_RowSchema_iface” on page 266
Remarks	TableSchema class.
Members	<p>All members of UltraLite_TableSchema, including all inherited members.</p> <ul style="list-style-type: none"> ◆ “AddRef Function” on page 269 ◆ “GetColumnCount Function” on page 266 ◆ “GetColumnDefault Function” on page 286 ◆ “GetColumnID Function” on page 266 ◆ “GetColumnName Function” on page 266 ◆ “GetColumnPrecision Function” on page 267 ◆ “GetColumnScale Function” on page 267 ◆ “GetColumnSize Function” on page 267 ◆ “GetColumnSQLType Function” on page 267 ◆ “GetColumnType Function” on page 268 ◆ “GetConnection Function” on page 269 ◆ “GetGlobalAutoincPartitionSize Function” on page 287 ◆ “GetID Function” on page 287 ◆ “GetIFace Function” on page 269 ◆ “GetIndexCount Function” on page 287 ◆ “GetIndexName Function” on page 287 ◆ “GetIndexSchema Function” on page 287 ◆ “GetName Function” on page 288 ◆ “GetOptimalIndex Function” on page 288 ◆ “GetPrimaryKey Function” on page 288 ◆ “GetUploadUnchangedRows Function” on page 288 ◆ “InPublication Function” on page 288 ◆ “IsColumnAutoinc Function” on page 289 ◆ “IsColumnCurrentDate Function” on page 289 ◆ “IsColumnCurrentTime Function” on page 289 ◆ “IsColumnCurrentTimestamp Function” on page 289 ◆ “IsColumnGlobalAutoinc Function” on page 289 ◆ “IsColumnInIndex Function” on page 290 ◆ “IsColumnNewUUID Function” on page 290 ◆ “IsColumnNullable Function” on page 290 ◆ “IsNeverSynchronized Function” on page 290 ◆ “Release Function” on page 270

Class UltraLite_TableSchema_iface

Synopsis	public UltraLite_TableSchema_iface
Derived classes	◆ “Class UltraLite_TableSchema ” on page 285
Remarks	TableSchema interface.
Members	All members of UltraLite_TableSchema_iface , including all inherited members. <ul style="list-style-type: none">◆ “GetColumnDefault Function” on page 286◆ “GetGlobalAutoincPartitionSize Function” on page 287◆ “GetID Function” on page 287◆ “GetIndexCount Function” on page 287◆ “GetIndexName Function” on page 287◆ “GetIndexSchema Function” on page 287◆ “GetName Function” on page 288◆ “GetOptimalIndex Function” on page 288◆ “GetPrimaryKey Function” on page 288◆ “GetUploadUnchangedRows Function” on page 288◆ “InPublication Function” on page 288◆ “IsColumnAutoinc Function” on page 289◆ “IsColumnCurrentDate Function” on page 289◆ “IsColumnCurrentTime Function” on page 289◆ “IsColumnCurrentTimestamp Function” on page 289◆ “IsColumnGlobalAutoinc Function” on page 289◆ “IsColumnInIndex Function” on page 290◆ “IsColumnNewUUID Function” on page 290◆ “IsColumnNullable Function” on page 290◆ “IsNeverSynchronized Function” on page 290

GetColumnDefault Function

Synopsis	virtual ULValue UltraLite_TableSchema_iface::GetColumnDefault (const ULValue & <i>column_id</i>)
Parameters	◆ column_id A 1-based ordinal number.
Remarks	Gets the default value for the column if it exists The Class ULValue object returned has the default contained as a string. The Class ULValue object returned is empty if the column does not contain a default value. <code>SQLC_COLUMN_NOT_FOUND</code> is set if the column name does not exist.

GetGlobalAutoincPartitionSize Function

Synopsis	virtual bool UltraLite_TableSchema_iface::GetGlobalAutoincPartitionSize (const ULValue & <i>column_id</i> ul_u_big * <i>size</i>)
Parameters	<ul style="list-style-type: none"> ◆ column_id A 1-based ordinal number. ◆ size output: The partition size for the column.
Remarks	Returns the partition size for a global autoincrement column.

GetID Function

Synopsis	virtual ul_table_num UltraLite_TableSchema_iface::GetID ()
Remarks	Gets the table ID.

GetIndexCount Function

Synopsis	virtual ul_index_num UltraLite_TableSchema_iface::GetIndexCount ()
Remarks	Returns the number of indexes in the table.

GetIndexName Function

Synopsis	virtual ULValue UltraLite_TableSchema_iface::GetIndexName (ul_index_num <i>index_id</i>)
Parameters	◆ index_id A 1-based ordinal number.
Remarks	Gets the index name given its 1-based ID. The Class ULValue object returned is empty if the index does not exist.

GetIndexSchema Function

Synopsis	virtual UltraLite_IndexSchema * UltraLite_TableSchema_iface::GetIndexSchema (const ULValue & <i>index_id</i>)
Parameters	◆ index_id The name or ID number identifying the index.
Remarks	Gets an IndexSchema object with the given name or id. UL_NULL is returned if the index does not exist

GetName Function

Synopsis virtual ULValue **UltraLite_TableSchema_iface::GetName()**

Remarks Gets the name of the table.

GetOptimalIndex Function

Synopsis virtual ULValue **UltraLite_TableSchema_iface::GetOptimalIndex(const ULValue & *column_id*)**

Parameters ♦ **column_id** A 1-based ordinal number.

Remarks Determine the best index to use for searching for a column value.
Returns the name of the index.

GetPrimaryKey Function

Synopsis virtual UltraLite_IndexSchema * **UltraLite_TableSchema_iface::GetPrimaryKey()**

Remarks Gets the primary key for the table.

GetUploadUnchangedRows Function

Synopsis virtual bool **UltraLite_TableSchema_iface::GetUploadUnchangedRows()**

Remarks Returns true if the table upload every row.
Rows will be uploaded even if they haven't been modified since the last download

InPublication Function

Synopsis virtual bool **UltraLite_TableSchema_iface::InPublication(const ULValue & *publication_name*)**

Parameters ♦ **publication_name** The name of the publication.

Remarks Returns true if the table is contained in the publication
SQLE_PUBLICATION_NOT_FOUND is set if the publication does not exist.

IsColumnAutoinc Function

Synopsis	virtual bool UltraLite_TableSchema_iface::IsColumnAutoinc (const ULValue & <i>column_id</i>)
Parameters	◆ column_id A 1-based ordinal number.
Remarks	Returns true if the column default is set to be auto increment. SQLE_COLUMN_NOT_FOUND is set if the column name does not exist

IsColumnCurrentDate Function

Synopsis	virtual bool UltraLite_TableSchema_iface::IsColumnCurrentDate (const ULValue & <i>column_id</i>)
Parameters	◆ column_id A 1-based ordinal number.
Remarks	Returns true if the column has a current date default.

IsColumnCurrentTime Function

Synopsis	virtual bool UltraLite_TableSchema_iface::IsColumnCurrentTime (const ULValue & <i>column_id</i>)
Parameters	◆ column_id A 1-based ordinal number.
Remarks	Returns true if the column has a current time default.

IsColumnCurrentTimestamp Function

Synopsis	virtual bool UltraLite_TableSchema_iface::IsColumnCurrentTimestamp (const ULValue & <i>column_id</i>)
Parameters	◆ column_id A 1-based ordinal number.
Remarks	Returns true if the column has a current timestamp default.

IsColumnGlobalAutoinc Function

Synopsis	virtual bool UltraLite_TableSchema_iface::IsColumnGlobalAutoinc (const ULValue & <i>column_id</i>)
Parameters	◆ column_id A 1-based ordinal number.

Remarks Returns true if the column default is set to be auto increment
SQLE_COLUMN_NOT_FOUND is set if the column name does not exist.

IsColumnInIndex Function

Synopsis virtual bool **UltraLite_TableSchema_iface::IsColumnInIndex**(
const ULValue & *column_id*
const ULValue & *index_id*
)

Parameters ♦ **column_id** A 1-based ordinal number identifying the column. You can get the *column_id* by calling [GetColumnCount Function](#).
♦ **index_id** A 1-based ordinal number identifying the index. You can get the number of indexes in a table by calling [GetIndexCount Function](#).

Remarks Returns true if the column is contained in the index - id must be 1-based.
SQLE_COLUMN_NOT_FOUND is set if the column name does not exist.
SQLE_INDEX_NOT_FOUND is set if the index does not exist.

IsColumnNewUUID Function

Synopsis virtual bool **UltraLite_TableSchema_iface::IsColumnNewUUID**(
const ULValue & *column_id*
)

Parameters ♦ **column_id** A 1-based ordinal number.

Remarks Returns true if the column has a UUID default.

IsColumnNullable Function

Synopsis virtual bool **UltraLite_TableSchema_iface::IsColumnNullable**(
const ULValue & *column_id*
)

Parameters ♦ **column_id** A 1-based ordinal number.

Remarks Returns true if the column is nullable SQLE_COLUMN_NOT_FOUND is set if the column name does not exist.

IsNeverSynchronized Function

Synopsis virtual bool **UltraLite_TableSchema_iface::IsNeverSynchronized**()

Remarks Returns true if the table is omitted from synchronization.

Class ULValue

Synopsis	public ULValue
Remarks	<p>ULValue object.</p> <p>The ULValue class is a wrapper for the data types stored in an UltraLite cursor. This allows you to store data without having to worry about the data type.</p> <p>ULValue contains many constructors and cast operators so that ULValue can be used seamlessly in most cases without explicitly instantiating a ULValue.</p>
Members	<p>All members of ULValue, including all inherited members.</p> <ul style="list-style-type: none"> ◆ “GetBinary Function” on page 292 ◆ “GetBinary Function” on page 292 ◆ “GetBinaryLength Function” on page 293 ◆ “GetString Function” on page 293 ◆ “GetString Function” on page 293 ◆ “GetStringLength Function” on page 294 ◆ “InDatabase Function” on page 294 ◆ “IsNull Function” on page 294 ◆ “operator bool Function” on page 300 ◆ “operator DECL_DATETIME Function” on page 300 ◆ “operator double Function” on page 300 ◆ “operator float Function” on page 300 ◆ “operator int Function” on page 300 ◆ “operator long Function” on page 300 ◆ “operator short Function” on page 300 ◆ “operator ul_s_big Function” on page 301 ◆ “operator ul_u_big Function” on page 301 ◆ “operator unsigned char Function” on page 301 ◆ “operator unsigned int Function” on page 301 ◆ “operator unsigned long Function” on page 301 ◆ “operator unsigned short Function” on page 301 ◆ “operator= Function” on page 301 ◆ “SetBinary Function” on page 294 ◆ “SetString Function” on page 295 ◆ “SetString Function” on page 295 ◆ “StringCompare Function” on page 295 ◆ “ULValue Function” on page 296 ◆ “ULValue Function” on page 296 ◆ “ULValue Function” on page 296 ◆ “ULValue Function” on page 296 ◆ “ULValue Function” on page 296

- ◆ “ULValue Function” on page 297
- ◆ “ULValue Function” on page 297
- ◆ “ULValue Function” on page 297
- ◆ “ULValue Function” on page 297
- ◆ “ULValue Function” on page 297
- ◆ “ULValue Function” on page 298
- ◆ “ULValue Function” on page 298
- ◆ “ULValue Function” on page 298
- ◆ “ULValue Function” on page 298
- ◆ “ULValue Function” on page 298
- ◆ “ULValue Function” on page 299
- ◆ “ULValue Function” on page 299
- ◆ “ULValue Function” on page 299
- ◆ “ULValue Function” on page 299
- ◆ “ULValue Function” on page 299
- ◆ “ULValue Function” on page 302

GetBinary Function

Synopsis

```
void ULValue::GetBinary(
    p_ul_binary bin
    size_t len
)
```

Parameters

- ◆ **bin** The binary structure to receive bytes.
- ◆ **len** The length of the buffer.

Remarks

Gets a binary value.

Retrieve the current value into a binary buffer, casting as required. If the buffer is too small then the value is truncated. Up to `len` characters are copied to the given buffer.

GetBinary Function

Synopsis

```
void ULValue::GetBinary(
    ul_byte * dst
    size_t len
    size_t * retr_len
)
```

Parameters

- ◆ **dst** The buffer to receive bytes.
- ◆ **len** The length of the buffer.
- ◆ **retr_len** output: The actual number of bytes returned.

Remarks Gets a binary value.

Retrieve the current value into a binary buffer, casting as required. If the buffer is too small then the value is truncated. Up to `len` bytes are copied to the given buffer. The number of bytes actually copied is returned in `retr_len`.

GetBinaryLength Function

Synopsis `size_t ULValue::GetBinaryLength()`

Remarks Gets the length of a Binary value.

Returns Number of bytes necessary to hold the binary value returned by [GetBinary Function](#).

GetString Function

Synopsis `void ULValue::GetString(char * dst, size_t len)`

Parameters

- ◆ **dst** The buffer to receive string value.
- ◆ **len** The length, in bytes, of `dst`.

Remarks Gets the String value.

Retrieve the current value into a string buffer, casting as required. The output string is always null-terminated, if the buffer is too small then the value is truncated. Up to `len` characters are copied to the given buffer, including the null terminator.

GetString Function

Synopsis `void ULValue::GetString(ul_wchar * dst, size_t len)`

Parameters

- ◆ **dst** The buffer to receive string value.
- ◆ **len** The length, in wide chars, of `dst`.

Remarks Gets a String value.

Retrieve the current value into a string buffer, casting as required. The output string is always null-terminated, if the buffer is too small then the

value is truncated. Up to `len` characters are be copied to the given buffer, including the null terminator.

GetStringLength Function

Synopsis `size_t ULValue::GetStringLength(
 bool fetching_as_wide
)`

Parameters ♦ **fetching_as_wide** The indicated fetch string type.

Remarks Gets the length of a String.
Intended usage is as follows.

```
len = v.GetStringLength();  
dst = new char_type[ len ];  
( dst, len ); GetString
```

Given this usage, the length may be different for different character types because of multi-element encodings. The `fetching_as_wide` parameter is used to indicate what character type is used with the [GetString Function](#) call.

Returns Number of bytes or wide chars required to hold the string returned by one of the [GetString Function](#) methods, including the null-terminator. This is not necessarily the number of characters in the string.

InDatabase Function

Synopsis `bool ULValue::InDatabase()`

Remarks Check if value is in the Database.

Returns Returns true if this object is referencing a cursor field

IsNull Function

Synopsis `bool ULValue::IsNull()`

Remarks Check if NULL.

Returns Returns true if this object is an empty ULValue object, or if it references a cursor field that is set to NULL.

SetBinary Function

Synopsis `void ULValue::SetBinary(
 ul_byte * src
 size_t len
)`

Parameters	<ul style="list-style-type: none"> ◆ src A buffer of bytes. ◆ len The length of the buffer.
Remarks	<p>Sets a Binary value.</p> <p>Set the value to reference the binary buffer provided. No bytes are copied from the provided buffer until the value is used.</p>

SetString Function

Synopsis	<pre>void ULValue::SetString(const char * val size_t len)</pre>
Parameters	<ul style="list-style-type: none"> ◆ val A pointer to the null-terminated string representation of this ULValue. ◆ len The length of the string.
Remarks	Cast a ULValue to a string.

SetString Function

Synopsis	<pre>void ULValue::SetString(const ul_wchar * val size_t len)</pre>
Parameters	<ul style="list-style-type: none"> ◆ val A pointer to the null-terminated unicode string representation of this ULValue. ◆ len The length of the string.
Remarks	Cast a ULValue to a unicode string.

StringCompare Function

Synopsis	<pre>ul_compare ULValue::StringCompare(const ULValue & value)</pre>
Parameters	<ul style="list-style-type: none"> ◆ value The string to compare with.
Remarks	<p>Compare strings.</p> <p>Compares strings, or string representations of ULValue objects</p>
Returns	<ul style="list-style-type: none"> ◆ Returns 0 if the strings are equal.

-
- ◆ Returns -1 if the current value compares less than `value` .
 - ◆ Returns 1 if the current value compares greater than `value` .
 - ◆ On error returns -3 if the sqlca of either ULValue object is not set or -2 if the string representation of either ULValue object is UL_NULL

ULValue Function

Synopsis **ULValue::ULValue()**

Remarks Construct a ULValue.

ULValue Function

Synopsis **ULValue::ULValue(**
const ULValue & *vSrc*
)

Parameters ◆ **vSrc** A value to be treated as a ULValue.

Remarks Construct a ULValue from a const.

ULValue Function

Synopsis **ULValue::ULValue(**
bool val
)

Parameters ◆ **val** A boolean value to be treated as a ULValue.

Remarks Construct a ULValue from a bool.

ULValue Function

Synopsis **ULValue::ULValue(**
short val
)

Parameters ◆ **val** A short value to be treated as a ULValue.

Remarks Construct a ULValue from a short.

ULValue Function

Synopsis **ULValue::ULValue(**
long val
)

Parameters ◆ **val** A long value to be treated as a ULValue.

Remarks Construct a ULValue from a long.

ULValue Function

Synopsis **ULValue::ULValue**(
int *val*
)

Parameters ♦ **val** An int value to be treated as a ULValue.

Remarks Construct a ULValue from an int.

ULValue Function

Synopsis **ULValue::ULValue**(
unsigned int *val*
)

Parameters ♦ **val** An unsigned int value to be treated as a ULValue.

Remarks Construct a ULValue from an unsigned int.

ULValue Function

Synopsis **ULValue::ULValue**(
float *val*
)

Parameters ♦ **val** A float value to be treated as a ULValue.

Remarks Construct a ULValue from a float.

ULValue Function

Synopsis **ULValue::ULValue**(
double *val*
)

Parameters ♦ **val** A double value to be treated as a ULValue.

Remarks Construct a ULValue from a double.

ULValue Function

Synopsis **ULValue::ULValue**(
unsigned char *val*
)

Parameters ♦ **val** An unsigned char to be treated as a ULValue.

Remarks Construct a ULValue from an unsigned char.

ULValue Function

Synopsis	ULValue::ULValue (unsigned short <i>val</i>)
Parameters	◆ val An unsigned short to be treated as a ULValue.
Remarks	Construct a ULValue from an unsigned short.

ULValue Function

Synopsis	ULValue::ULValue (unsigned long <i>val</i>)
Parameters	◆ val An unsigned long to be treated as a ULValue.
Remarks	Construct a ULValue from an unsigned long.

ULValue Function

Synopsis	ULValue::ULValue (const ul_u_big & <i>val</i>)
Parameters	◆ val A ul_u_big value to be treated as a ULValue.
Remarks	Construct a ULValue from a ul_u_big.

ULValue Function

Synopsis	ULValue::ULValue (const ul_s_big & <i>val</i>)
Parameters	◆ val A ul_s_big value to be treated as a ULValue.
Remarks	Construct a ULValue from a ul_s_big.

ULValue Function

Synopsis	ULValue::ULValue (const p_ul_binary <i>val</i>)
Parameters	◆ val A ul_binary value to be treated as a ULValue.
Remarks	Construct a ULValue from a ul_binary.

ULValue Function

Synopsis **ULValue::ULValue(**
DECL_DATETIME & *val*
)

Parameters ♦ **val** A datetime value to be treated as a ULValue.

Remarks Construct a ULValue from a datetime.

ULValue Function

Synopsis **ULValue::ULValue(**
const char * *val*
)

Parameters ♦ **val** A pointer to a string to be treated as a ULValue.

Remarks Construct a ULValue from a string.

ULValue Function

Synopsis **ULValue::ULValue(**
const ul_wchar * *val*
)

Parameters ♦ **val** A pointer to a unicode string to be treated as a ULValue.

Remarks Construct a ULValue from a unicode string.

ULValue Function

Synopsis **ULValue::ULValue(**
const char * *val*
size_t *len*
)

Parameters ♦ **val** A buffer holding the string to be treated as a ULValue.

♦ **len** The length of the buffer.

Remarks Construct a ULValue from a buffer of characters.

ULValue Function

Synopsis **ULValue::ULValue(**
const ul_wchar * *val*
size_t *len*
)

Parameters ♦ **val** A buffer holding the string to be treated as a ULValue.

◆ **len** The length of the buffer.

Remarks Construct a ULValue from a buffer of unicode characters.

operator DECL_DATETIME Function

Synopsis **ULValue::operator DECL_DATETIME()**

Remarks Cast a ULValue to a datetime.

operator bool Function

Synopsis **ULValue::operator bool()**

Remarks Cast a ULValue to a boolean.

operator double Function

Synopsis **ULValue::operator double()**

Remarks Cast a ULValue to a double.

operator float Function

Synopsis **ULValue::operator float()**

Remarks Cast a ULValue to a float.

operator int Function

Synopsis **ULValue::operator int()**

Remarks Cast a ULValue to an int.

operator long Function

Synopsis **ULValue::operator long()**

Remarks Cast a ULValue to a long.

operator short Function

Synopsis **ULValue::operator short()**

Remarks Cast a ULValue to a short.

operator ul_s_big Function

Synopsis **ULValue::operator ul_s_big()**

Remarks Cast a ULValue to a signed big int.

operator ul_u_big Function

Synopsis **ULValue::operator ul_u_big()**

Remarks Cast a ULValue to an unsigned big int.

operator unsigned char Function

Synopsis **ULValue::operator unsigned char()**

Remarks Cast a ULValue to a char.

operator unsigned int Function

Synopsis **ULValue::operator unsigned int()**

Remarks Cast a ULValue to an unsigned int.

operator unsigned long Function

Synopsis **ULValue::operator unsigned long()**

Remarks Cast a ULValue to an unsigned long.

operator unsigned short Function

Synopsis **ULValue::operator unsigned short()**

Remarks Cast a ULValue to an unsigned short.

operator= Function

Synopsis **ULValue & ULValue::operator=(
const ULValue & *other*
)**

Parameters ♦ **other** The value to be assigned to a ULValue.

Remarks Override the = operator for ULValues.

~ULValue Function

Synopsis	ULValue::~~ULValue()
Remarks	Destructor for ULValue.

CHAPTER 14

UltraLite Static C++ API Reference

About this chapter

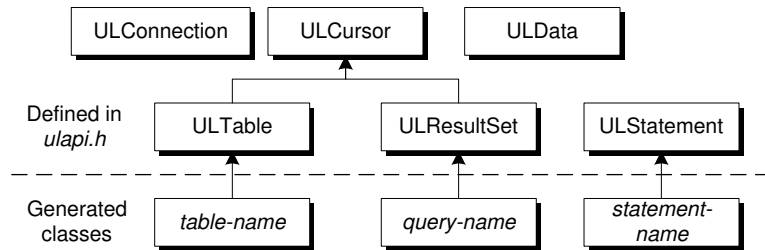
This chapter describes the UltraLite Static C++ API.

Contents

Topic:	page
C++ API class hierarchy	304
C++ API language elements	305
ULConnection class	306
ULCursor class	319
ULData class	331
ULResultSet class	340
ULTable class	341
Generated result set class	347
Generated statement class	350
Generated table class	352

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 Static 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 208.](#)
- ◆ [“ULEnablePalmRecordDB function” on page 210.](#)
- ◆ [“ULEnableStrongEncryption function” on page 211.](#)
- ◆ [“ULEnableUserAuthentication function” on page 212.](#)
- ◆ [“ULRegisterErrorCallback function” on page 213.](#)
- ◆ [“ULRegisterSchemaUpgradeObserver function” on page 216.](#)

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	<p>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.</p> <p>For embedded SQL users, opening a ULConnection object is equivalent to the EXEC SQL CONNECT statement.</p>

Close method

Prototype	bool Close ()
Description	<p>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.</p> <p>Closing a connection rolls back any outstanding changes.</p> <p>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 (deprecated)” on page 314.</p>
Returns	<p>true (1) if successful.</p> <p>false (0) if unsuccessful.</p>
Example	<p>The following example closes a ULConnection object:</p> <pre>conn.Close();</pre>
See also	“Open method” on page 313

Commit method

Prototype	bool Commit()
Description	Commits outstanding changes to the database.
Returns	<p>true (1) if successful.</p> <p>false (0) if unsuccessful.</p>
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 316](#)

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 432](#).

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 \(SQLCA\)” \[ASA Programming Guide,](#)

GetDatabaseID method

Prototype ul_u_long **ULGetDatabaseID()**

Description ULGetDatabaseID returns the current database ID used for global autoincrement. It returns the value set by the last call to SetDatabaseID or UL_INVALID_DATABASE_ID if the ID was never set.

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” [*MobiLink Clients*, page 294]
“Declaring default global autoincrement columns” [*MobiLink Clients*, page 291]

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 432.

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 [ASA Error Messages](#)

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](#)” [*ASA Error Messages*, page 2].
- ◆ **stream_error** The communication stream error code from the last synchronization. For a listing, see “[MobiLink Communication Error Messages](#)” [*ASA Error Messages*, page 549].
- ◆ **upload_ok** Set to **true** if the upload was successful; **false** otherwise.
- ◆ **ignored_rows** Set to **true** if uploaded rows were ignored; **false** otherwise.
- ◆ **auth_status** The synchronization authentication status. For more information, see “[auth_status parameter](#)” on page 419.
- ◆ **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 420.
- ◆ **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 428.

Returns

The method returns a boolean value.

true Success.

false Failure.

See also

“[PalmLaunch method \[deprecated\]](#)” on page 335

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 <code>SetDatabaseID</code> .
Returns	The percent usage of the available global autoincrement values.
See also	“SetDatabaseID method” on page 316

GrantConnectTo method

Prototype	bool GrantConnectTo (<i>userid</i> , <i>password</i>)
Parameters	userid Character array holding the user ID. The maximum length is 16 characters. password Character array holding the password for <i>userid</i> . 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 in UltraLite” [UltraLite Database User’s Guide, page 40] “Authenticating users” on page 47 “RevokeConnectFrom method” on page 315

InitSynchInfo method

Prototype	an_SQL_code InitSynchInfo (ul_synch_info * <i>synch_info</i>)
Description	Initializes the <i>synch_info</i> 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 317
----------	--

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 313](#)

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 309](#)

LastFetchOK method

Prototype	bool LastFetchOK()
Description	<p>Provides a convenient way of checking that the most recent fetch of a row succeeded (true) or failed (false).</p> <p>The value is reset by any subsequent UltraLite database operation, including those on other connections.</p>
Returns	<p>true (1) if successful.</p> <p>false (0) if unsuccessful.</p>
Example	<p>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:</p> <pre> tb.Open(&conn); tb.Last(); tb.GetID(iVal); if(tb.LastFetchOK()){ ... operations on success... } </pre>
See also	<p>“AfterLast method” on page 320</p> <p>“First method” on page 322</p>

Open method

Prototype	<pre> bool Open (ULData* db, ul_char* userid, ul_char* password, ul_char* name = SQLNULL) bool Open (SQLCA * sqlca, char * start-params) </pre>
Description	Open a connection to a database. The ULData object must be open for this call to succeed.
Parameters	<p>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.</p> <p>userid The user ID argument is a placeholder reserved for possible future use. It is ignored.</p> <p>☞ For more information on user IDs and UltraLite, see “User authentication in UltraLite” [UltraLite Database User’s Guide, page 40].</p>

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.

start-params A connection string, consisting of a semicolon-separated list of keyword=value pairs.

☞ For information on allowed keywords, see [“Connection Parameters” \[UltraLite Database User’s Guide, page 63\]](#).

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 306](#)

Reopen method (deprecated)

Prototype **bool Reopen ()**

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

Description

Deprecated feature

The Reopen method is no longer needed and is deprecated. Use `ULConnection::Open` instead.

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 313](#)

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 432](#).

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 308](#)

[“Timestamp-based synchronization” \[MobiLink Administration Guide, page 48\]](#)

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 in UltraLite” \[UltraLite Database User’s Guide, page 40\]](#)

[“Authenticating users” on page 47](#)

[“GrantConnectTo method” on page 311](#)

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 306](#)

RollbackPartialDownload method

Roll back the changes from a failed synchronization.

Prototype bool **RollbackPartialDownload ()**

Description When a communication error occurs during the download phase of synchronization, UltraLite can apply the downloaded changes, so that the synchronization can be resumed from the place it was interrupted. If the download changes are not needed (the user or application does not want to resume the download at this point), RollbackPartialDownload rolls back the failed download transaction.

See also

- ◆ [“Resuming failed downloads” \[MobiLink Administration Guide, page 74\]](#)
- ◆ [“Keep Partial Download synchronization parameter” \[MobiLink Clients, page 321\]](#)
- ◆ [“Partial Download Retained synchronization parameter” \[MobiLink Clients, page 324\]](#)
- ◆ [“Resume Partial Download synchronization parameter” \[MobiLink Clients, page 327\]](#)

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_DATABASE_ID option [database]” [ASA Database Administration Guide, page 656] “GlobalAutoincUsage method” on page 311

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]” [ASA SQL Reference, page 630] “StopSynchronizationDelete method” on page 317

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]” [ASA SQL Reference, page 630] “StartSynchronizationDelete method” on page 317

Synchronize method

Prototype	bool Synchronize (ul_synch_info * <i>synch_info</i>)
Description	Synchronizes an UltraLite database.

☞ For a detailed description of the members of the *synch_info* structure, see [“Synchronization parameters” on page 417](#).

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 = ULStream();
synch_info.stream_parms =
    UL_TEXT( "host=localhost" );
conn.Synchronize( &synch_info );
```

See also [“Synchronization parameters” on page 417](#)

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 304](#).

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 324](#)

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 324](#)

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 321](#)
[“Last method” on page 325](#)

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 320](#)
[“First method” on page 322](#)

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 354](#)

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 325](#)
[“Update method” on page 329](#)

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 321](#)
[“Last method” on page 325](#)

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	<p>Gets a value from the specified column.</p> <p>column A 2-byte integer. The first column is column 1.</p> <p>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 <i>including</i> the space used for the terminator.</p> <p>isNULL If a value in a column is NULL, isNull is set to true. In this case, the value argument is not meaningful.</p>
Returns	<p>true (1) if successful.</p> <p>false (0) if unsuccessful.</p>
See also	<p>“Get generated method” on page 352</p> <p>“Set method” on page 328</p>

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 <code>tb</code> and places the number of columns in the variable <code>numCol</code> :

```
tb.Open( &conn );
numCol = tb.GetColumnCount();
```

GetColumnSize method

Prototype	ul_length GetColumnSize (ul_column_num <i>column-index</i>)
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 324](#)

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 319](#):

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 319](#)

[“Get generated method” on page 352](#)

[“GetColumnSQLType method” on page 324](#)

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 319](#):

Example The following example gets the column type for the third column in the table **tb**:

```
tb.Open( &conn );  
colType = tb.GetColumnSQLType( 3 );
```

See also [“Data types enumeration” on page 319](#)

[“Get generated method” on page 352](#)

[“GetColumnType method” on page 324](#)

GetSQLCode method

This is a convenience method that calls the `ULConnection::GetSQLCode` method.

☞ For more information see [“GetSQLCode method” on page 309](#).

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 [“Accessing data” on page 46](#).

See also [“Delete method” on page 321](#)

[“Update method” on page 329](#)

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 `ULCursor` object is not open.

See also [“Open method” on page 327](#)

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 320 “First method” on page 322
----------	--

LastCodeOK method

This is a convenience method that calls the **ULConnection::LastCodeOK** method.

☞ For more information see [“LastCodeOK method” on page 312](#).

LastFetchOK method

This is a convenience method that calls the **ULConnection::LastFetchOK** method.

☞ For more information see [“LastFetchOK method” on page 326](#).

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 327 “Relative method” on page 327
----------	--

Open method

Prototype	bool Open (ULConnection * <i>conn</i>)
Description	<p>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.</p> <p>When using Open from the ULTable subclass of ULCursor, do not open two connections on a ULTable object at one time.</p>
Returns	<p>true (1) if successful.</p> <p>false (0) if unsuccessful.</p>
Example	<p>The following example opens a result set object (which extends the cursor class) and moves the cursor position to the first row:</p> <pre>rs.Open(&conn); rs.BeforeFirst(); rs.Next();</pre>
See also	<p>“Close method” on page 321</p> <p>“Open method” on page 348</p>

Previous method

Prototype	bool Previous ()
Description	Moves the cursor position to the previous row in the table or result set.
Returns	<p>true (1) if successful.</p> <p>false (0) if unsuccessful.</p>
Example	<p>The following example moves to the last row in a table:</p> <pre>tb.Open(&conn); tb.AfterLast(); tb.Previous();</pre>
See also	<p>“Next method” on page 326</p> <p>“Relative method” on page 327</p>

Relative method

Prototype	bool Relative (ul_fetch_offset <i>offset</i>)
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 <i>offset</i>

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 326](#)
[“Previous method” on page 327](#)

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 open 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 313](#)

Set method

Prototype **bool Set(ul_column_num colnum, value)**

	<p><i>value:</i></p> <ul style="list-style-type: none"> p_ul_binary <i>buffer-name</i>, ul_length <i>buffer-length</i> ul_char * <i>buffer-name</i>, ul_length <i>buffer-length</i> = 0 DECL_DATETIME <i>date-value</i> DECL_UNSIGNED_BIGINT <i>bigint-value</i> unsigned char <i>char-value</i> double <i>double-value</i> float <i>float-value</i> [unsigned] long <i>long-value</i> [unsigned] short <i>short-value</i>
Description	<p>Sets a value in the specified column, for the current row.</p> <p>colnum A 2-byte integer. The first column is column 1.</p> <p>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.</p>
Returns	<p>true (1) if successful.</p> <p>false (0) if unsuccessful.</p>
See also	<p>“Get method” on page 322</p>

SetColumnNull method

Prototype	int SetColumnNull (ul_column_num <i>column-index</i>)
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	<p>true (1) if successful.</p> <p>false (0) if unsuccessful.</p>
See also	<p>“SetNull<Column> generated method” on page 355</p>

Update method

Prototype	bool Update ()
Description	Updates a row in the table with values specified in previous Set methods.
Returns	<p>true (1) if successful.</p> <p>false (0) if unsuccessful.</p>

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 321](#)

[“Insert method” on page 325](#)

ULData class

Object	Represents an UltraLite database.
Prototype	<pre>ULData db; db.Open();</pre>
Description	<p>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.</p> <p>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.</p> <p>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.</p> <p>For embedded SQL users, opening a ULData object is equivalent to calling db_init.</p> <p>☞ For its position in the API hierarchy, see “C++ API class hierarchy” on page 304.</p>
Example	<p>The following example declares a ULData object and opens it:</p>

```
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

Use **ULSetSynchInfo** to save the state of the application before calling **ULData.Close**. Use the **Open** method when the application is reactivated. For more information, see “[Open method](#)” on page 333 and “[ULSetSynchInfo function](#)” on page 386.

Returns	<p>true (1) if successful.</p> <p>false (0) if unsuccessful.</p>
Example	The following example closes a ULData object:

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

See also [“Open method” on page 333](#)

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 when not connected to the database (but after any **UENable** 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 222](#).

Returns

- ◆ **true** Indicates that database files was successfully deleted.
- ◆ **false** You can obtain detailed error information by using `GetSQLCode`. The usual reason for failure is that an incorrect filename was supplied or that access to the file was denied, perhaps because it is opened by an application.

Initialize method

Prototype `bool Initialize (SQLCA * ca)`

Description Initializes UltraLite for communications with a database. The method does not start the database. Typically, applications start the database when connecting (`ULConnection::Open`).

Use this function before any other data management functions, including `ULConnection::Open`.

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.

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 333](#)

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: <ul style="list-style-type: none"> ◆ 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 [deprecated]” on page 335. ◆ 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 “[Setting UltraLite database properties](#)” [*UltraLite Database User’s Guide*, page 33].

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 222.

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 331

“[Setting UltraLite database properties](#)” [*UltraLite Database User’s Guide*, page 33]

“[UL_STORE_PARMS macro](#)” on page 222

PalmExit method [deprecated]

Prototype

bool **PalmExit**(SQLCA *ca)

bool **PalmExit**(ul_synch_info * synch_info)

Description

Deprecated feature

This function is no longer required. Use `ULData::Close` instead.

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 conduit synchronizes with the MobiLink synchronization server through a TCP/IP or HTTP stream. You can tune the synchronization by specifying a set of stream parameters or network protocol options 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 417](#).

Returns

`true` (1) if successful.

`false` (0) if unsuccessful

PalmLaunch method [deprecated]

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

Deprecated feature

This function is no longer required. Use `ULData::Initialize` instead.

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 417](#).

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 [deprecated]

Prototype

```
bool Reopen ( )
```

```
bool Reopen( SQLCA* ca )
```

Description

Deprecated feature

This function is no longer required. Use `ULData::Open` or `ULData::Initialize` instead.

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 333](#)

StartDatabase method

Prototype `bool StartDatabase (ul_char const * parms);`

Description Start a database if the database is not already running. This function is required when developing applications that combine the Static C++ API and the C++ Component. See [“Combining UltraLite C/C++ interfaces” on page 108](#).

You must call `ULData::Initialize()` before calling `StartDatabase`. If you use the `StartDatabase` method to start a database, any information in `UL_STORE_PARMS` is ignored.

The return value is true if the database was already running or successfully started. Error information is returned in the SQLCA.

Parameters ♦ **parms** A connection string identifying the database to start. Typically, this includes only a database file:

```
DBF=mydatabase.udb
```

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

false (0) if unsuccessful.

StopDatabase method

Prototype `bool StopDatabase (ul_char const * parms);`

Description Stop a database. This function is not commonly needed, as UltraLite automatically stops the database when all connections are closed.

This function may be useful when developing applications that combine the Static C++ API and the C++ Component. See [“Combining UltraLite C/C++ interfaces” on page 108](#).

You must call `ULData::Close()` after calling `StartDatabase`.

The return value is true if the database was successfully stopped. Error information is returned in the SQLCA.

Parameters

- ◆ **parms** A connection string identifying the database to stop. Typically, this includes only a database file:

```
DBF=mydatabase.udb
```

Returns

true (1) if successful.

false (0) if unsuccessful.

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 319](#), and [“Generated result set class” on page 347](#).

☞ For its position in the API hierarchy, see [“C++ API class hierarchy” on page 304](#).

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 348](#)

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 304.

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::StopSynchronizationDelete** method.

See also “[StartSynchronizationDelete method](#)” on page 317

“[StopSynchronizationDelete method](#)” on page 317

Find method

Equivalent to the FindNext method.

☞ See “[FindNext method](#)” on page 343.

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 354.

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 342](#)

[“FindNext method” on page 343](#)

[“FindPrevious method” on page 343](#)

[“LookupBackward method” on page 344](#)

[“LookupForward method” on page 345](#)

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 354](#).

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 341](#)

[“FindNext method” on page 343](#)

[“FindPrevious method” on page 343](#)

[“LookupBackward method” on page 344](#)

[“LookupForward method” on page 345](#)

FindNext method

Prototype	<code>bool FindNext(ul_column_num ncols)</code>
Description	<p>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.</p> <p>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 354.</p> <p>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().</p>
Parameters	<p>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 <i>ncols</i> value of 1.</p>
Returns	<p>true (1) if successful.</p> <p>false (0) if unsuccessful.</p>
See also	<p>“FindFirst method” on page 341</p> <p>“FindLast method” on page 342</p> <p>“FindPrevious method” on page 343</p> <p>“LookupBackward method” on page 344</p> <p>“LookupForward method” on page 345</p>

FindPrevious method

Prototype	<code>bool FindPrevious(ul_column_num ncols)</code>
Description	<p>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.</p> <p>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 354.</p> <p>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().</p>

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 <i>ncols</i> value of 1.
Returns	true (1) if successful. false (0) if unsuccessful.
See also	“FindFirst method” on page 341 “FindLast method” on page 342 “FindNext method” on page 343 “LookupBackward method” on page 344 “LookupForward method” on page 345

Lookup method

Equivalent to the `LookupForward` method.

☞ See [“LookupForward method” on page 345](#)

GetRowCount method

Prototype	<code>ul_ul_long GetRowCount()</code>
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 317 “StopSynchronizationDelete method” on page 317

LookupBackward method

Prototype	<code>bool LookupBackward(ul_column_num <i>ncols</i>)</code>
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 <code>Open</code> method. The

default index is the primary key. For more information, see [“Open method” on page 354](#).

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 <i>ncols</i> value of 1.
Returns	true (1) if successful. false (0) if unsuccessful.
See also	“FindFirst method” on page 341 “FindLast method” on page 342 “FindNext method” on page 343 “FindPrevious method” on page 343 “LookupForward method” on page 345


LookupForward method

Prototype	bool LookupForward (ul_column_num <i>ncols</i>)
Description	<p>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.</p> <p>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 354.</p> <p>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().</p>
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 <i>ncols</i> value of 1.

Returns **true** (1) if successful.
 false (0) if unsuccessful.

See also [“FindFirst method” on page 341](#)
 [“FindLast method” on page 342](#)
 [“FindNext method” on page 343](#)
 [“FindPrevious method” on page 343](#)
 [“LookupBackward method” on page 344](#)

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 <pre><i>result-set</i> rs; rs.Open();</pre> <p><i>result-set</i>: generated name</p>
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 304 .
See also	“ULCursor class” on page 319 “ul_add_statement system procedure” [UltraLite Database User’s Guide, page 210]

Get<Column> generated method

Prototype	bool Getcolumn-name (type* <i>variable</i> , [ul_length* <i>length</i> ,] bool* <i>isNull</i> = UL_NULL)
Description	Retrieves a value from <i>column-name</i> . 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 348

Open method

Prototype	bool Open (ULConnection * conn, datatype <i>value</i> , ...)
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 340](#)

Set<Column> generated method

Prototype	bool Setcolumn-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 347](#)

[“SetNull<Column> generated method” on page 349](#)

SetNull<Column> generated method

Prototype	bool SetNull <i>column-name</i> ()
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 348

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 304.

Execute method

Prototype	<code>bool Execute(ULConnection* <i>conn</i>, [<i>datatype column-name</i>,...])</code>
Description	<p>Executes a named statement that does not return a result set. Any change made is not permanent until it is committed.</p> <p>When a statement is defined using <code>ul_add_statement</code>, you supply placeholders for the values, and supply them at run time. The generated prototype has a data type and name for each value.</p> <p>If the set of parameters is omitted, each value must be set using <code>SetNull</code> or <code>SetParameter</code> before the <code>Execute</code> is called.</p>
Parameters	<p>conn The connection on which the statement is to be executed.</p> <p>datatype A member of the UltraLite data type enumeration.</p> <p>column-name The name of the column.</p>
Returns	<p>true (1) if successful.</p> <p>false (0) if unsuccessful.</p>
See also	<p>“<code>ul_add_statement</code> system procedure” [<i>UltraLite Database User’s Guide</i>, page 210]</p> <p>“<code>SetNull</code> method” on page 350</p> <p>“<code>SetParameter</code> method” on page 351</p>

SetNull method


Prototype	<code>bool SetNull(ULConnection* <i>conn</i>, <i>ul_column_num column-index</i>)</code>
Description	Sets the specified column to NULL.

Parameters	<p>conn The connection on which the statement is to be executed.</p> <p>column-index The number of the column. The first column in the table has a value of one.</p>
Returns	<p>true (1) if successful.</p> <p>false (0) if unsuccessful.</p>

SetParameter method

Prototype	<p>virtual bool SetParameter(int <i>argnum</i>, <i>value-reference</i>)</p> <p><i>value-reference</i>:</p> <ul style="list-style-type: none"> [unsigned] long & <i>value</i> p_ul_binary <i>value</i> unsigned char & <i>value</i> ul_char * <i>value</i> double & <i>value</i> float & <i>value</i> [unsigned] short & <i>value</i> DECL_DATETIME <i>value</i> DECL_BIGINT <i>value</i> DECL_UNSIGNED_BIGINT <i>value</i>
Description	<p>This method sets parameters for statements before the Execute method is called.</p> <p>To set parameters to NULL, use the SetNull method.</p>
Parameters	<p>argnum An identifier for the argument to be set. The first argument is 1, the second 2, and so on.</p> <p>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.</p>
Returns	<p>true (1) if successful.</p> <p>false (0) if unsuccessful. If you supply a parameter of the wrong data type, the method fails.</p>
See also	<p>“Execute method” on page 350</p> <p>“SetNull method” on page 350</p>

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	<pre>Tablename tb; tb.Open();</pre> <p><i>Tablename:</i> generated name</p>
Description	The UltraLite generator defines a class for each table in a named 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 304.
See also	“ ULCursor class ” on page 319 “ ULTable class ” on page 341

Get generated method

Prototype	<pre>bool Get (ul_column_num column-index, value-declaration, bool* is-null = UL_NULL);</pre> <p><i>value-declaration:</i> ul_char * <i>buffer-name</i>, ul_length <i>buffer-length</i> p_ul_binary <i>buffer-name</i>, ul_length <i>buffer-length</i> DECL_DATETIME & <i>date-value</i> { DECL_BIGINT DECL_UNSIGNED_BIGINT } & <i>bigint-value</i> unsigned char & <i>char-value</i> double & <i>double-value</i> float & <i>float-value</i> [unsigned] long & <i>integer-value</i> [unsigned] short & <i>short-value</i></p>
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 319](#)
[“Get method” on page 322](#)
[“Get<Column> generated method” on page 353](#)
[“GetColumnSize method” on page 323](#)

Get<Column> generated method

Prototype `bool Getcolumn-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 352](#)

GetSize<Column> generated method

Prototype	ul_length GetSize <i>column-name</i> ()
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 324

Open method

Prototype	bool Open (ULConnection* <i>conn</i>) bool Open (ULConnection* <i>conn</i> , ul_index_num <i>index</i>)
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 355 .
Returns	When the table is opened, the cursor is positioned before the first row. true (1) if successful. false (0) if unsuccessful.
Example	The following example declares a generated object for a table named ULProduct, and opens it: <pre>ULData db; ULConnection conn; ULProduct tb; db.Open(); conn.Open(&db, "DBA", "SQL"); tb.Open(&conn);</pre>
See also	“Close method” on page 321 “Index enumeration” on page 355

Set<Column> generated method

Prototype	bool Set <i>column-name</i> ()
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 329

SetNull<Column> generated method

Prototype	bool SetNull <i>column-name</i> ()
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 329

Index enumeration

Prototype	enum{ <i>index-name</i> ,... }
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 354

CHAPTER 15

Embedded SQL API Reference

About this chapter

This chapter lists functions that can be used in UltraLite embedded SQL applications. It does not include the embedded SQL EXEC SQL statement and the SQL statements that can be included in your application. For information about these statements, see [“Developing Applications Using Embedded SQL”](#) on page 61.

Use the EXEC SQL INCLUDE SQLCA command to include prototypes for the functions in this chapter.

Contents

Topic:	page
db_fini function	359
db_init function	360
db_start_database function	361
db_stop_database function	362
ULActiveSyncStream function	363
ULChangeEncryptionKey function	364
ULClearEncryptionKey function	365
ULCountUploadRows function	366
ULDropDatabase function	367
ULGetDatabaseID function	368
ULGetLastDownloadTime function	369
ULGetSynchResult function	370
ULGlobalAutoincUsage function	372
ULGrantConnectTo function	373
ULHTTPSStream function	374
ULHTTPStream function	375
ULIsSynchronizeMessage function	376
ULPalmDBStream function (deprecated)	377

Topic:	page
ULPalmExit function (deprecated)	378
ULPalmLaunch function (deprecated)	379
ULResetLastDownloadTime function	380
ULRetrieveEncryptionKey function	381
ULRevokeConnectFrom function	382
ULRollbackPartialDownload function	383
ULSaveEncryptionKey function	384
ULSetDatabaseID function	385
ULSetSynchInfo function	386
ULSocketStream function	387
ULSynchronize function	388

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.

Call **db_fini** once for each *SQLCA* being used.

See also [“db_init function” on page 360](#)

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:

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

Deprecated feature

It is recommended that you call **ULEnable** functions immediately *after* calling **db_init**. Calling **ULEnable** functions before **db_init** is deprecated.

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 359](#)

db_start_database function

Prototype	unsigned int db_start_database (SQLCA * <i>sqlca</i> , char * <i>parms</i>);
Parameters	<p>sqlca A pointer to a SQLCA structure. For information, see “Initializing the SQL Communications Area” on page 64.</p> <p>parms A NULL-terminated string containing a semi-colon-delimited list of parameter settings, each of the form KEYWORD=<i>value</i>. Typically, only a filename is required. For example,</p> <pre>"DBF=c:\\db\\mydatabase.db"</pre> <p>☞ For an available list of connection parameters, see “Connection Parameters” [<i>UltraLite Database User's Guide</i>, page 63].</p>
Description	<p>Start a database if the database is not already running. This function is required when developing applications that combine embedded SQL and the C++ Component. See “Combining UltraLite C/C++ interfaces” on page 108.</p> <p>If you use the <code>db_start_database</code> function to start a database, any information in <code>UL_STORE_PARMS</code> is ignored.</p> <p>If the database was already running or was successfully started, the return value is true (non-zero) and <code>SQLCODE</code> is set to 0. Error information is returned in the <code>SQLCA</code>.</p>

db_stop_database function

Prototype	unsigned int db_stop_database (SQLCA * <i>sqlca</i> , char * <i>parms</i>);
Parameters	<p>sqlca A pointer to a SQLCA structure. For information, see “Initializing the SQL Communications Area” on page 64.</p> <p>parms A NULL-terminated string containing a semi-colon-delimited list of parameter settings, each of the form KEYWORD=<i>value</i>. Typically, only a database file is needed. For example,</p> <pre>"DBF=c:\\db\\mydatabase.db"</pre> <p>☞ For an available list of connection parameters, see “Connection Parameters” [UltraLite Database User’s Guide, page 63].</p>
Description	<p>Stop a database. This function is not commonly needed, as UltraLite automatically stops the database when all connections are closed.</p> <p>This function may be useful when developing applications that combine embedded SQL and the C++ Component. See “Combining UltraLite C/C++ interfaces” on page 108.</p> <p>This function does not stop a database that has existing connections.</p> <p>A return value of TRUE indicates that there were no errors.</p>

ULActiveSyncStream function

Prototype	<code>ul_stream_defn ULActiveSyncStream(void);</code>
Description	<p>Defines an ActiveSync stream suitable for synchronization.</p> <p>The ActiveSync stream is available only on Windows CE devices.</p> <p>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.</p>
See also	<p>“ULIsSynchronizeMessage function” on page 376</p> <p>“ULSynchronize function” on page 388</p> <p>“Synchronize method” on page 317</p> <p>“ActiveSync protocol options” [<i>MobiLink Clients</i>, page 341]</p>

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 [“Encrypting data” on page 87](#)

ULClearEncryptionKey function

Prototype	<pre>ul_bool ULClearEncryptionKey(ul_u_long * creator, ul_u_long * feature-num);</pre>
Description	<p>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.</p> <p>This function clears the encryption key.</p>
Parameters	<p>creator A pointer to the creator ID of the feature holding the encryption key. A value of NULL is the default.</p> <p>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.</p>
See also	<p>“ULRetrieveEncryptionKey function” on page 381</p> <p>“ULSaveEncryptionKey function” on page 384</p> <p>“Palm OS considerations” [<i>UltraLite Database User’s Guide</i>, page 38]</p>

ULCountUploadRows function

Prototype	<pre>ul_u_long ULCountUploadRows (SQLCA * sqlca, ul_publication_mask publication_mask, ul_u_long threshold);</pre>
Description	<p>Returns the number of rows that need to be synchronized, either in a set of publications or in the whole database.</p> <p>One use of the function is to prompt users to synchronize.</p>
Parameters	<p>sqlca A pointer to the SQLCA.</p> <p>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.:</p> <pre>UL_PUB_PUB1 UL_PUB_PUB2</pre> <p>☞ For more information on publication masks, see “Designing sets of data to synchronize separately” [<i>MobiLink Clients</i>, page 280].</p> <p>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.</p>
Example	<p>The following call checks the entire database for the number of rows to be synchronized:</p> <pre>count = ULCountUploadRows(sqlca, 0, 0);</pre> <p>The following call checks publications PUB1 and PUB2 for a maximum of 1000 rows:</p> <pre>count = ULCountUploadRows(sqlca, UL_PUB_PUB1 UL_PUB_PUB2, 1000);</pre> <p>The following call checks to see if any rows need to be synchronized:</p> <pre>count = ULCountUploadRows(sqlca, UL_SYNC_ALL, 1);</pre>

ULDropDatabase function

Prototype `ul_bool ULDropDatabase (SQLCA * sqlca, ul_char * store-params);`

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 when not connected to the database (but after any **ULEnable** functions have been called)

Parameters **sqlca** A pointer to the SQLCA.
store-params 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 222.

Returns

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

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

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

ULGetDatabaseID function

Prototype	ul_u_long ULGetDatabaseID (SQLCA * <i>sqlca</i>)
Description	ULGetDatabaseID returns the current database ID used for global autoincrement. It returns the value set by the last call to SetDatabaseID or UL_INVALID_DATABASE_ID if the ID was never set.
Parameters	sqlca A pointer to the SQLCA.

ULGetLastDownloadTime function

Prototype	<pre>ul_bool ULGetLastDownloadTime(SQLCA * <i>sqlca</i>, ul_publication_mask <i>publication-mask</i>, DECL_DATETIME * <i>value</i>);</pre>
Description	Obtains the last time a specified publication was downloaded.
Parameters	<p>sqlca A pointer to the SQLCA.</p> <p>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.:</p> <pre>UL_PUB_PUB1 UL_PUB_PUB2</pre> <p>☞ For more information on publication masks, see “Designing sets of data to synchronize separately” [<i>MobiLink Clients</i>, page 280].</p> <p>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.</p>
Returns	<ul style="list-style-type: none"> ◆ true Indicates that <i>value</i> is successfully populated by the last download time of the publication specified by <i>publication-mask</i>. ◆ false Indicates that <i>publication-mask</i> specifies more than one publication or that the publication is undefined. If the return value is false, the contents of <i>value</i> are not meaningful.
Examples	<p>The following call populates the dt structure with the date and time that publication UL_PUB_PUB1 was downloaded:</p> <pre>DECL_DATETIME dt; ret = ULGetLastDownloadTime(&sqlca, UL_PUB_PUB1, &dt);</pre> <p>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.</p> <pre>ret = ULGetLastDownloadTime(&sqlca, UL_SYNC_ALL, &dt);</pre>
See also	<p>“UL_SYNC_ALL macro” on page 224</p> <p>“UL_SYNC_ALL_PUBS macro” on page 224</p>

ULGetSynchResult function

Prototype	<code>ul_bool ULGetSynchResult(ul_synch_result * <i>synch-result</i>);</code>
Description	<p>Stores the results of the most recent synchronization, so that appropriate action can be taken in the application:</p> <p>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.</p> <p>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 <code>SQLCODE</code> value set in the connection reflect the result of the connecting operation itself. The synchronization status and results are written to the HotSync log only. To obtain extended synchronization result information, call ULGetSynchResult when connected to the database.</p>
Parameters	<p>synch-result A structure to hold the synchronization result. It is defined in <i>ulglobal.h</i> as follows:</p> <pre>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;</pre> <p>where the individual members have the following meanings:</p> <ul style="list-style-type: none">◆ sql_code The SQL code from the last synchronization. For a list of SQL codes, see “Error messages indexed by Adaptive Server Anywhere SQLCODE” [<i>ASA Error Messages</i>, page 2].◆ stream_error A structure of type <code>ul_stream</code> error. ☞ For more information, see “stream_error synchronization parameter” on page 440.◆ upload_ok Set to true if the upload was successful; false otherwise.◆ ignored_rows Set to true if uploaded rows were ignored; false otherwise.

- ◆ **auth_status** The synchronization authentication status. For more information, see [“auth_status parameter” on page 419](#).
- ◆ **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 420](#).
- ◆ **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 428](#).

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" );
}

```

ULGlobalAutoincUsage function

Prototype	short ULGlobalAutoincUsage (SQLCA * <i>sqlca</i>);
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 385

ULGrantConnectTo function

Prototype	<pre>void ULGrantConnectTo(SQLCA * <i>sqlca</i>, ul_char * <i>userid</i>, ul_char * <i>password</i>);</pre>
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	<p>sqlca A pointer to the SQLCA.</p> <p>userid Character array holding the user ID. The maximum length is 16 characters.</p> <p>password Character array holding the password for <i>userid</i>. The maximum length is 16 characters.</p>
See also	<p>“User authentication in UltraLite” [UltraLite Database User’s Guide, page 40]</p> <p>“Authenticating users” on page 85</p> <p>“ULRevokeConnectFrom function” on page 382</p>

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 388](#)
[“Synchronize method” on page 317](#)
[“stream synchronization parameter” on page 438](#)
[“HTTPS protocol options” \[MobiLink Clients, page 347\]](#)

ULHTTPStream function

Prototype	<code>ul_stream_defn ULHTTPStream(void);</code>
Description	<p>Defines an UltraLite HTTP stream suitable for synchronization via HTTP.</p> <p>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.</p>
See also	<p>“ULSynchronize function” on page 388</p> <p>“Synchronize method” on page 317</p> <p>“stream synchronization parameter” on page 438</p> <p>“HTTP protocol options” [MobiLink Clients, page 346]</p>

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 140](#)

ULPalmDBStream function (deprecated)

Prototype	<code>ul_stream_defn ULPalmDBStream(void);</code>
Description	<p>Defines a stream under the Palm Computing Platform suitable for HotSync and Scout Sync.</p> <p>This function is deprecated. The stream parameter is not needed for HotSync synchronization, and may be UL_NULL.</p>
See also	<p>“HotSync protocol options” [<i>MobiLink Clients</i>, page 343]</p> <p>“Synchronize method” on page 317</p>

ULPalmExit function (deprecated)

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

Description

Deprecated feature

ULPalmExit is no longer required and is a deprecated feature. Use ULSetSynchInfo and db_fini to exit the application instead.

Saves application state for UltraLite applications on the Palm Computing Platform, and writes out an upload stream for HotSync synchronization.

Parameters

sqlca A pointer to the SQLCA.

synch_info A synchronization structure.

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

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

☞ For information on the members of the *synch_info* structure, see [“Synchronization Parameters Reference” on page 415](#).

Returns

The function returns a Boolean value.

true Success.

false Failure.

ULPalmLaunch function (deprecated)

Prototype	<pre> 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; </pre>
Parameters	<p>sqlca A pointer to the SQLCA.</p> <p>synch_info A synchronization structure. For information on the members of this structure, see “Synchronization parameters” on page 417.</p> <p>If you are using TCP/IP or HTTP synchronization, supply UL_NULL as <i>synch_info</i>.</p>
Description	<div style="border: 1px solid black; padding: 5px;"> <p>Deprecated feature ULPalmLaunch is no longer required and is a deprecated feature. Use <code>db_init</code> to launch the application instead.</p> </div> <p>This function restores application state for UltraLite applications on the Palm Computing Platform. This function is required by all UltraLite Palm applications.</p>
See also	<p>“Restoring state in UltraLite Palm applications” on page 121</p> <p>“ULEnableFileDB function” on page 208</p> <p>“ULEnablePalmRecordDB function” on page 210</p>

ULResetLastDownloadTime function

Prototype	<code>void ULResetLastDownloadTime(SQLCA * <i>sqlca</i>, ul_publication_mask <i>publication-mask</i>);</code>
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.: <code>UL_PUB_PUB1 UL_PUB_PUB2</code> ☞ For more information on publication masks, see “ publication synchronization parameter ” on page 432.
Example	The following function call resets the last download time for all tables: <code>ULResetLastDownloadTime(&sqlca, UL_SYNC_ALL);</code>
See also	“ ULGetLastDownloadTime function ” on page 369 “ Timestamp-based synchronization ” [<i>MobiLink Administration Guide</i> , page 48]

ULRetrieveEncryptionKey function

Prototype	<pre>ul_bool ULRetrieveEncryptionKey(ul_char * <i>key</i>, ul_u_short <i>len</i>, ul_u_long * <i>creator</i>, ul_u_long * <i>feature-num</i>);</pre>
Description	<p>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.</p> <p>This function retrieves the encryption key from memory.</p>
Parameters	<p>key A pointer to a buffer in which to hold the retrieved encryption key.</p> <p>len The length of the buffer that holds the encryption key with a terminating null character.</p> <p>creator A pointer to the creator ID of the feature holding the encryption key. A value of NULL is the default.</p> <p>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.</p>
Returns	<ul style="list-style-type: none"> ◆ 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	<p>“ULClearEncryptionKey function” on page 365</p> <p>“ULSaveEncryptionKey function” on page 384</p> <p>“Palm OS considerations” [UltraLite Database User’s Guide, page 38]</p>

ULRevokeConnectFrom function

Prototype	void ULRevokeConnectFrom (SQLCA * <i>sqlca</i> , ul_char * <i>userid</i>);
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 in UltraLite” [UltraLite Database User’s Guide, page 40] “Authenticating users” on page 85 “ULGrantConnectTo function” on page 373

ULRollbackPartialDownload function

Roll back the changes from a failed synchronization.

Prototype

void **ULRollbackPartialDownload** (SQLCA * *sqlca*)

Parameters

- ◆ **sqlca** A pointer to the SQL Communications Area.
In the static C++ API the SQLCA is declared in the header file as **sqlca**.
In the C++ Component use the `Sqlca.GetCA()` method.

Description

When a communication error occurs during the download phase of synchronization, UltraLite can apply the downloaded changes, so that the synchronization can be resumed from the place it was interrupted. If the download changes are not needed (the user or application does not want to resume the download at this point), `ULRollbackPartialDownload` rolls back the failed download transaction.

See also

- ◆ “Resuming failed downloads” [*MobiLink Administration Guide*, page 74]
- ◆ “Keep Partial Download synchronization parameter” [*MobiLink Clients*, page 321]
- ◆ “Partial Download Retained synchronization parameter” [*MobiLink Clients*, page 324]
- ◆ “Resume Partial Download synchronization parameter” [*MobiLink Clients*, page 327]

ULSaveEncryptionKey function

Prototype	<pre>ul_bool ULSaveEncryptionKey(ul_char * <i>key</i>, ul_u_long * <i>creator</i>, ul_u_long * <i>feature-num</i>);</pre>
Description	<p>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.</p> <p>This function saves the encryption key in Palm dynamic memory.</p>
Parameters	<p>key A pointer to the encryption key.</p> <p>creator A pointer to the creator ID of the feature holding the encryption key. A value of NULL is the default.</p> <p>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.</p>
Returns	<ul style="list-style-type: none">◆ true if the operation is successful.◆ false if the operation is unsuccessful.
See also	<p>“ULClearEncryptionKey function” on page 365</p> <p>“ULRetrieveEncryptionKey function” on page 381</p> <p>“Palm OS considerations” [UltraLite Database User’s Guide, page 38]</p>

ULSetDatabaseID function

Prototype	<code>void ULSetDatabaseID(SQLCA * <i>sqlca</i>, ul_u_long <i>id</i>);</code>
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 372

ULSetSynchInfo function

Prototype	<code>ul_bool ULSetSynchInfo(SQLCA * <i>sqlca</i>, ul_synch_info * <i>synch_info</i>);</code>
Description	For HotSync synchronization on the Palm OS, use <code>ULSetSynchInfo</code> to store the synchronization parameters for use when HotSync runs. Typically, <code>ULSetSynchInfo</code> is called just before closing the application by calling <code>db_fini</code> or <code>ULData::Close</code> .
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 417 .

ULSocketStream function

Prototype	<code>ul_stream_defn ULSocketStream(void);</code>
Description	Defines an UltraLite socket stream suitable for synchronization via TCP/IP.
See also	“ULSynchronize function” on page 388 “Synchronize method” on page 317

ULSynchronize function

Prototype	<code>void ULSynchronize(SQLCA * <i>sqlca</i>, ul_synch_info * <i>synch_info</i>);</code>
Description	<p>Initiates synchronization in an UltraLite application.</p> <p>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.</p>
Parameters	<p>sqlca A pointer to the SQLCA.</p> <p>synch_info A synchronization structure. For information on the members of this structure, see “Synchronization parameters” on page 417.</p>
See also	<p>“MobiLink Synchronization Server Options” [<i>MobiLink Administration Guide</i>, page 189]</p> <p>“START SYNCHRONIZATION DELETE statement [MobiLink]” [<i>ASA SQL Reference</i>, page 630]</p>

CHAPTER 16

UltraLite ODBC API Reference

About this chapter This chapter describes those parts of the ODBC interface supported by UltraLite

It is not a comprehensive ODBC reference. It is intended as a quick reference to complement the main reference for ODBC, which is the Microsoft [ODBC SDK documentation](#).

Contents	Topic:	page
	SQLAllocHandle function	391
	SQLBindCol function	392
	SQLBindParameter function	393
	SQLConnect function	394
	SQLDescribeCol function	395
	SQLDisconnect function	396
	SQLEndTran function	397
	SQLExecDirect function	398
	SQLExecute function	399
	SQLFetch function	400
	SQLFetchScroll function	401
	SQLFreeHandle function	402
	SQLGetCursorName function	403
	SQLGetData function	404
	SQLGetDiagRec function	405
	SQLGetInfo function	406
	SQLNumResultCols function	407
	SQLPrepare function	408
	SQLRowCount function	409

Topic:	page
SQLSetCursorName function	410
SQLSetConnectionName function	411
SQLSetSuspend function	412
SQLSynchronize function	413

SQLAllocHandle function

Allocates a handle.

Prototype

```
UL_FN_SPEC SQLRETURN UL_FN_MOD SQLAllocHandle(
    SQLSMALLINT HandleType,
    SQLHANDLE InputHandle,
    SQLHANDLE * OutputHandle);
```

Parameters

- ◆ **HandleType** The type of handle to be allocated. UltraLite supports the following handle types:
 - ◆ SQL_HANDLE_ENV (environment handle)
 - ◆ SQL_HANDLE_DBC (connection handle)
 - ◆ SQL_HANDLE_STMT (statement handle)
- ◆ **InputHandle** The handle in whose context the new handle is to be allocated. For a connection handle, this is the environment handle; for a statement handle, this is the connection handle.
- ◆ **OutputHandle** Pointer to a buffer in which to return the new handle.

Remarks

ODBC uses handles to provide the context for database operations. An environment handle provides the context for communication with a data source, like the SQL Communications Area in other interfaces. A connection handle provides a context for all database operations. A statement handle manages result sets and data modification. A descriptor handle manages the handling of result set data types.

See also

- ◆ [SQLAllocHandle](#) in the Microsoft *ODBC Programmer's Reference*.

SQLBindCol function

Binds a result set column to an application data buffer, for UltraLite ODBC.

Prototype

```
UL_FN_SPEC SQLRETURN UL_FN_MOD SQLBindCol (  
SQLHSTMT StatementHandle,  
SQLUSMALLINT ColumnNumber,  
SQLSMALLINT TargetType,  
SQLPOINTER TargetValue,  
SQLLEN BufferLength,  
SQLLEN * StrLen_or_Ind );
```

Parameters

- ◆ **StatementHandle** A handle for the statement that is to return a result set.
- ◆ **ColumnNumber** The number of the column in the result set to bind to an application data buffer.
- ◆ **TargetType** The identifier of the data type of the TargetValue pointer.
- ◆ **TargetValue** A pointer to the data buffer to bind to the column.
- ◆ **BufferLength** The length of the TargetValue buffer in bytes.
- ◆ **StrLen_or_Ind** Pointer to the length or indicator buffer to bind to the column. For strings, the length buffer holds the length of the actual string that was returned, which may be less than the length allowed by the column.

Remarks

To exchange information between your application and the database, ODBC binds buffers in the application to database objects such as columns. SQLBindCol is used when executing a query to identify a buffer in your application as a place that UltraLite puts the value of a specified column.

See also

- ◆ [SQLBindCol](#) in the Microsoft *ODBC Programmer's Reference*.

SQLBindParameter function

Binds a buffer parameter to a parameter marker in a SQL statement, for UltraLite ODBC.

Prototype

```
UL_FN_SPEC SQLRETURN UL_FN_MOD SQLBindParameter (
    SQLHSTMT StatementHandle,
    SQLUSMALLINT ParameterNumber,
    SQLSMALLINT ParamType,
    SQLSMALLINT CType,
    SQLSMALLINT SqType,
    SQLULEN ColDef,
    SQLSMALLINT Scale,
    SQLPOINTER rgbValue,
    SQLLEN cbValueMax,
    SQLLEN * StrLen_or_Ind );
```

Parameters

- ◆ **StatementHandle** A statement handle.
- ◆ **ParameterNumber** The number of the parameter marker in the statement, in sequential order counting from 1.
- ◆ **ParamType** The parameter type. One of the following:
 - ◆ SQL_PARAM_INPUT
 - ◆ SQL_PARAM_INPUT_OUTPUT
 - ◆ SQL_PARAM_OUTPUT
- ◆ **CType** The C data type of the parameter.
- ◆ **SQLType** The SQL data type of the parameter.
- ◆ **ColDef** The size of the column or expression of the parameter marker.
- ◆ **Scale** The number of decimal digits for the column or expression of the parameter marker.
- ◆ **rgbValue** A pointer to a buffer for the parameter's data.
- ◆ **cbValueMax** The length of the rgbValue buffer.
- ◆ **StrLen_or_Ind** A pointer to a buffer for the parameter's length.

Remarks

To exchange information between your application and the database, ODBC binds buffers in the application to database objects such as columns. SQLBindParameter is used when executing a statement, to identify a buffer in your application as a place that UltraLite gets or sets the value of a specified parameter in a query.

See also

- ◆ [SQLBindParameter](#) in the Microsoft *ODBC Programmer's Reference*.

SQLConnect function

Connects to a database, for UltraLite ODBC.

Prototype

```
UL_FN_SPEC SQLRETURN UL_FN_MOD SQLConnect (  
SQLHDBC ConnectionHandle,  
SQLTCHAR * ServerName,  
SQLSMALLINT NameLength1,  
SQLTCHAR * UserName,  
SQLSMALLINT NameLength2,  
SQLTCHAR * Authentication,  
SQLSMALLINT NameLength3);
```

Parameters

- ◆ **ConnectionHandle** The connection handle.
- ◆ **ServerName** A connection string that defines the database to which your application connects. UltraLite ODBC does not use ODBC data sources. Instead, supply a connection string containing the schema file and database file parameters, together with optional other parameters.

The following is an example of a ServerName parameter:

```
(SQLTCHAR*)UL_TEXT(  
    "schema_file=customer.usm;dbf=customer.udb"  
)
```

For a complete list of connection parameters, see [“Connection Parameters”](#) [*UltraLite Database User’s Guide*, page 63].

- ◆ **NameLength1** The length of * ServerName.
- ◆ **UserName** The user ID to use when connecting. The user ID can alternatively be specified in the connection string supplied to the ServerName parameter.
- ◆ **NameLength2** The length of * UserName.
- ◆ **Authentication** The password to use when connecting. The password can alternatively be specified in the connection string supplied to the ServerName parameter.
- ◆ **NameLength3** The length of * Authentication.

Remarks

Connects to a database. For information about UltraLite connection parameters, see [“Connection Parameters”](#) [*UltraLite Database User’s Guide*, page 63].

See also

- ◆ [SQLConnect](#) in the Microsoft *ODBC Programmer’s Reference*.

SQLDescribeCol function

Returns the result descriptor for a column in the result set.

The result descriptor includes the column name, column size, data type, number of decimal digits, and nullability.

Prototype

```
UL_FN_SPEC SQLRETURN UL_FN_MOD SQLDescribeCol (
    SQLHSTMT StatementHandle,
    SQLUSMALLINT ColumnNumber,
    SQLTCHAR * ColumnName,
    SQLSMALLINT BufferLength,
    SQLSMALLINT * NameLength,
    SQLSMALLINT * Data Type,
    SQLULEN * ColumnSize,
    SQLSMALLINT * DecimalDigits,
    SQLSMALLINT * Nullable );
```

Parameters

- ◆ **StatementHandle** A statement handle.
- ◆ **ColumnNumber** The 1-based column number of result data.
- ◆ **ColumnName** Pointer to a buffer in which to return the column name.
- ◆ **BufferLength** The length of *ColumnName, in characters.
- ◆ **NameLength** Pointer to a buffer in which to return the total number of bytes (excluding the null-termination byte) available to return in *ColumnName.
- ◆ **Data Type** Pointer to a buffer in which to return the SQL data type of the column.
- ◆ **ColumnSize** Pointer to a buffer in which to return the size of the column on the data source.
- ◆ **DecimalDigits** Pointer to a buffer in which to return the number of decimal digits of the column on the data source.
- ◆ **Nullable** Pointer to a buffer in which to return a value that indicates whether the column allows NULL values.

See also

- ◆ [SQLDescribeCol](#) in the Microsoft *ODBC Programmer's Reference*

SQLDisconnect function

Disconnects the application from a database, for UltraLite ODBC.

Prototype

```
UL_FN_SPEC SQLRETURN UL_FN_MOD SQLDisconnect (  
SQLHDBC ConnectionHandle );
```

Parameters

◆ **ConnectionHandle** The handle for the connection to be closed.

Remarks

Once SQLDisconnect is called, no further operations can be carried out against the database without opening a new connection.

See also

- ◆ [“SQLConnect function” on page 394](#)
- ◆ [SQLDisconnect](#) in the Microsoft *ODBC Programmer’s Reference*

SQLEndTran function

Commits or rolls back a transaction, for UltraLite ODBC.

Prototype

```
UL_FN_SPEC SQLRETURN UL_FN_MOD SQLEndTran (  
SQLSMALLINT HandleType,  
SQLHANDLE Handle,  
SQLSMALLINT CompletionType);
```

Parameters

- ◆ **HandleType** The type of handle to be allocated. UltraLite supports the following handle types:
 - ◆ SQL_HANDLE_ENV
 - ◆ SQL_HANDLE_DBC
 - ◆ SQL_HANDLE_STMT
- ◆ **Handle** The connection handle indicating the scope of the transaction.
- ◆ **CompletionType** One of the following two values:
 - ◆ SQL_COMMIT
 - ◆ SQL_ROLLBACK

See also

- ◆ [SQLEndTran](#) in the Microsoft *ODBC Programmer's Reference*

SQLExecDirect function

Executes a SQL statement, for UltraLite ODBC.

Prototype

```
UL_FN_SPEC SQLRETURN UL_FN_MOD SQLExecDirect (  
SQLHSTMT StatementHandle,  
SQLTCHAR * StatementText,  
SQLINTEGER TextLength);
```

Parameters

- ◆ **StatementHandle** A statement handle.
- ◆ **StatementText** The text of the SQL statement.
- ◆ **TextLength** The length of * StatementText.

Remarks

Unlike SQLExecute, the statement does not need to be prepared before being executed using SQLExecDirect.

SQLExecDirect has slower performance than SQLExecute for statements executed repeatedly.

See also

- ◆ [SQLExecDirect](#) in the Microsoft *ODBC Programmer's Reference*

SQLExecute function

Executes a prepared SQL statement, for UltraLite ODBC.

Prototype

```
UL_FN_SPEC SQLRETURN UL_FN_MOD SQLExecute (  
SQLHSTMT StatementHandle );
```

Parameters

◆ **StatementHandle** The handle for the statement to be executed.

Remarks

The statement must be prepared using `SQLPrepare` before it can be executed. If the statement has parameter markers, they must be bound to variables using `SQLBindParameter` before execution.

You can use `SQLExecDirect` to execute a statement without preparing it first. `SQLExecDirect` has slower performance than `SQLExecute` for statements executed repeatedly.

See also

- ◆ [“SQLBindParameter function” on page 393](#)
- ◆ [“SQLPrepare function” on page 408](#)
- ◆ [SQLExecute](#) in the Microsoft *ODBC Programmer’s Reference*.

SQLFetch function

Fetches the next row from a result set and returns data for all bound columns, for UltraLite ODBC.

Prototype

```
UL_FN_SPEC SQLRETURN UL_FN_MOD SQLFetch (  
SQLHSTMT StatementHandle );
```

Parameters

◆ **StatementHandle** A statement handle.

Remarks

Before fetching rows, you must have bound the columns in the result set to buffers using `SQLBindCol`. To fetch a row other than the next row in the result set, use `SQLFetchScroll`.

See also

- ◆ [“SQLFetchScroll function” on page 401](#)
- ◆ [“SQLBindCol function” on page 392](#)
- ◆ [SQLFetch](#) in the Microsoft *ODBC Programmer’s Reference*.

SQLFetchScroll function

Fetches the specified row from the result set and returns data for all bound columns.

Prototype `UL_FN_SPEC SQLRETURN UL_FN_MOD SQLFetchScroll (SQLHSTMT StatementHandle, SQLSMALLINT FetchOrientation, SQLLEN FetchOffset);`

Parameters

- ◆ **StatementHandle** A statement handle.
- ◆ **FetchOrientation** The type of fetch.
- ◆ **FetchOffset** The number of the row to fetch. The interpretation depends on the value of `FetchOrientation`.

Remarks

Before fetching rows, you must have bound the columns in the result set to buffers using `SQLBindCol`. `SQLFetchScroll` is for use in those cases where the more straightforward `SQLFetch` is not appropriate.

See also

- ◆ [“SQLFetch function” on page 400](#)
- ◆ [“SQLBindCol function” on page 392](#)
- ◆ [SQLFetchScroll](#) in the Microsoft *ODBC Programmer’s Reference*.

SQLFreeHandle function

Frees resources for a handle allocated for UltraLite ODBC.

Prototype

```
UL_FN_SPEC SQLRETURN UL_FN_MOD SQLFreeHandle (  
SQLSMALLINT HandleType,  
SQLHANDLE Handle);
```

Parameters

- ◆ **HandleType** The type of handle to be allocated. UltraLite supports the following handle types:
 - ◆ SQL_HANDLE_ENV
 - ◆ SQL_HANDLE_DBC
 - ◆ SQL_HANDLE_STMT
- ◆ **Handle** The handle to be freed.

Remarks

SQLFreeHandle should be called for each handle allocated using SQLAllocHandle, when the handle is no longer needed.

See also

- ◆ [“SQLAllocHandle function” on page 391](#)
- ◆ [SQLFreeHandle](#) in the Microsoft *ODBC Programmer's Reference*.

SQLGetCursorName function

Returns the name associated with a cursor for a specified statement, for UltraLite ODBC.

Prototype

```
UL_FN_SPEC SQLRETURN UL_FN_MOD SQLGetCursorName (  
    SQLHSTMT StatementHandle,  
    SQLTCHAR * CursorName,  
    SQLSMALLINT BufferLength,  
    SQLSMALLINT * NameLength );
```

Parameters

- ◆ **StatementHandle** A statement handle.
- ◆ **CursorName** Pointer to a buffer in which to return the name of the cursor associated with *StatementHandle*.
- ◆ **BufferLength** The length of **CursorName*.
- ◆ **NameLength** Pointer to memory in which to return the total number of bytes (excluding the null-termination character) available to return in **CursorName*.

See also

- ◆ [“SQLSetCursorName function” on page 410](#)
- ◆ [SQLGetCursorName](#) in the Microsoft *ODBC Programmer’s Reference*.

SQLGetData function

Retrieves data for a single column in the result set. SQLGetData is typically used to retrieve variable-length data in parts.

Prototype

```
UL_FN_SPEC SQLRETURN UL_FN_MOD SQLGetData (  
SQLHSTMT StatementHandle,  
SQLUSMALLINT ColumnNumber,  
SQLSMALLINT TargetType,  
SQLPOINTER TargetValue,  
SQLLEN BufferLength,  
SQLLEN * StrLen_or_Ind );
```

Parameters

- ◆ **StatementHandle** A statement handle.
- ◆ **ColumnNumber** The number of the column in the result set to bind.
- ◆ **TargetType** The output handle.
- ◆ **TargetValue** A pointer to the data buffer to bind to the column.
- ◆ **BufferLength** The length of the TargetValue buffer in bytes.
- ◆ **StrLen_or_Ind** Pointer to the length or indicator buffer to bind to the column.

See also

- ◆ [SQLGetData](#) in the Microsoft *ODBC Programmer's Reference*.

SQLGetDiagRec function

Returns the current values of multiple fields of a diagnostic status record, for UltraLite ODBC.

Prototype	<pre> UL_FN_SPEC SQLRETURN UL_FN_MOD SQLGetDiagRec (SQLSMALLINT <i>HandleType</i>, SQLHANDLE <i>Handle</i>, SQLSMALLINT <i>RecNumber</i>, SQLTCHAR * <i>Sqlstate</i>, SQLINTEGER * <i>NativeError</i>, SQLTCHAR * <i>MessageText</i>, SQLSMALLINT <i>BufferLength</i>, SQLSMALLINT * <i>TextLength</i>); </pre>
Parameters	<ul style="list-style-type: none"> ◆ HandleType The type of handle to be allocated. UltraLite supports the following handle types: <ul style="list-style-type: none"> ◆ SQL_HANDLE_ENV ◆ SQL_HANDLE_DBC ◆ SQL_HANDLE_STMT ◆ Handle The input handle ◆ RecNumber The output handle. ◆ Sqlstate The ANSI/ISO SQLSTATE value of the error. For a listing, see “Error messages indexed by SQLSTATE” [ASA Error Messages, page 36]. ◆ NativeError The SQLCODE value of the error. For a listing, see “Error messages indexed by Adaptive Server Anywhere SQLCODE” [ASA Error Messages, page 2]. ◆ MessageText The text of the error or status message. ◆ BufferLength The length of the MessageText buffer in bytes. ◆ TextLength Pointer to a buffer in which to return the total number of bytes (excluding the null-termination byte) available to return in *MessageText.
See also	<ul style="list-style-type: none"> ◆ SQLGetDiagRec in the Microsoft <i>ODBC Programmer’s Reference</i>.

SQLGetInfo function

Returns general information about the current ODBC driver and data source, for UltraLite ODBC.

Prototype

```
UL_FN_SPEC SQLRETURN UL_FN_MOD SQLGetInfo (  
SQLHDBC ConnectionHandle,  
SQLUSMALLINT InfoType,  
SQLPOINTER InfoValue,  
SQLSMALLINT BufferLength,  
SQLSMALLINT ODBC FAR * StringLength );
```

Parameters

- ◆ **ConnectionHandle** A connection handle.
- ◆ **InfoType** The type of information returned. The only type supported is `SQL_DBMS_VER`. The information returned is a character string identifying the current release of the software.
- ◆ **InfoValue** Pointer to a buffer in which to return the information.
- ◆ **BufferLength** The length of the `InfoValue` buffer in bytes.
- ◆ **StringLength** Pointer to a buffer in which to return the total number of bytes (excluding the null-termination character for character data) available to return in `*InfoValue`.

See also

- ◆ [SQLGetInfo](#) in the Microsoft *ODBC Programmer's Reference*.

SQLNumResultCols function

Returns the number of columns in a result set, for UltraLite ODBC.

Prototype

```
UL_FN_SPEC SQLRETURN UL_FN_MOD SQLNumResultCols (  
SQLHSTMT StatementHandle,  
SQLSMALLINT * ColumnCount );
```

Parameters

- ◆ **StatementHandle** A statement handle.
- ◆ **ColumnCount** Pointer to a buffer in which to return the total number of columns in the result set.

See also

- ◆ [SQLNumResultCols](#) in the Microsoft *ODBC Programmer's Reference*.

SQLPrepare function

Prepares a SQL statement for execution, for UltraLite ODBC.

Prototype

```
UL_FN_SPEC SQLRETURN UL_FN_MOD SQLPrepare (  
SQLHSTMT StatementHandle,  
SQLTCHAR * StatementText,  
SQLINTEGER TextLength );
```

Parameters

- ◆ **StatementHandle** A statement handle.
- ◆ **StatementText** Pointer to a buffer that holds the SQL statement text.
- ◆ **TextLength** The length of *StatementText.

See also

- ◆ [“SQLExecute function” on page 399](#)
- ◆ [SQLPrepare](#) in the Microsoft *ODBC Programmer’s Reference*.

SQLRowCount function

Returns the number of rows affected by an INSERT, UPDATE, or DELETE operation, for UltraLite ODBC.

Prototype

```
UL_FN_SPEC SQLRETURN UL_FN_MOD SQLRowCount (  
SQLHSTMT StatementHandle,  
SQLLEN * RowCount );
```

Parameters

- ◆ **StatementHandle** A statement handle.
- ◆ **RowCount** Pointer to a buffer in which the number of rows is returned.

See also

- ◆ [SQLRowCount](#) in the Microsoft *ODBC Programmer's Reference*.

SQLSetCursorName function

Sets the name of a cursor associated with a SQL statement, for UltraLite ODBC.

Prototype

```
UL_FN_SPEC SQLRETURN UL_FN_MOD SQLSetCursorName (  
SQLHSTMT StatementHandle,  
SQLTCHAR * CursorName,  
SQLSMALLINT NameLength );
```

Parameters

- ◆ **StatementHandle** A statement handle.
- ◆ **CursorName** Pointer to a buffer holding the cursor name.
- ◆ **NameLength** The length of *CursorName.

See also

- ◆ [“SQLGetCursorName function” on page 403](#)
- ◆ [SQLSetCursorName](#) in the Microsoft *ODBC Programmer’s Reference*.

SQLSetConnectionName function

Sets a connection name for the suspend and restore operation, for UltraLite ODBC. This function is specific to UltraLite and is not part of the ODBC standard.

Prototype	<code>UL_FN_SPEC SQLRETURN UL_FN_MOD SQLSetConnectionName (SQLHSTMT <i>StatementHandle</i>, SQLTCHAR * <i>ConnectionName</i>, SQLSMALLINT <i>NameLength</i>);</code>
Parameters	<ul style="list-style-type: none">◆ StatementHandle A statement handle.◆ ConnectionName Pointer to a buffer holding the connection name.◆ NameLength The length of *ConnectionName
Remarks	SQLSetConnectionName is used to provide a connection name for use in the suspend and restore operation, together with SQLSetSuspend. Set the connection name before opening a connection in order to restore application state.
See also	<ul style="list-style-type: none">◆ “Saving state in UltraLite Palm applications” on page 120◆ “SQLSetSuspend function” on page 412

SQLSetSuspend function

Indicates whether the state of open cursors should be saved on closing the application, for UltraLite ODBC. This function is specific to UltraLite and is not part of the ODBC standard.

Prototype

```
UL_FN_SPEC SQLRETURN UL_FN_MOD SQLSetSuspend (  
SQLSMALLINT HandleType,  
SQLHSTMT StatementHandle,  
SQLSMALLINT TrueFalse );
```

Parameters

- ◆ **HandleType** The type of handle to be allocated. UltraLite supports the following handle types:
 - ◆ SQL_HANDLE_ENV
 - ◆ SQL_HANDLE_DBC
 - ◆ SQL_HANDLE_STMT
- ◆ **StatementHandle** A statement handle.
- ◆ **TrueFalse** The output handle.

See also

[“Saving state in UltraLite Palm applications” on page 120](#)

SQLSynchronize function

Synchronizes data in the database using MobiLink synchronization, for UltraLite ODBC. This function is specific to UltraLite and is not part of the ODBC standard.

Prototype `UL_FN_SPEC SQLRETURN UL_FN_MOD SQLSynchronize (SQLHDBC ConnectionHandle, ul_synch_info * SynchInfo);`

Parameters

- ◆ **ConnectionHandle** A handle.
- ◆ **SynchInfo** The structure holding synchronization information. See [“Synchronization Parameters Reference” on page 415](#).

Remarks SQLSynchronize is an extension to ODBC. It initiates a MobiLink synchronization operation.

See also

- ◆ [“Synchronization Parameters Reference” on page 415](#).
- ◆ *MobiLink Administration Guide*

CHAPTER 17

Synchronization Parameters Reference

About this chapter

This chapter provides reference information about synchronization parameters.

Contents

Topic:	page
Synchronization parameters	417
auth_parms parameter	418
auth_status parameter	419
auth_value synchronization parameter	420
checkpoint_store synchronization parameter	421
disable_concurrency synchronization parameter	422
download_only synchronization parameter	423
keep_partial_download synchronization parameter	424
ignored_rows synchronization parameter	425
new_password synchronization parameter	426
num_auth_parms parameter	427
observer synchronization parameter	428
partial_download_retained synchronization parameter	429
password synchronization parameter	430
ping synchronization parameter	431
publication synchronization parameter	432
resume_partial_download synchronization parameter	433
security synchronization parameter	434
security_parms synchronization parameter	435
send_column_names synchronization parameter	436
send_download_ack synchronization parameter	437

Topic:	page
stream synchronization parameter	438
stream_error synchronization parameter	440
stream_parms synchronization parameter	443
upload_ok synchronization parameter	444
upload_only synchronization parameter	445
user_data synchronization parameter	446
user_name synchronization parameter	447
version synchronization parameter	448

Synchronization parameters

The synchronization parameters are members of a structure that is provided as an argument in the call to synchronize. The **ul_synch_info** structure that holds the synchronization parameters is defined in *ulglobal.h* as follows:

```

struct ul_synch_info {
    ul_char *      user_name;
    ul_char *      password;
    ul_char *      new_password;
    ul_char *      version;
    p_ul_stream_defn stream;
    ul_char *      stream_parms;
    p_ul_stream_defn security;
    ul_char *      security_parms;
    ul_synch_observer_fn observer;
    ul_void *      user_data;
    ul_publication_mask publication;
    ul_bool        upload_only;
    ul_bool        download_only;
    ul_bool        send_download_ack;
    ul_bool        send_column_names;
    ul_bool        ping;
    ul_bool        checkpoint_store;
    ul_bool        disable_concurrency;
    ul_byte        num_auth_parms;
    ul_char * *    auth_parms;
    ul_bool        keep_partial_download;
    ul_bool        resume_partial_download;

    // fields set on output
    ul_stream_error stream_error;
    ul_bool        upload_ok;
    ul_bool        ignored_rows;
    ul_auth_status auth_status;
    ul_s_long      auth_value;
    ul_bool        partial_download_retained;

    p_ul_synch_info  init_verify;
};

```

The **init_verify** field is reserved for internal use.

Use **UL_TEXT** around constant strings

The **UL_TEXT** macro allows constant strings to be compiled as single-byte strings or wide-character strings. Use this macro to enclose all constant strings supplied as members of the **ul_synch_info** structure so that the compiler handles these parameters correctly.

☞ For a description of the role of each synchronization parameter, see “Synchronization parameters” [*MobiLink Clients*, page 316].

auth_parms parameter

Function Provides parameters to a custom user authentication script.

Usage Set the parameters as follows:

```
ul_char * Params[ 3 ] = { UL_TEXT( "parm1" ),
                          UL_TEXT( "parm2" ),
                          UL_TEXT( "parm3" ) };

// ...
info.num_auth_parms = 3;
info.auth_parms = Params;
```

See also

- ◆ [“Authentication Parameters synchronization parameter”](#) [*MobiLink Clients*, page 316]
- ◆ [“num_auth_parms parameter”](#) on page 427
- ◆ [“authenticate_parameters connection event”](#) [*MobiLink Administration Guide*, page 334]
- ◆ [“authenticate_user connection event”](#) [*MobiLink Administration Guide*, page 336]

auth_status parameter

Function Reports the status of MobiLink user authentication.

Usage Access the parameter as follows:

```
ul_synch_info info;
// ...
returncode = info.auth_status;
```

Allowed values After synchronization, the parameter must hold one of the following values. If a custom **authenticate_user** synchronization script at the consolidated database returns a different value, the value is interpreted according to the rules given in “[authenticate_user connection event](#)” [*MobiLink Administration Guide*, page 336].

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 the time of synchronization but will expire soon.
UL_AUTH_STATUS_EXPIRED	3000	Authorization failed: user ID or password have expired.
UL_AUTH_STATUS_INVALID	4000	Authorization failed: bad user ID or password.
UL_AUTH_STATUS_IN_USE	5000	Authorization failed: user ID is already in use.

See also

- ◆ “[Authentication Status synchronization parameter](#)” [*MobiLink Clients*, page 317]
- ◆ “[Authenticating MobiLink Users](#)” [*MobiLink Clients*, page 9].

auth_value synchronization parameter

Function	Reports return values from custom user authentication synchronization scripts.
Default	The values set by the default MobiLink user authentication mechanism are described in “authenticate_user connection event” [<i>MobiLink Administration Guide</i> , page 336].
Usage	<p>The parameter is read-only.</p> <p>Access the parameter as follows:</p> <pre>ul_synch_info info; // ... returncode = info.auth_value;</pre>
See also	<ul style="list-style-type: none">◆ “Authentication Value synchronization parameter” [<i>MobiLink Clients</i>, page 318]◆ “authenticate_user connection event” [<i>MobiLink Administration Guide</i>, page 336]◆ “authenticate_user_hashed connection event” [<i>MobiLink Administration Guide</i>, page 340]◆ “auth_status parameter” on page 419

checkpoint_store synchronization parameter

Function	Adds additional checkpoints of the database during synchronization to limit database growth during the synchronization process.
Default	By default, limited checkpointing is done.
Usage	Set the parameter as follows: <pre>ul_synch_info info; // ... info.checkpoint_store = ul_true ;</pre>
See also	“Checkpoint Store synchronization parameter” [<i>MobiLink Clients</i> , page 318]

disable_concurrency_synchronization parameter

Function	Disallow database access from other threads during synchronization.
Default	By default, data access is available. Data access is read-write during the download phase, and read-only otherwise.
Usage	Set the parameter as follows: <pre>ul_synch_info info; // ... info.checkpoint_store = ul_false ;</pre>
See also	<ul style="list-style-type: none">◆ “Disable Concurrency synchronization parameter” [<i>MobiLink Clients</i>, page 319]◆ “Understanding concurrency in UltraLite” [<i>UltraLite Database User’s Guide</i>, page 58]

download_only synchronization parameter

Function Do not upload any changes from the UltraLite database during this synchronization.

Default The parameter is an optional Boolean value, and by default is false.

Usage Set the parameter as follows:

```
ul_synch_info info;  
// ...  
info.download_only = ul_true;
```

See also

- ◆ [“Download Only synchronization parameter”](#) [*MobiLink Clients*, page 320]
- ◆ [“Including read-only tables in an UltraLite database”](#) [*MobiLink Clients*, page 283].
- ◆ [“upload_only synchronization parameter”](#) on page 445

keep_partial_download synchronization parameter

Function	On download errors, hold on to partial downloads rather than rolling back all changes.
Default	The parameter is an optional Boolean value, and by default is false.
Usage	Set the parameter as follows: <pre>ul_synch_info info; // ... info.keep_partial_download = ul_true;</pre>
See also	<ul style="list-style-type: none">◆ “Resuming failed downloads” [<i>MobiLink Administration Guide</i>, page 74]◆ “Keep Partial Download synchronization parameter” [<i>MobiLink Clients</i>, page 321]

ignored_rows synchronization parameter

Function

Reports if any rows were ignored by the MobiLink synchronization server during synchronization because of absent scripts.

The parameter is read-only.

new_password synchronization parameter

Function Sets a new MobiLink password associated with the user name.

Default There is no default.

Usage Set the parameter as follows:

```
ul_synch_info info;  
// ...  
info.password = UL_TEXT( "myoldpassword" );  
info.new_password = UL_TEXT( "mynewpassword" );
```

See also

- ◆ [“New Password synchronization parameter”](#) [*MobiLink Clients*, page 322]
- ◆ [“Authenticating MobiLink Users”](#) [*MobiLink Clients*, page 9].

num_auth_parms parameter

Function	The number of authentication parameter strings passed to a custom authentication script.
Default	No parameters passed to a custom authentication script.
Usage	The parameter is used together with auth_parms to supply information to custom authentication scripts. ☞ For more information, see “auth_parms parameter” on page 418 .
See also	<ul style="list-style-type: none">◆ “Number of Authentication Parameters parameter” [MobiLink Clients, page 322]◆ “auth_parms parameter” on page 418◆ “authenticate_parameters connection event” [MobiLink Administration Guide, page 334]◆ “authenticate_user connection event” [MobiLink Administration Guide, page 336]

observer synchronization parameter

Function

A pointer to a callback function that monitors synchronization.

See also

- ◆ “Observer synchronization parameter” [*MobiLink Clients*, page 323]
- ◆ “Monitoring and canceling synchronization” on page 53
- ◆ “user_data synchronization parameter” on page 446

partial_download_retained synchronization parameter

Function	Indicates that a partial downloads was held as a result of failure during the download phase of synchronization.
Default	The parameter is set during synchronization if a download error occurs and a partial download was retained.
See also	<ul style="list-style-type: none">◆ “Resuming failed downloads” [<i>MobiLink Administration Guide</i>, page 74]◆ “Partial Download Retained synchronization parameter” [<i>MobiLink Clients</i>, page 324]

password synchronization parameter

Function	A string specifying the MobiLink password associated with the user_name . This user name and password are separate from any database user ID and password, and serves to identify and authenticate the application to the MobiLink synchronization server.
Default	There is no default.
Usage	Set the parameter as follows: <pre>ul_synch_info info; // ... info.password = UL_TEXT("mypassword");</pre>
See also	<ul style="list-style-type: none">◆ “Password synchronization parameter” [<i>MobiLink Clients</i>, page 324]◆ “Authenticating MobiLink Users” [<i>MobiLink Clients</i>, page 9].

ping synchronization parameter

Function	<p>Confirm communications between the UltraLite client and the MobiLink synchronization server. When this parameter is set to true, no synchronization takes place.</p> <p>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.</p> <p>If the ping succeeds, the MobiLink server issues an information message. If the ping does not succeed, it issues an error message.</p> <p>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.</p> <p>The MobiLink synchronization server may execute the following scripts, if they exist, for a ping request:</p> <ul style="list-style-type: none"> ◆ begin_connection ◆ authenticate_user ◆ authenticate_user_hashed ◆ end_connection
Default	The parameter is an optional Boolean value, and by default is false.
Usage	<p>Set the parameter as follows:</p> <pre>ul_synch_info info; // ... info.ping = ul_true;</pre>
See also	<ul style="list-style-type: none"> ◆ “Ping synchronization parameter” [<i>MobiLink Clients</i>, page 325] ◆ “-pi option” [<i>MobiLink Clients</i>, page 144]

publication synchronization parameter

Function	Specifies the publications to be synchronized.
Default	If you do not specify a publication, all data is synchronized.
Usage	The UltraLite generator identifies the publications specified on the <i>ulgen -v</i> command line option as upper case constants with the name <code>UL_PUB_pubname</code> , where <code>pubname</code> is the name given to the <code>-v</code> option. For example, the following command line generates a publication identified by the constant <code>UL_PUB_SALES</code> :

```
ulgen -v sales ...
```

When synchronizing, set the publication parameter to a **publication mask**: an OR'd list of publication constants. For example:

```
ul_synch_info info;  
// ...  
info.publication = UL_PUB_MYPUB1 | UL_PUB_MYPUB2 ;
```

The special publication mask `UL_SYNC_ALL` describes all the tables in the database, whether in a publication or not. The mask `UL_SYNC_ALL_PUBS` describes all tables in publications in the database.

See also

- ◆ [“Publication synchronization parameter”](#) [*MobiLink Clients*, page 326]
- ◆ [“The UltraLite Generator”](#) [*UltraLite Database User’s Guide*, page 89]
- ◆ [“Designing sets of data to synchronize separately”](#) [*MobiLink Clients*, page 280]

resume_partial_download synchronization parameter

Function	Resume a synchronization interrupted by a communication failuer during the downloads.
Default	The parameter is a Boolean value, and by default is false.
Usage	Set the parameter as follows: <pre>ul_synch_info info; // ... info.resume_partial_download = ul_true;</pre>
See also	<ul style="list-style-type: none">◆ “Resuming failed downloads” [<i>MobiLink Administration Guide</i>, page 74]◆ “Resume Partial Download synchronization parameter” [<i>MobiLink Clients</i>, page 327]

security synchronization parameter

Function	Set the UltraLite client to use Certicom encryption technology when exchanging messages with the MobiLink synchronization server. <div style="border: 1px solid black; padding: 5px;"><p>Separately-licensable option required</p><p>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” [<i>Introducing SQL Anywhere Studio</i>, page 4].</p></div>
Default	The Security parameter is null by default, corresponding to no transport-layer security.
Usage	The security stream is specified in addition to the synchronization stream. Allowed values are as follows: <ul style="list-style-type: none">◆ ULSecureCerticomTLSStream() Elliptic-curve transport-layer security provided by Certicom.◆ ULSecureRSATLSStream() RSA transport-layer security provided by Certicom. <p>☞ For more information, including libraries that you must link against, see “Security synchronization parameter” [<i>MobiLink Clients</i>, page 328].</p>
See also	<ul style="list-style-type: none">◆ “Security synchronization parameter” [<i>MobiLink Clients</i>, page 328]◆ “MobiLink Transport-Layer Security” [<i>MobiLink Administration Guide</i>, page 165].

security_parms synchronization parameter

Function	<p>Sets the parameters required when using transport-layer security. This parameter must be used together with the security parameter.</p> <p>☞ For more information, see “security synchronization parameter” on page 434.</p>
Usage	<p>The <code>ULSecureCerticomTLSStream()</code> and <code>ULSecureRSATLSStream()</code> security parameters take a string composed of the following optional parameters, supplied in an semicolon-separated string.</p> <ul style="list-style-type: none"> ◆ 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. <p>For example:</p> <pre>ul_synch_info info; ... info.stream = ULStream(); info.security = ULSecureCerticomTLSStream(); info.security_parms = UL_TEXT("certificate_company=Sybase") UL_TEXT(";") UL_TEXT("certificate_unit=Sales");</pre> <p>The security_parms parameter is a string, and by default is null.</p> <p>If you use secure synchronization, you must also use the <code>-r</code> command-line option on the UltraLite generator. For more information, see “The UltraLite Generator” [<i>UltraLite Database User’s Guide</i>, page 89].</p>
See also	<ul style="list-style-type: none"> ◆ “Security Parameters synchronization parameter” [<i>MobiLink Clients</i>, page 329]

send_column_names synchronization parameter

Function When **send_column_names** is set to **ul_true** UltraLite sends each column name to the MobiLink synchronization server. By default UltraLite does not send column names.

This parameter is typically used together with the `-za` or `-ze` switch on the MobiLink synchronization server for automatically generating synchronization scripts.

See also

- ◆ [“Send Column Names synchronization parameter”](#) [*MobiLink Clients*, page 330]
- ◆ [“-za option”](#) [*MobiLink Administration Guide*, page 219]

send_download_ack synchronization parameter

Function	<p>Set this boolean parameter to true to instruct the MobiLink synchronization server that the client will provide a download acknowledgement.</p> <p>If the client does send a download acknowledgement, the MobiLink synchronization server worker thread must wait for the client to apply the download. If the client does not send a download acknowledgement, the MobiLink synchronization server is freed up sooner for its next synchronization.</p>
Default	<p>The default setting is false.</p>
See also	<ul style="list-style-type: none">◆ “Send Download Acknowledgement synchronization parameter” [<i>MobiLink Clients</i>, page 331]

stream synchronization parameter

Function Set the MobiLink synchronization stream to use for synchronization.
☞ For more information, see “[stream_parms synchronization parameter](#)” on page 443.

Default The parameter has no default value, and must be explicitly set.

Usage

```
//Embedded SQL
ul_synch_info info;
...
info.stream = ULSocketStream();

//Static C++ API
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 stream functions are available, but not all are available on all target platforms:

Stream	Description
ULActiveSyncStream()	ActiveSync synchronization (Windows CE only). ☞ For a list of stream parameters, see “ ActiveSync protocol options ” [<i>MobiLink Clients</i> , page 341].
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 protocol options ” [<i>MobiLink Clients</i> , page 346].

Stream	Description
ULHTTPStream()	<p>Synchronize via the HTTPS synchronization stream.</p> <p>The HTTPS stream uses SSL or TLS as its underlying protocol. It operates over Internet protocols (HTTP and TCP/IP).</p> <p>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” [<i>Introducing SQL Anywhere Studio</i>, page 4].</p> <p>☞ For a list of stream parameters, see “HTTPS protocol options” [<i>MobiLink Clients</i>, page 347].</p>
ULSocketStream()	<p>Synchronize via TCP/IP.</p> <p>☞ For a list of stream parameters, see “TCP/IP protocol options” [<i>MobiLink Clients</i>, page 345].</p>

See also

- ◆ “Stream Type synchronization parameter” [*MobiLink Clients*, page 332]

stream_error synchronization parameter

Function	Sets a structure to hold communications error reporting information.
Access methods	This feature is not available to Java applications.
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 is listed in *sserror.h*. The following are the meaningful constants:

Constant	Value
STREAM_ID_TCPIP	0
STREAM_ID_PALM_CONDUIT	3
STREAM_ID_PALM_SS	4
STREAM_ID_HTTP	7
STREAM_ID_HTTPS	8
STREAM_ID_SECURE	10
STREAM_ID_CERTICOM	12
STREAM_ID_JAVA_CERTICOM	13
STREAM_ID_CERTICOM_SSL	14
STREAM_ID_CERTICOM_TLS	15
STREAM_ID_WIRESTRM	16
STREAM_ID_ACTIVASYNC	23
STREAM_ID_RSA_TLS	24

Constant	Value
STREAM_ID_JAVA_RSA	25

- ◆ **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](#)” [*ASA Error Messages*, page 549]. For the error code suffixes, see *sserror.h*.

In this version, to find the constant associated with each number you must count down the number of lines prefixed by **DO_STREAM_Error** in *sserror.h*. For example, to find the constant for error number 10, you use the tenth **DO_STREAM_ERROR** entry in *sserror.h*, which is as follows:

```
DO_STREAM_ERROR( WRITE )
```

The constant associated with this error is therefore **STREAM_ERROR_WRITE**.

- ◆ **system_error_code** A system-specific error code. For more information on the error code, you must look at your platform documentation. For Windows platforms, this is the Microsoft Developer Network documentation.

The following are common system errors on Windows:

- **10048 (WSAADDRINUSE)** Address already in use.
- **10053 (WSAECONNABORTED)** Software caused connection abort.
- **10054 (WSAECONNRESET)** The other side of the communication closed the socket.
- **10060 (WSAETIMEDOUT)** Connection timed out.
- **10061 (WSAECONNREFUSED)** Connection refused. Typically, this means that the MobiLink server is not running or is not listening on the specified port.

☞ For a complete listing, see [the Microsoft Developer Network web site](#).

- ◆ **error_string** An application-provided error message. The string may or may not be empty. A non-empty **error_string** provides information in addition to the **stream_error_code**. For instance, for a write error (error

code 9) the error string is a number showing how many bytes it was trying to write.

Usage

Check for `SQLE_COMMUNICATIONS_ERROR` as follows:

```
Connection conn;
auto ul_synch_info info;
...
conn.InitSynchInfo( &info );
info.stream_error.error_string = error_buff;
info.stream_error.error_string_length =
    sizeof( error_buff );
if( !conn.Synchronize( &synch_info ) ){
    if( SQLCODE == SQLE_COMMUNICATIONS_ERROR ){
        printf( error_buff );
        // more error headline here
    }
}
```

See also

- ◆ [“Stream Error synchronization parameter”](#) [*MobiLink Clients*, page 332]

stream_parms synchronization parameter

Function	<p>Sets network protocol options to configure the synchronization stream.</p> <p>A semi-colon separated list of option assignments. Each assignment is of the form <i>keyword=value</i>, where the allowed sets of keywords depends on the synchronization stream.</p> <p>For a list of available options for each stream, see the following sections:</p> <ul style="list-style-type: none"> ◆ “ActiveSync protocol options” [<i>MobiLink Clients</i>, page 341] ◆ “HotSync protocol options” [<i>MobiLink Clients</i>, page 343] ◆ “HTTP protocol options” [<i>MobiLink Clients</i>, page 346] ◆ “HTTPS protocol options” [<i>MobiLink Clients</i>, page 347] ◆ “TCP/IP protocol options” [<i>MobiLink Clients</i>, page 345]
Default	The parameter is optional, is a string, and by default is null.
Usage	<p>Set the parameter as follows:</p> <pre> ul_synch_info info; // ... info.stream_parms= UL_TEXT("host=myserver;port=2439"); </pre>
See also	<ul style="list-style-type: none"> ◆ “Stream Parameters synchronization parameter” [<i>MobiLink Clients</i>, page 334] ◆ “Network protocol options for UltraLite synchronization clients” [<i>MobiLink Clients</i>, page 341].

upload_ok synchronization parameter

Function Reports the status of MobiLink uploads. The MobiLink synchronization server provides this information to the client.

The parameter is read-only.

Usage After synchronization, the **upload_ok** parameter holds **true** if the upload was successful, and **false** otherwise.

Access the parameter as follows:

```
ul_synch_info info;  
// ...  
returncode = info.upload_ok;
```

See also ♦ [“Upload OK synchronization parameter”](#) [*MobiLink Clients*, page 336]

upload_only synchronization parameter

Function	Indicates that there should be no downloads in the current synchronization, which can save communication time, especially over slow communication links. When set to true, the client waits for the upload acknowledgement from the MobiLink synchronization server, after which it terminates the synchronization session successfully.
Default	The parameter is an optional Boolean value, and by default is false.
Usage	Set the parameter to true as follows: <pre>ul_synch_info info; // ... info.upload_only = ul_true;</pre>
See also	<ul style="list-style-type: none">◆ “Upload Only synchronization parameter” [<i>MobiLink Clients</i>, page 337]◆ “Synchronizing high-priority changes” [<i>MobiLink Clients</i>, page 282]◆ “download_only synchronization parameter” on page 423

user_data synchronization parameter

Function	Make application-specific information available to the synchronization observer.
Usage	When implementing the synchronization observer callback function observer , you can make application-specific information available by providing information using user_data .
See also	<ul style="list-style-type: none">◆ “User Data synchronization parameter” [MobiLink Clients, page 338]◆ “observer synchronization parameter” on page 428

user_name synchronization parameter

Function	A string specifying the user name that uniquely identifies the MobiLink client to the MobiLink synchronization server. MobiLink uses this value to determine the download content, to record the synchronization state, and to recover from interruptions during synchronization.
Default	The parameter is required, and is a string.
Usage	Set the parameter as follows: <pre>ul_synch_info info; // ... info.user_name= UL_TEXT("mluser");</pre>
See also	<ul style="list-style-type: none">◆ “User Name synchronization parameter” [<i>MobiLink Clients</i>, page 338]◆ “Authenticating MobiLink Users” [<i>MobiLink Clients</i>, page 9].◆ “MobiLink users” [<i>MobiLink Clients</i>, page 7].

version synchronization parameter

Function	Each synchronization script in the consolidated database is marked with a version string. For example, there may be two different download_cursor scripts, identified by different version strings. The version string allows an UltraLite application to choose from a set of synchronization scripts.
Default	The parameter is a string, and by default is the MobiLink default version string.
Usage	Set the parameter as follows: <pre>ul_synch_info info; // ... info.version = UL_TEXT("default");</pre>
See also	<ul style="list-style-type: none">◆ “Version synchronization parameter” [<i>MobiLink Clients</i>, page 339]◆ “Script versions” [<i>MobiLink Administration Guide</i>, page 239].

Index

Symbols

#define	
UltraLite applications	221
~ULSqlca function	
UltraLite C++ Component API	229
~ULSqlcaWrap function	
UltraLite C++ Component API	233
~ULValue function	
UltraLite C++ Component API	302
16-bit signed integer UltraLite embedded SQL data type	69
32-bit signed integer UltraLite embedded SQL data type	69
4-byte floating point UltraLite embedded SQL data type	69
8-byte floating point UltraLite embedded SQL data type	69

Numbers

10054	
synchronization stream system errors (UltraLite clients)	441

A

ActiveSync	
about	140
adding to UltraLite applications	140
class names	137
MFC UltraLite applications	141
supported versions	140
ULIsSynchronizeMessage function	376
UltraLite message	221
WindowProc function	141
AddRef function	
UltraLite C++ Component API	269
AES encryption algorithm	
UltraLite embedded SQL databases	87
UltraLite Static C++ API databases	49
AfterLast function	
UltraLite C++ Component API	245
AfterLast method (ULCursor class)	

UltraLite Static C++ API	320
an_SQL_code UltraLite data type	
UltraLite Static C++ API	305
applications	
building the sample UltraLite embedded SQL application	186
building UltraLite embedded SQL	97
compiling UltraLite embedded SQL	97
deploying UltraLite on Palm Computing Platform 128	
preprocessing UltraLite embedded SQL	97
writing in UltraLite embedded SQL	180
writing UltraLite embedded SQL	61
auth_parms synchronization parameter about (UltraLite C/C++)	418
auth_status synchronization parameter about (UltraLite C/C++)	419
auth_value synchronization parameter about (UltraLite C/C++)	420
autoCommit mode	
UltraLite C++ component	32

B

BeforeFirst function	
UltraLite C++ Component API	245
BeforeFirst method (ULCursor class)	
UltraLite Static C++ API	321
benefits	
UltraLite C++ component	4
UltraLite embedded SQL	7
UltraLite Static C++ API	5
binary UltraLite embedded SQL data type	70
build processes	
single-file embedded SQL applications	100
UltraLite embedded SQL applications	97
building	
sample UltraLite embedded SQL application	186
UltraLite embedded SQL applications	97
UltraLite Static C++ API applications	58

C

C++ API	<i>see also</i> UltraLite Static C++ API
---------	--

cache_size persistent storage parameter			
about	223	communications errors	
casting		UltraLite embedded SQL	92
data types in UltraLite C++ component	28	compiler directives	
Certicom		UltraLite applications	221
unavailable on Power PC	127	UNDER_CE	224
ChangeEncryptionKey function		UNDER_PALM_OS	225
UltraLite C++ Component API	237	compiler options	
changeEncryptionKey method		UltraLite C++ component development	39
UltraLite embedded SQL	87	compilers	
UltraLite Static C++ API	49	Palm Computing Platform	114
character string UltraLite embedded SQL data type		Windows CE	132
fixed length	70	compiling	
variable length	70	UltraLite embedded SQL applications	97
checkpoint_store synchronization parameter		UltraLite Static C++ API applications	58, 59
MobiLink synchronization	421	configuring	
class names		development tools for UltraLite embedded SQL	
ActiveSync synchronization	137	102	
ClientParms registry entry		CONNECT statement	
MobiLink conduit	126	UltraLite embedded SQL	66
Close method (ULConnection class)		connecting	
UltraLite Static C++ API	306	UltraLite C++ component databases	17
Close method (ULCursor class)		Connection object	
UltraLite Static C++ API	321	UltraLite C++ component	17
Close method (ULData class)		connections	
do not use on Palm Computing Platform	331	SQLCAs in UltraLite C/C++	106
UltraLite Static C++ API	331	UltraLite embedded SQL	66
CLOSE statement		conventions	
UltraLite embedded SQL	81	documentation	xiv
CodeWarrior		CountUploadRows function	
converting projects	117	UltraLite C++ Component API	237
creating UltraLite projects	116	CountUploadRows method (ULConnection class)	
expanded mode UltraLite applications	119	UltraLite Static C++ API	307
installing UltraLite plug-in	115	CreateAndOpenDatabase function	
stationery for UltraLite	116	UltraLite C++ Component API	250
UltraLite C/C++ development	115	cursors	
using UltraLite plug-in	117	UltraLite embedded SQL	81
combining interfaces		CustDB application	
UltraLite C/C++	108	building for Palm Computing Platform	118
Commit function		building for Windows CE	134
UltraLite C++ Component API	237		
commit method		D	
UltraLite C++ component	32	data manipulation	
Commit method (ULConnection class)		dynamic SQL in UltraLite C++ component	21
UltraLite Static C++ API	306	table API in UltraLite C++ component	26
commits		data types	
UltraLite C++ component	32	accessing in UltraLite C++ component	27
		casting in UltraLite C++ component	28

UltraLite embedded SQL	68	DECL_VARCHAR macro	
UltraLite enumeration	319	UltraLite embedded SQL	69
UltraLite SQL enumeration	320	declaration section	
database files		UltraLite embedded SQL	68
changing the encryption key in UltraLite embedded SQL	87	DECLARE statement	
changing the encryption key in UltraLite Static C++ API	49	UltraLite embedded SQL	81
defragmenting in UltraLite C/C++	110	declaring	
encrypting in UltraLite embedded SQL	87	UltraLite host variables	68
encrypting in UltraLite Static C++ API	49	defragmenting	
obfuscating	221	UltraLite C/C++ databases	110
obfuscating in UltraLite embedded SQL	87	Delete function	
obfuscating in UltraLite Static C++ API	49	UltraLite C++ Component API	278
UltraLite (Windows CE)	136	Delete method (ULCursor class)	
database schemas		UltraLite Static C++ API	321
accessing in UltraLite C++ component	33	DeleteAllRows function	
upgrading in UltraLite C++ component	15	UltraLite C++ Component API	278
DatabaseManager object		DeleteAllRows method (ULTable class)	
UltraLite C++ component	17	UltraLite Static C++ API	341
databases		deleting	
connecting in UltraLite C++ component	17	rows in UltraLite C++ component	31
schema information in UltraLite C++ component	33	dependencies	
DatabaseSchema object		UltraLite embedded SQL	102
UltraLite C++ component	33	deploying	
db_fini function		UltraLite applications on Palm Computing Platform	128
do not use on the Palm Computing Platform	359	UltraLite C/C++ applications (Windows CE)	137
UltraLite embedded SQL syntax	359	UltraLite databases	209
db_init function		UltraLite on Palm Computing Platform	128
UltraLite embedded SQL syntax	360	UltraLite to Palm Computing Platform	128
db_start_database function		descUltraLite ODBC interface	
UltraLite embedded SQL syntax	361	SQLDisconnect function	396
db_stop_database function		development	
UltraLite embedded SQL syntax	362	UltraLite C++ component	13, 105
decimal UltraLite embedded SQL data type, packed	69	UltraLite Static C++	6
DECL_BINARY macro		development platforms	
UltraLite embedded SQL	69	UltraLite C++	9
DECL_DATETIME data type		development process	
UltraLite Static C++ API	305	UltraLite embedded SQL	7
DECL_DATETIME macro		development tools	
UltraLite embedded SQL	69	configuring for UltraLite embedded SQL	102
DECL_DECIMAL macro		preprocessing UltraLite embedded SQL	102
UltraLite embedded SQL	69	UltraLite embedded SQL	102
DECL_FIXCHAR macro		directives	
UltraLite embedded SQL	69	UltraLite applications	221
DECL_FIXCHAR macro		disable_concurrency synchronization parameter	
UltraLite embedded SQL	69	UltraLite C/C++	422
		DML	

UltraLite C++ component	21	ULEnableStrongEncryption (UltraLite)	211
documentation		ULEnableUserAuthentication (UltraLite)	212
conventions	xiv	ULGetDatabaseID (UltraLite)	368
SQL Anywhere Studio	xii	ULGetLastDownloadTime (UltraLite)	369
download acknowledgements		ULGetSynchResult (UltraLite)	370
send_download_ack synchronization parameter		ULGlobalAutoincUsage (UltraLite)	372
(embedded SQL)	437	ULGrantConnectTo (UltraLite)	373
send_download_ack synchronization parameter		ULHTTPStream (UltraLite)	374
(static C++ API)	437	ULHTTPStream (UltraLite)	375
download_only synchronization parameter		ULIsSynchronizeMessage (UltraLite)	376
about (UltraLite C/C++)	423	ULPalmDBStream (UltraLite)	377
download-only synchronization		ULPalmExit (UltraLite)	378
download_only synchronization parameter		ULPalmLaunch (UltraLite)	379
(UltraLite C/C++)	423	ULResetLastDownloadTime (UltraLite)	380
Drop method (ULData class)		ULRetrieveEncryptionKey (UltraLite)	381
UltraLite Static C++ API	332	ULRevokeConnectFrom (UltraLite)	382
DropDatabase function		ULSaveEncryptionKey (UltraLite)	384
UltraLite C++ Component API	250	ULSetDatabaseID (UltraLite)	385
DT_BINARY UltraLite embedded SQL data type	72	ULSetSynchInfo (UltraLite)	386
DT_LONGVARCHAR UltraLite embedded SQL		ULSocketStream (UltraLite)	387
data type	72	ULSynchronize (UltraLite)	388
dynamic SQL		eMbedded Visual C++	
adding to UltraLite embedded SQL	108	obtaining UltraLite	132
adding to UltraLite Static C++ API	108	emulator	
UltraLite C++ component development	21	Windows CE for UltraLite C/C++ applications	
E		137	
embedded SQL		encryption	
combining with UltraLite C++ Component	108	changing keys in UltraLite embedded SQL	87
cursors (UltraLite)	81	changing keys in UltraLite Static C++ API	49
fetching data (UltraLite)	80	changing UltraLite encryption keys (embedded	
UltraLite benefits	7	SQL)	364
UltraLite data access	61	ULEnableStrongEncryption function in UltraLite	
UltraLite functions	357	C/C++	211
UltraLite host variables	68	UltraLite C++ component development	36
UltraLite sample program	180	UltraLite databases using embedded SQL	87
UltraLite tutorial	178	UltraLite databases using Static C++ API	49
using in UltraLite	357	UltraLite embedded SQL databases	87
embedded SQL library functions		UltraLite Static C++ API	44
ULActiveSyncStream (UltraLite)	363	UltraLite Static C++ API databases	49
ULChangeEncryptionKey (UltraLite)	364	error checking	
ULClearEncryptionKey (UltraLite)	365	UltraLite ODBC interface	405
ULCountUploadRows (UltraLite)	366	error handling	
ULDropDatabase (UltraLite)	367	UltraLite C++ component	34
ULEnableFileDB (UltraLite)	208	UltraLite C/C++	204, 213
ULEnableGenericSchema (UltraLite)	209	errors	
ULEnablePalmRecordDB (UltraLite)	210	codes (UltraLite)	64
		communications errors in UltraLite embedded	

SQL	92	UltraLite Static C++ API	343
handling in UltraLite C++ component	34	FindPrevious function	
SQLCODE (UltraLite)	64	UltraLite C++ Component API	280
sqlcode SQLCA field (UltraLite)	64	FindPrevious method (ULTable class)	
EXEC SQL		UltraLite Static C++ API	343
UltraLite embedded SQL development	63	First function	
Execute method (generated statement class)		UltraLite C++ Component API	245
UltraLite Static C++ API	350	First method (ULCursor class)	
ExecuteQuery function		UltraLite Static C++ API	322
UltraLite C++ Component API	261	functions	
ExecuteStatement function		UltraLite embedded SQL	357
UltraLite C++ Component API	261		
expanded mode		G	
Palm OS UltraLite applications	119	generated databases	
UltraLite plug-in for CodeWarrior	118	naming in UltraLite	117
F		generated result set class	
feedback		UltraLite Static C++ API	350
documentation	xviii	generating multi-segment code	
providing	xviii	UltraLite	122
FETCH statement		Get function	
UltraLite embedded SQL	80, 81	UltraLite C++ Component API	246
fetching		Get method (generated table class)	
UltraLite embedded SQL	80	UltraLite Static C++ API	352
Finalize function		Get method (ULCursor class)	
UltraLite C++ Component API	230	UltraLite Static C++ API	322
Find function		GetBinary function	
UltraLite C++ Component API	279	UltraLite C++ Component API	292
Find method (ULTable class)		GetBinaryLength function	
UltraLite Static C++ API	341	UltraLite C++ Component API	293
find methods		GetByteChunk function	
UltraLite C++ component	28	UltraLite C++ Component API	272
find mode		GetCA function	
UltraLite C++ component	27	UltraLite C++ Component API	230
FindBegin function		GetCA method (ULConnection class)	
UltraLite C++ Component API	279	UltraLite Static C++ API	307
FindFirst function		GetCollationName function	
UltraLite C++ Component API	279	UltraLite C++ Component API	253
FindFirst method (ULTable class)		GetColumn method (generated result set class)	
UltraLite Static C++ API	341	UltraLite Static C++ API	347
FindLast function		GetColumn method (generated table class)	
UltraLite C++ Component API	280	UltraLite Static C++ API	353
FindLast method (ULTable class)		GetColumnCount function	
UltraLite Static C++ API	342	UltraLite C++ Component API	257, 266
FindNext function		GetColumnCount method (ULCursor class)	
UltraLite C++ Component API	280	UltraLite Static C++ API	323
FindNext method (ULTable class)		GetColumnDefault function	
		UltraLite C++ Component API	286

GetColumnID function			GetLastIdentity method (ULConnection class)	
UltraLite C++ Component API		266	UltraLite Static C++ API	308
GetColumnName function			GetLength function	
UltraLite C++ Component API		257, 266	UltraLite C++ Component API	272
GetColumnPrecision function			GetName function	
UltraLite C++ Component API		267	UltraLite C++ Component API	258, 288
GetColumnScale function			GetNewUUID function	
UltraLite C++ Component API		267	UltraLite C++ Component API	238
GetColumnSize function			GetOptimalIndex function	
UltraLite C++ Component API		267	UltraLite C++ Component API	288
GetColumnSize method (ULCursor class)			GetParameter function	
UltraLite Static C++ API		323	UltraLite C++ Component API	231
GetColumnSQLType function			GetParameterCount function	
UltraLite C++ Component API		267	UltraLite C++ Component API	231
GetColumnSQLType method (ULCursor class)		324	GetPlan function	
GetColumnType function			UltraLite C++ Component API	261
UltraLite C++ Component API		268	GetPrimaryKey function	
GetColumnType method (ULCursor class)		324	UltraLite C++ Component API	288
GetConnection function			GetPublicationCount function	
UltraLite C++ Component API		269	UltraLite C++ Component API	253
GetConnectionNum function			GetPublicationID function	
UltraLite C++ Component API		237	UltraLite C++ Component API	253
GetDatabaseID function			GetPublicationMask function	
UltraLite C++ Component API		237	UltraLite C++ Component API	238, 254
GetDatabaseID method (ULConnection class)			GetPublicationName function	
UltraLite Static C++ API		308	UltraLite C++ Component API	254
GetDatabaseProperty function			GetReferencedIndexName function	
UltraLite C++ Component API		237	UltraLite C++ Component API	258
GetGlobalAutoincPartitionSize function			GetReferencedTableName function	
UltraLite C++ Component API		287	UltraLite C++ Component API	258
GetID function			GetRowCount function	
UltraLite C++ Component API		258, 287	UltraLite C++ Component API	246
GetIFace function			GetRowCount method (ULTable class)	
UltraLite C++ Component API		269	UltraLite Static C++ API	344
GetIndexCount function			GetSchema function	
UltraLite C++ Component API		287	UltraLite C++ Component API	238, 262, 264, 281
GetIndexName function			GetSignature function	
UltraLite C++ Component API		287	UltraLite C++ Component API	254
GetIndexSchema function			GetSizeColumn method (generated table class)	
UltraLite C++ Component API		287	UltraLite Static C++ API	354
GetLastDownloadTime function			GetSqlca function	
UltraLite C++ Component API		238	UltraLite C++ Component API	239
GetLastDownloadTime method (ULConnection class)			GetSQLCode function	
UltraLite Static C++ API		308	UltraLite C++ Component API	231
GetLastIdentity function			GetSQLCode method (ULConnection class)	
UltraLite C++ Component API		238	UltraLite Static C++ API	309
			GetSQLCode method (ULCursor class)	

UltraLite Static C++ API	325	grantConnectTo method	
GetSQLCount function		UltraLite C++ component development	35
UltraLite C++ Component API	231	GrantConnectTo method (ULConnection class)	
GetSQLErrorOffset function		UltraLite Static C++ API	311
UltraLite C++ Component API	232		
GetState function		H	
UltraLite C++ Component API	246	handling errors	
GetStreamReader function		UltraLite C/C++	204, 213
UltraLite C++ Component API	246	HasResultSet function	
GetStreamWriter function		UltraLite C++ Component API	262
UltraLite C++ Component API	262, 281	header files	
GetString function		UltraLite Static C++ API	304
UltraLite C++ Component API	293	host variables	
GetStringChunk function		expressions in UltraLite embedded SQL	74
UltraLite C++ Component API	273	UltraLite embedded SQL	68
GetStringLength function		UltraLite scope	73
UltraLite C++ Component API	294	UltraLite usage	72
GetSuspend function		HotSync synchronization	
UltraLite C++ Component API	239, 246	Palm Computing Platform	125
GetSynchResult function		hpp file	
UltraLite C++ Component API	239	UltraLite Static C++ API	304
GetSynchResult method (ULConnection class)		HTTP synchronization	
UltraLite Static C++ API	309	Palm Computing Platform in UltraLite	127
GetTableCount function		HTTPS synchronization	
UltraLite C++ Component API	254	Palm Computing Platform in UltraLite	127
GetTableName function			
UltraLite C++ Component API	254, 258	I	
GetTableSchema function		icons	
UltraLite C++ Component API	255	used in manuals	xvi
GetUploadUnchangedRows function		ignored_rows synchronization parameter	
UltraLite C++ Component API	288	UltraLite C/C++	425
GetUtilityULValue function		import libraries	
UltraLite C++ Component API	239	UltraLite C++ component	38
global autoincrement		INCLUDE statement	
ULGlobalAutoincUsage function	372	SQLCA (UltraLite)	64
ULSetDatabaseID function (UltraLite embedded SQL)	385	InDatabase function	
UltraLite Static C++ API	311, 316	UltraLite C++ Component API	294
global database identifier		index enumeration (generated table class)	
UltraLite embedded SQL	385	UltraLite Static C++ API	355
UltraLite Static C++ API	316	indexes	
GlobalAutoincUsage function		schema information in UltraLite C++ component	
UltraLite C++ Component API	239	33	
GlobalAutoincUsage method (ULConnection class)		IndexSchema object	
UltraLite Static C++ API	311	UltraLite C++ component development	33
GrantConnectTo function		indicator variables	
UltraLite C++ Component API	239	NULL (UltraLite)	78
		UltraLite embedded SQL	78

Initialize function			
UltraLite C++ Component API	232	UltraLite C++ Component API	259
Initialize method (ULData class)		IsForeignKeyCheckOnCommit function	
UltraLite Static C++ API	332	UltraLite C++ Component API	259
InitSynchInfo function		IsForeignKeyNullable function	
UltraLite C++ Component API	240	UltraLite C++ Component API	259
InitSynchInfo method		IsNeverSynchronized function	
about	52	UltraLite C++ Component API	290
InitSynchInfo method (ULConnection class)		IsNull function	
UltraLite Static C++ API	311	UltraLite C++ Component API	247, 294
InPublication function		IsOpen method (ULConnection class)	
UltraLite C++ Component API	288	UltraLite Static C++ API	312
Insert function		IsOpen method (ULCursor class)	
UltraLite C++ Component API	281	UltraLite Static C++ API	325
Insert method (ULCursor class)		IsOpen method (ULData class)	
UltraLite Static C++ API	325	UltraLite Static C++ API	333
insert mode		IsPrimaryKey function	
UltraLite C++ component	27	UltraLite C++ Component API	259
InsertBegin function		IsUniqueIndex function	
UltraLite C++ Component API	281	UltraLite C++ Component API	259
inserting		IsUniqueKey function	
rows in UltraLite C++ component	30	UltraLite C++ Component API	259
installing			
Palm Computing Platform UltraLite	128	K	
UltraLite plug-in for CodeWarrior	115	keep_partial_download synchronization parameter	
Windows CE UltraLite	132	UltraLite C/C++	424
IsCaseSensitive function			
UltraLite C++ Component API	255	L	
IsColumnAutoinc function		last download timestamp	
UltraLite C++ Component API	289	resetting in UltraLite databases	315, 380
IsColumnCurrentDate function		ULGetLastDownloadTime function	369
UltraLite C++ Component API	289	Last function	
IsColumnCurrentTime function		UltraLite C++ Component API	247
UltraLite C++ Component API	289	Last method (ULCursor class)	
IsColumnCurrentTimestamp function		UltraLite Static C++ API	325
UltraLite C++ Component API	289	LastCodeOK function	
IsColumnDescending function		UltraLite C++ Component API	232
UltraLite C++ Component API	258	LastCodeOK method (ULConnection class)	
IsColumnGlobalAutoinc function		UltraLite Static C++ API	312
UltraLite C++ Component API	289	LastCodeOK method (ULCursor class)	
IsColumnInIndex function		UltraLite Static C++ API	326
UltraLite C++ Component API	290	LastFetchOK function	
IsColumnNewUUID function		UltraLite C++ Component API	232
UltraLite C++ Component API	290	LastFetchOK method (ULCursor class)	
IsColumnNullable function		UltraLite Static C++ API	313, 326
UltraLite C++ Component API	290	LAUNCH_SUCCESS_FIRST	
IsForeignKey function		UltraLite Static C++ API	336
		library functions	

RollbackPartialDownload (UltraLite Static C++ API)	316	ULSaveEncryptionKey (UltraLite embedded SQL)	384
ULActiveSyncStream (UltraLite embedded SQL)	363	ULSetDatabaseID (UltraLite embedded SQL)	385
ULChangeEncryptionKey (UltraLite embedded SQL)	364	ULSetSynchInfo (UltraLite embedded SQL)	386
ULClearEncryptionKey (UltraLite embedded SQL)	365	ULSocketStream (UltraLite embedded SQL)	387
ULCountUploadRows (UltraLite embedded SQL)	366	ULStoreDefragFini (UltraLite C/C++)	218
ULDropDatabase (UltraLite embedded SQL)	367	ULStoreDefragInit (UltraLite C/C++)	219
ULEnableFileDB (UltraLite C/C++)	208	ULStoreDefragStep (UltraLite C/C++)	220
ULEnableGenericSchema (UltraLite C/C++)	209	ULSynchronize (UltraLite embedded SQL)	388
ULEnablePalmRecordDB (UltraLite C/C++)	210	UltraLite embedded SQL	357
ULEnableStrongEncryption (UltraLite C/C++)	211	linking	
ULEnableUserAuthentication (UltraLite C/C++)	212	UltraLite applications in UltraLite	133
ULGetDatabaseID (UltraLite embedded SQL)	368	UltraLite C++ component applications	38
ULGetLastDownloadTime (UltraLite embedded SQL)	369	UltraLite Static C++ API applications	59
ULGetSynchResult (UltraLite embedded SQL)	370	Lookup function	
ULGlobalAutoincUsage (UltraLite embedded SQL)	372	UltraLite C++ Component API	281
ULGrantConnectTo (UltraLite embedded SQL)	373	Lookup method (ULTable class)	
ULHTTPStream (UltraLite embedded SQL)	374	UltraLite Static C++ API	344
ULHTTPStream (UltraLite embedded SQL)	375	lookup methods	
ULIsSynchronizeMessage (UltraLite embedded SQL)	376	UltraLite C++ component	28
ULPalmDBStream (UltraLite embedded SQL)	377	lookup mode	
ULPalmExit (UltraLite embedded SQL)	378	UltraLite C++ component	27
ULPalmLaunch (UltraLite embedded SQL)	379	LookupBackward function	
ULRegisterErrorCallback (UltraLite C/C++)	204, 213	UltraLite C++ Component API	282
ULRegisterSchemaUpgradeObserver (UltraLite C/C++)	206, 216	LookupBackward method (ULTable class)	
ULResetLastDownloadTime (UltraLite embedded SQL)	380	UltraLite Static C++ API	344
ULRetrieveEncryptionKey (UltraLite embedded SQL)	381	LookupBegin function	
ULRevokeConnectFrom (UltraLite embedded SQL)	382	UltraLite C++ Component API	282
ULRollbackPartialDownload (UltraLite embedded SQL)	383	LookupForward function	
		UltraLite C++ Component API	282
		LookupForward method (ULTable class)	
		UltraLite Static C++ API	345
		M	
		macros	
		UL_ENABLE_OBFUSCATION	221
		UL_ENABLE_SEGMENTS	222
		UL_ENABLE_USER_AUTH	222
		UL_OMIT_COLUMN_INFO	222
		UL_STORE_PARMS	222
		UL_SYNC_ALL	224
		UL_SYNC_ALL_PUBS	224
		UL_TEXT	224
		UL_USE_DLL	224
		UltraLite applications	221
		makefiles	
		UltraLite embedded SQL	102

Metrowerks CodeWarrior			
creating UltraLite projects	116		
MFC			
ActiveSync for UltraLite	141		
modes			
UltraLite C++ component	27		
monitoring synchronization			
observer synchronization parameter (UltraLite C/C++)	428		
moveFirst method (Table object)			
UltraLite C++ component development	23, 26		
moveNext method (Table object)			
UltraLite C++ component development	23, 26		
multi-row queries			
cursors (UltraLite)	81		
multi-segment code			
generating in UltraLite	122		
multi-threaded applications			
UltraLite C++ component	19		
UltraLite embedded SQL	66		
N			
network protocol options			
UltraLite C/C++	443		
new_password synchronization parameter			
UltraLite C/C++	426		
UltraLite UltraLite C/C++	426		
newsgroups			
technical support	xviii		
Next function			
UltraLite C++ Component API	247		
Next method (ULCursor class)			
UltraLite Static C++ API	326		
NULL			
UltraLite indicator variables	78		
UltraLite Static C++ API	322		
NULL-terminated string UltraLite embedded SQL			
data type	69		
NULL-terminated TCHAR character string			
UltraLite SQL data type	70		
NULL-terminated UNICODE character string			
UltraLite SQL data type	70		
NULL-terminated WCHAR character string			
UltraLite SQL data type	70		
NULL-terminated wide character string UltraLite			
SQL data type	70		
num_auth_parms synchronization parameter			
num_auth_parms (UltraLite C/C++)	427		
O			
obfuscating			
compiler directive	221		
UltraLite databases	221		
UltraLite embedded SQL databases	87		
UltraLite Static C++ API databases	49		
obfuscation			
UltraLite C++ component development	36		
UltraLite databases using embedded SQL	88		
UltraLite databases using Static C++ API	50		
UltraLite embedded SQL databases	87		
UltraLite Static C++ API databases	49		
objects			
generated result set	347		
generated statement	350		
generated table	352		
ULConnection	306		
ULCursor	319		
ULData	331		
ULResultSet	340		
ULTable	341		
observer synchronization parameter			
about (UltraLite C/C++)	428		
UltraLite embedded SQL example	95		
UltraLite Static C++ API example	56		
Open method (generated result set class)			
UltraLite Static C++ API	348		
Open method (generated table class)			
UltraLite Static C++ API	354		
open method (Table object)			
UltraLite C++ component development	23		
Open method (ULConnection class)			
UltraLite Static C++ API	313		
Open method (ULCursor class)			
UltraLite Static C++ API	327		
Open method (ULData class)			
UltraLite Static C++ API	333		
OPEN statement			
UltraLite embedded SQL	81		
openByIndex method (Table object)			
UltraLite C++ component development	23		
OpenConnection function			
UltraLite C++ Component API	251		
OpenTable function			
UltraLite C++ Component API	240		

-
- | | | | |
|--|---------------|----------|--|
| OpenTableWithIndex function | | | |
| UltraLite C++ Component API | 240 | | |
| operator bool function | | | |
| UltraLite C++ Component API | 300 | | |
| operator DECL_DATETIME function | | | |
| UltraLite C++ Component API | 300 | | |
| operator double function | | | |
| UltraLite C++ Component API | 300 | | |
| operator float function | | | |
| UltraLite C++ Component API | 300 | | |
| operator int function | | | |
| UltraLite C++ Component API | 300 | | |
| operator long function | | | |
| UltraLite C++ Component API | 300 | | |
| operator short function | | | |
| UltraLite C++ Component API | 300 | | |
| operator ul_s_big function | | | |
| UltraLite C++ Component API | 301 | | |
| operator ul_u_big function | | | |
| UltraLite C++ Component API | 301 | | |
| operator unsigned char function | | | |
| UltraLite C++ Component API | 301 | | |
| operator unsigned int function | | | |
| UltraLite C++ Component API | 301 | | |
| operator unsigned long function | | | |
| UltraLite C++ Component API | 301 | | |
| operator unsigned short function | | | |
| UltraLite C++ Component API | 301 | | |
| operator= function | | | |
| UltraLite C++ Component API | 301 | | |
| P | | | |
| packed decimal UltraLite embedded SQL data type | | | |
| 69 | | | |
| Palm Computing Platform | | | |
| applications in UltraLite C++ | 114 | | |
| file-based data store | 208 | | |
| HotSync synchronization in UltraLite | 125 | | |
| HTTP synchronization in UltraLite | 127 | | |
| installing UltraLite applications | 128 | | |
| platform requirements | 114 | | |
| record-based data store | 210 | | |
| security | 127 | | |
| segments | 122, 123 | | |
| TCP/IP synchronization in UltraLite | 127 | | |
| UltraLite Static C++ API | 314, 328, 337 | | |
| version 4.0 | 208, 210 | | |
| PalmExit method (ULData class) | | | |
| UltraLite Static C++ API | | 334 | |
| PalmLaunch method (ULData class) | | | |
| UltraLite Static C++ API | | 335 | |
| partial_download_retained synchronization | | | |
| parameter | | | |
| UltraLite C/C++ | | 429 | |
| password synchronization parameter | | | |
| about (UltraLite C/C++) | | 430 | |
| passwords | | | |
| authentication in UltraLite C++ component | | 35 | |
| UltraLite C/C++ synchronization | | 426 | |
| PATH environment variable | | | |
| HotSync | | 114 | |
| performance | | | |
| UltraLite cache_size parameter | | 223 | |
| permissions | | | |
| UltraLite embedded SQL | | 63 | |
| persistent storage | | | |
| cache_size parameter | | 223 | |
| Windows CE | | 136 | |
| PilotMain function | | | |
| UltraLite applications | | 121, 125 | |
| ping synchronization parameter | | | |
| about (UltraLite C/C++) | | 431 | |
| platforms | | | |
| supported in UltraLite C++ | | 9 | |
| prefix files | | | |
| about | | 117 | |
| CodeWarrior | | 123 | |
| prepared statements | | | |
| UltraLite C++ component | | 21 | |
| preparedStatement class | | | |
| UltraLite C++ component | | 21 | |
| PrepareStatement function | | | |
| UltraLite C++ Component API | | 241 | |
| preprocessing | | | |
| development tool settings for UltraLite embedded | | | |
| SQL | | 102 | |
| UltraLite embedded SQL applications | | 97 | |
| Previous function | | | |
| UltraLite C++ Component API | | 247 | |
| Previous method (ULCursor class) | | | |
| UltraLite Static C++ API | | 327 | |
| program structure | | | |
| UltraLite embedded SQL | | 63 | |
| projects | | | |

adding statements in UltraLite Static C++ API	43	resume_partial_download synchronization	
UltraLite Static C++ API	43	parameter	433
publication creation wizard		UltraLite embedded SQL	383
using in UltraLite static C++	167	UltraLite Static C++ API	316
publication synchronization parameter		result set schemas	
about (UltraLite C/C++)	432	UltraLite C++ component	24
publications		result sets	
publication synchronization parameter (UltraLite		UltraLite C++ component	24
C/C++)	432	resume_partial_download synchronization	
schema information in UltraLite C++ component		parameter	
33		UltraLite C/C++	433
PublicationSchema object		RevokeConnectFrom function	
UltraLite C++ component development	33	UltraLite C++ Component API	241
Q		RevokeConnectFrom method (ULConnection class)	
queries		UltraLite Static C++ API	315
single-row (UltraLite embedded SQL)	80	revokeConnectionFrom method	
UltraLite Static C++ API	43	UltraLite C++ component development	35
R		Rollback function	
registry		UltraLite C++ Component API	241
ClientParms registry entry	126	rollback method	
Relative function		UltraLite C++ component	32
UltraLite C++ Component API	247	Rollback method (ULConnection class)	
Relative method (ULCursor class)		UltraLite Static C++ API	316
UltraLite Static C++ API	327	RollbackPartialDownload function	
Release function		UltraLite C++ Component API	241
UltraLite C++ Component API	270	UltraLite Static C++ API	316
Reopen method		rollbacks	
UltraLite C/C++	120	UltraLite C++ component	32
Reopen method (ULConnection class)		rows	
deprecated function	314	accessing current in UltraLite C++ component	27
Reopen method (ULCursor class)		accessing in UltraLite C++ Component tutorial	158
UltraLite Static C++ API	328	runtime libraries	
Reopen method (ULData class)		UltraLite C++ component	38
deprecated function	337	runtime library	
ResetLastDownloadTime function		Windows CE	133, 224
UltraLite C++ Component API	241	S	
ResetLastDownloadTime method (ULConnection		sample application	
class)		building for Palm Computing Platform	118
UltraLite Static C++ API	315	building for Windows CE	134
restartable downloads		saving state	
keep_partial_download synchronization		UltraLite on Palm OS	120
parameter	424	schema changes	
partial_download_retained synchronization		UltraLite C/C++ callback function	206
parameter	429	schema files	
		creating in UltraLite C++ component	15

-
- upgrading in UltraLite C++ component 15
 - schema upgrades
 - UltraLite C/C++ callback function 206
 - UltraLite databases 209
 - schemas
 - accessing in UltraLite C++ component 33
 - upgrading in UltraLite C++ component 15
 - script versions
 - version synchronization parameter (UltraLite C/C++) 448
 - scrolling
 - UltraLite C++ component 26
 - security
 - changing the encryption key in UltraLite embedded SQL 87
 - changing the encryption key in UltraLite Static C++ API 49
 - database obfuscation 221
 - obfuscation in UltraLite embedded SQL 87
 - obfuscation in UltraLite Static C++ API 49
 - security synchronization parameter (UltraLite C/C++) 434
 - security_parms synchronization parameter (UltraLite C/C++) 435
 - send_column_names synchronization parameter (UltraLite C/C++) 436
 - UltraLite C/C++ applications 127
 - UltraLite database encryption 49, 87
 - unavailable on Power PC 127
 - security synchronization parameter
 - about (UltraLite C/C++) 434
 - security_parms synchronization parameter
 - about (UltraLite C/C++) 435
 - segments
 - about 122, 123
 - explicitly assigning 123
 - generating multi-segment code 122
 - Palm Computing Platform 122–124, 222
 - reducing UltraLite requirements 222
 - user-defined code 124
 - SELECT statement
 - single row (UltraLite embedded SQL) 80
 - UltraLite C++ component development 23
 - send_column_names synchronization parameter
 - about (UltraLite C/C++) 436
 - send_download_ack synchronization parameter
 - about (embedded SQL) 437
 - about (static C++ API) 437
 - SET CONNECTION statement
 - multiple connections in UltraLite embedded SQL 67
 - Set function
 - UltraLite C++ Component API 283
 - Set method (ULCursor class)
 - UltraLite Static C++ API 328
 - SetBinary function
 - UltraLite C++ Component API 294
 - SetColumn method (generated result set)
 - UltraLite Static C++ API 348
 - SetColumn method (generated table class)
 - UltraLite Static C++ API 355
 - SetColumnNull method (ULCursor class)
 - UltraLite Static C++ API 329
 - SetDatabaseID function
 - UltraLite C++ Component API 241
 - SetDatabaseID method (ULConnection class)
 - UltraLite Static C++ API 316
 - SetDefault function
 - UltraLite C++ Component API 283
 - SetNull function
 - UltraLite C++ Component API 283
 - SetNull method (generated statement class)
 - UltraLite Static C++ API 350
 - SetNullColumn method (generated result set class)
 - UltraLite Static C++ API 349
 - SetNullColumn method (generated table class)
 - UltraLite Static C++ API 355
 - SetParameter function
 - UltraLite C++ Component API 262
 - SetParameter method (generated statement class)
 - UltraLite Static C++ API 351
 - SetParameter method (ULResultSet class)
 - UltraLite Static C++ API 340, 351
 - SetParameterNull function
 - UltraLite C++ Component API 262
 - SetReadPosition function
 - UltraLite C++ Component API 274
 - SetString function
 - UltraLite C++ Component API 295
 - SetSuspend function
 - UltraLite C++ Component API 242, 247
 - SetSynchInfo function
 - UltraLite C++ Component API 242
 - Shutdown function

UltraLite C++ Component API	242, 251	SQLExecute function	
SQL Anywhere Studio		UltraLite ODBC interface	399
documentation	xii	SQLFetch function	
SQL Communications Area		UltraLite ODBC interface	400
UltraLite C/C++	106	SQLFetchScroll function	
UltraLite embedded SQL	64	UltraLite ODBC interface	401
SQL preprocessor		SQLFreeHandle function	
UltraLite embedded SQL applications	97	UltraLite ODBC interface	402
UltraLite example	100	SQLGetCursorName function	
sqlaid SQLCA field		UltraLite ODBC interface	403
UltraLite embedded SQL	64	SQLGetData function	
SQLAllocHandle function		UltraLite ODBC interface	404
UltraLite ODBC interface	391	SQLGetDiagRec function	
SQLBindCol function		UltraLite ODBC interface	405
UltraLite ODBC interface	392	SQLGetInfo function	
SQLBindParameter function		UltraLite ODBC interface	406
UltraLite ODBC interface	393	SQLNumResultCols function	
SQLCA		UltraLite ODBC interface	407
fields (UltraLite)	64	sqlpp utility	
multiple (UltraLite embedded SQL)	66	UltraLite embedded SQL applications	97
UltraLite C/C++	106	SQLPrepare function	
UltraLite embedded SQL	64	UltraLite ODBC interface	408
sqlcabc SQLCA field		SQLRowCount function	
UltraLite embedded SQL	64	UltraLite ODBC interface	409
SQLCODE		SQLSetConnectionName function	
UltraLite C++ component error handling	34	UltraLite ODBC interface	411
UltraLite C/C++ error handling	204	SQLSetCursorName function	
sqlcode SQLCA field		UltraLite ODBC interface	410
UltraLite embedded SQL	64	SQLSetSuspend function	
SQLConnect function		UltraLite ODBC interface	412
UltraLite ODBC interface	394	sqlstate SQLCA field	
SQLDescribeCol function		UltraLite embedded SQL	65
UltraLite ODBC interface	395	SQLSynchronize function	
SQLDisconnect function		UltraLite ODBC interface	413
UltraLite ODBC interface	396	sqlwarn SQLCA field	
SQLEndTran function		UltraLite embedded SQL	65
UltraLite ODBC interface	397	StartDatabase method (ULData class)	
sqlerrd SQLCA field		UltraLite Static C++ API	338
UltraLite embedded SQL	65	StartSynchronizationDelete function	
sqlerrmc SQLCA field		UltraLite C++ Component API	242
UltraLite embedded SQL	65	StartSynchronizationDelete method (ULConnection class)	
sqlerrml SQLCA field		UltraLite Static C++ API	317
UltraLite embedded SQL	64	StopDatabase method (ULData class)	
sqlerrp SQLCA field		UltraLite Static C++ API	338
UltraLite embedded SQL	65	StopSynchronizationDelete function	
SQLExecDirect function		UltraLite C++ Component API	242
UltraLite ODBC interface	398		

-
- StopSynchronizationDelete method (ULConnection class)
 - UltraLite Static C++ API 317
 - stream definition functions
 - GetSynchResult method 309
 - ULActiveSyncStream (UltraLite embedded SQL) 363
 - ULHTTPSSStream (UltraLite embedded SQL) 374
 - ULHTTPStream (UltraLite) 375
 - ULPalmDBStream (embedded SQL) 377
 - ULSetDatabaseID (embedded SQL) 385
 - ULSocketStream (UltraLite embedded SQL) 387
 - stream synchronization parameter
 - about (embedded SQL) 438
 - about (static C++ API) 438
 - stream_error synchronization parameter
 - about (UltraLite C/C++) 440
 - ul_stream_error structure (UltraLite C/C++) 440
 - stream_parms synchronization parameter
 - about (UltraLite C/C++) 443
 - string UltraLite embedded SQL data type
 - fixed length 70
 - NULL-terminated 69
 - variable length 70
 - StringCompare function
 - UltraLite C++ Component API 295
 - strings
 - UL_TEXT macro 224
 - strong encryption
 - UltraLite databases 211
 - UltraLite embedded SQL 87
 - UltraLite Static C++ API 49
 - StrToUUID function
 - UltraLite C++ Component API 243
 - support
 - newsgroups xviii
 - supported platforms
 - UltraLite C++ 9
 - synchronization
 - adding to UltraLite embedded SQL applications 89
 - adding to UltraLite Static C++ API applications 51
 - canceling in UltraLite embedded SQL 92
 - canceling in UltraLite Static C++ API 53
 - checkpoint_store parameter in UltraLite C/C++ 421
 - committing changes in UltraLite embedded SQL 91
 - committing changes in UltraLite Static C++ API 53
 - disable_concurrency parameter in UltraLite C/C++ 422
 - HotSync in UltraLite 125
 - HTTP in UltraLite C++ 127
 - ignored_rows parameter in UltraLite C/C++ 425
 - initial in UltraLite embedded SQL 91
 - initial in UltraLite Static C++ API 53
 - invoking in UltraLite embedded SQL 90
 - invoking in UltraLite Static C++ API 52
 - monitoring in UltraLite embedded SQL 92
 - monitoring in UltraLite Static C++ API 53
 - Palm Computing Platform in UltraLite 125
 - TCP/IP in UltraLite 127
 - troubleshooting in UltraLite C++ 309
 - troubleshooting in UltraLite embedded SQL 370
 - UltraLite C++ Component tutorial 160
 - UltraLite embedded SQL 89
 - UltraLite embedded SQL example 90
 - UltraLite embedded SQL tutorial 187
 - UltraLite ODBC interface 413
 - UltraLite Static C++ API 51, 317
 - UltraLite Static C++ API example 52
 - UltraLite static C++ tutorial 174
 - Windows CE for UltraLite 140
 - synchronization errors
 - communications errors in UltraLite embedded SQL 92
 - synchronization functions
 - ULSetSynchInfo (UltraLite embedded SQL) 386
 - synchronization parameters
 - auth_parms (UltraLite C/C++) 418
 - auth_status (UltraLite C/C++) 419
 - auth_value (UltraLite C/C++) 420
 - download_only (UltraLite C/C++) 423
 - new_password (UltraLite C/C++) 426
 - new_password in UltraLite C/C++ 426
 - num_auth_parms (UltraLite C/C++) 427
 - observer (UltraLite C/C++) 428
 - password (UltraLite C/C++) 430
 - ping (UltraLite C/C++) 431
 - publication (UltraLite C/C++) 432
 - security (UltraLite C/C++) 434
 - security_parms (UltraLite C/C++) 435

send_column_names (UltraLite C/C++)	436	UltraLite C++	9
send_download_ack (embedded SQL)	437	TCP/IP synchronization	
send_download_ack (static C++ API)	437	Palm Computing Platform in UltraLite	127
stream (embedded SQL)	438	technical support	
stream (static C++ API)	438	newsgroups	xviii
stream_error (UltraLite C/C++)	440	threads	
stream_parms (UltraLite C/C++)	443	multi-threaded UltraLite C++ component	
upload_ok (UltraLite C/C++)	444	applications	19
upload_only (UltraLite C/C++)	445	UltraLite embedded SQL	66
user_data (UltraLite C/C++)	446	timestamp structure UltraLite embedded SQL data	
user_name (UltraLite C/C++)	447	type	71
version (UltraLite C/C++)	448	tips	
synchronization status		UltraLite development	53, 91
GetSynchResult method	309	transaction processing	
ULGetSynchResult function	370	UltraLite C++ component	32
synchronization streams		transactions	
stream synchronization parameter (embedded SQL)	438	UltraLite C++ component	32
stream synchronization parameter (static C++ API)	438	transport-layer security	
stream_error synchronization parameter (UltraLite C/C++)	440	unavailable on Power PC	127
stream_parms synchronization parameter (UltraLite C/C++)	443	troubleshooting	
ULActiveSyncStream (UltraLite C/C++)	438	committing changes before synchronization in UltraLite embedded SQL	91
ULHTTPStream (UltraLite C/C++)	438	committing changes before synchronization in UltraLite Static C++ API	53
ULSocketStream (UltraLite C/C++)	438	ping synchronization parameter (UltraLite C/C++)	431
Synchronize function		previous synchronization	309, 370
UltraLite C++ Component API	243	UltraLite development	53, 91
Synchronize method (ULConnection class)		upload_ok synchronization parameter (UltraLite C/C++)	444
UltraLite Static C++ API	317	TruncateTable function	
sysAppLaunchCmdNormalLaunch		UltraLite C++ Component API	283
UltraLite applications	121, 125	truncation	
system_error_code values		FETCH (UltraLite)	79
synchronization stream errors (UltraLite clients)	441	tutorials	
		UltraLite C++ Component	147
		UltraLite embedded SQL	178
		UltraLite Static C++	164
T		U	
Table object		UL_AS_SYNCHRONIZE macro	
UltraLite C++ component development	23	ActiveSync UltraLite messages	221
tables		UL_AUTH_STATUS_EXPIRED auth_status value	
defining in UltraLite Static C++ API	43	about	419
schema information in UltraLite C++ component	33	UL_AUTH_STATUS_IN_USE auth_status value	
TableSchema object		about	419
UltraLite C++ component development	33	UL_AUTH_STATUS_INVALID auth_status value	
target platforms			

-
- about 419
 - UL_AUTH_STATUS_UNKNOWN auth_status value 419
 - UL_AUTH_STATUS_VALID auth_status value 419
 - UL_AUTH_STATUS_VALID_BUT_EXPIRES_SOON auth_status value 419
 - ul_binary data UltraLite type
 - UltraLite Static C++ API 305
 - ul_char data UltraLite type
 - UltraLite Static C++ API 305
 - ul_column_num UltraLite data type
 - UltraLite Static C++ API 305
 - UL_ENABLE_OBFUSCATION macro
 - about 221
 - UL_ENABLE_SEGMENTS macro
 - about 222
 - UL_ENABLE_USER_AUTH macro
 - about 222
 - ul_fetch_offset UltraLite data type
 - UltraLite Static C++ API 305
 - ul_length UltraLite data type
 - UltraLite Static C++ API 305
 - UL_NULL 305
 - UL_OMIT_COLUMN_INFO macro
 - about 222
 - UL_STORE_PARMS macro
 - about 222
 - ul_stream_error structure
 - about (UltraLite C/C++) 440
 - UL_SYNC_ALL macro
 - about 224
 - publication mask 432
 - UL_SYNC_ALL_PUBS macro
 - about 224
 - publication mask 432
 - ul_synch_info structure
 - about 52, 90
 - UltraLite C/C++ 417
 - ul_synch_status structure
 - UltraLite embedded SQL 93
 - UltraLite Static C++ API 54
 - UL_TEXT macro
 - about 224
 - UL_USE_DLL macro
 - about 224
 - ULActiveSyncStream function
 - setting synchronization stream (UltraLite C/C++) 438
 - UltraLite embedded SQL syntax 363
 - Windows CE 140
 - ulapi.h
 - UltraLite Static C++ API 304
 - ULChangeEncryptionKey function
 - UltraLite embedded SQL syntax 364
 - using in UltraLite embedded SQL 87
 - using in UltraLite Static C++ API 49
 - ULClearEncryptionKey function
 - UltraLite embedded SQL syntax 365
 - ULConduitStream function
 - setting synchronization stream (UltraLite C/C++) 438
 - ULConnection class
 - ResetLastDownloadTime method 315
 - RevokeConnectFrom method 315
 - UltraLite Static C++ API 45, 306
 - ULCountUploadRows function
 - UltraLite embedded SQL syntax 366
 - ULCursor class
 - UltraLite Static C++ API 319, 352
 - ULData class
 - UltraLite Static C++ API 45, 331
 - ULDropDatabase function
 - UltraLite embedded SQL syntax 367
 - ULEnableFileDB function
 - UltraLite C/C++ syntax 208
 - UltraLite Static C++ API 44, 304
 - ULEnableGenericSchema function
 - deprecated feature 209
 - UltraLite C/C++ syntax 209
 - ULEnablePalmRecordDB function
 - UltraLite C/C++ syntax 210
 - UltraLite Static C++ API 44, 304
 - ULEnableStrongEncryption function
 - UltraLite C/C++ syntax 211
 - UltraLite Static C++ API 44, 304
 - ULEnableUserAuthentication function
 - about 47, 85
 - UltraLite C/C++ syntax 212
 - UltraLite Static C++ API 44, 304
 - ulgen utility

UltraLite Static C++ API	58	ULRetrieveEncryptionKey function	
ULGetDatabaseID function		UltraLite embedded SQL syntax	381
UltraLite embedded SQL syntax	368	ULRevokeConnectFrom function	
ULGetLastDownloadTime function		UltraLite embedded SQL syntax	382
UltraLite embedded SQL syntax	369	ULRollbackPartialDownload function	
ULGetSynchResult function		UltraLite embedded SQL	383
UltraLite embedded SQL syntax	370	ulrt9.dll	
ULGlobalAutoincUsage function		linking UltraLite C++ component applications	39
UltraLite embedded SQL syntax	372	ulrtcw9.dll	
ulglobal.h		linking UltraLite applications	39
ul_synch_info structure (UltraLite C/C++)	417	ulrtw9.dll	
UltraLite Static C++ API	304	linking UltraLite applications	39
ULGrantConnectTo function		ULSaveEncryptionKey function	
UltraLite embedded SQL syntax	373	UltraLite embedded SQL syntax	384
ULHTTPSSStream function		ULSecureCerticomTLSStream function	
setting synchronization stream (UltraLite C/C++)		about (UltraLite C/C++)	434
438		UltraLite plug-in for CodeWarrior	118
UltraLite embedded SQL syntax	374	ULSecureRSATLSStream function	
Windows CE	143	about (UltraLite C/C++)	434
ULHTTPStream function		UltraLite plug-in for CodeWarrior	118
setting synchronization stream (UltraLite C/C++)		ULSetDatabaseID function	
438		UltraLite embedded SQL syntax	385
UltraLite embedded SQL syntax	375	ULSetSynchInfo function	
Windows CE	143	UltraLite embedded SQL syntax	386
ULInitDatabaseManager function		using	125
UltraLite C++ component	18	ULSocketStream function	
ULInitSynchInfo function		setting synchronization stream (UltraLite C/C++)	
about	90	438	
ULIsSynchronizeMessage function		UltraLite embedded SQL syntax	387
ActiveSync	140	Windows CE	143
UltraLite embedded SQL syntax	376	ULSqlca class	
ULPalmDBStream function		UltraLite C++ Component API	229
UltraLite embedded SQL syntax	377	ULSqlca function	
ULPalmExit function		UltraLite C++ Component API	229
deprecated function	378	ULSqlcaBase class	
ULPalmLaunch function		UltraLite C++ Component API	230
deprecated function	379	ULSqlcaWrap class	
ULRegisterErrorCallback function		UltraLite C++ Component API	233
UltraLite C/C++ syntax	204, 213	ULSqlcaWrap function	
UltraLite Static C++ API	304	UltraLite C++ Component API	233
ULRegisterSchemaUpgradeObserver function		ULStoreDefragFini function	
UltraLite C/C++ syntax	206, 216	UltraLite C/C++ syntax	218
UltraLite Static C++ API	304	ULStoreDefragInit function	
ULResetLastDownloadTime function		UltraLite C/C++ syntax	219
UltraLite embedded SQL syntax	380	ULStoreDefragStep function	
ULResultSet class		UltraLite C/C++ syntax	220
UltraLite Static C++ API	340	ULSynchronize function	

serial port on Palm Computing Platform	127	SQLExecute function	399
syntax	388	SQLFetch function	400
UltraLite embedded SQL tutorial	187	SQLFetchScroll function	401
ULTable class		SQLFreeHandle function	402
UltraLite Static C++ API	341	SQLGetCursorName function	403
ULTable objects		SQLGetData function	404
reopening	120	SQLGetDiagRec function	405
UltraLite		SQLGetInfo function	406
Static C++ API	304	SQLNumResultCols function	407
static development	42	SQLPrepare function	408
UltraLite C++		SQLRowCount function	409
supported platforms	9	SQLSetConnectionName function	411
UltraLite C++ API		SQLSetCursorName function	410
obfuscating UltraLite databases	88	SQLSetSuspend function	412
UltraLite C++ Component		SQLSynchronize function	413
tutorials	147	UltraLite plug-in for CodeWarrior	
UltraLite C++ component		converting projects	117
benefits	4	creating projects	116
creating schema files	15	installing	115
data manipulation with dynamic SQL	21	using	117
data manipulation with Table API	26	UltraLite projects	
development	13, 105	CodeWarrior	116
encryption	36	UltraLite runtime libraries	
upgrading database schemas	15	UltraLite C++ component	38
UltraLite C/C++		UltraLite runtime library	
about	3	deploying	137
architecture	10	UltraLite Static C++ API	
combining interfaces	108	adding statements to projects	43
Palm OS	120	benefits	5
UltraLite databases		class hierarchy	304
deploying on Palm Computing Platform	128	compiling applications	59
encrypting in embedded SQL	87	defining projects	43
encrypting in Static C++ API	49	defining tables	43
Windows CE	136	development	6, 42
UltraLite embedded SQL		generating classes	58
authorization	63	header files	304
synchronization	89	linking applications	59
UltraLite namespace		obfuscating UltraLite databases	50
UltraLite C++ component	14	Palm Computing Platform	314, 328, 337
UltraLite ODBC interface		query classes	46
SQLAllocHandle function	391	Reopen methods	120
SQLBindCol function	392	synchronization	51
SQLBindParameter function	393	table classes	46
SQLConnect function	394	tutorial	164
SQLDescribeCol function	395	UltraLite_ classes	
SQLEndTran function	397	using the UltraLite namespace	14
SQLExecDirect function	398	UltraLite_Connection class	

UltraLite C++ Component API	234	ULValue function	
UltraLite_Connection_iface class		UltraLite C++ Component API	296–299
UltraLite_Cursor_iface class	236	UNDER_CE compiler directive	
UltraLite C++ Component API	245	about	224
UltraLite_DatabaseManager class		UNDER_PALM_OS compiler directive	
UltraLite C++ Component API	249	about	225
UltraLite_DatabaseManager_iface class		Update function	
UltraLite C++ Component API	250	UltraLite C++ Component API	283
UltraLite_DatabaseSchema class		Update method (ULCursor class)	
UltraLite C++ Component API	252	UltraLite Static C++ API	329
UltraLite_DatabaseSchema_iface class		update mode	
UltraLite C++ Component API	253	UltraLite C++ component	27
UltraLite_IndexSchema class		UpdateBegin function	
UltraLite C++ Component API	256	UltraLite C++ Component API	284
UltraLite_IndexSchema_iface class		updating	
UltraLite C++ Component API	257	rows in UltraLite C++ component	30
UltraLite_PreparedStatement class		UpgradeSchemaFromFile function	
UltraLite C++ Component API	260	UltraLite C++ Component API	244
UltraLite_PreparedStatement_iface class		UpgradeSchemaFromFile method	
UltraLite C++ Component API	261	UltraLite C++ component development	15
UltraLite_ResultSet class		upgrading	
UltraLite C++ Component API	263	database schemas in UltraLite C++ component	15
UltraLite_ResultSet_iface class		UltraLite databases	209
UltraLite C++ Component API	264	upload only synchronization	
UltraLite_ResultSetSchema class		upload_only synchronization parameter	
UltraLite C++ Component API	265	(UltraLite C/C++)	445
UltraLite_RowSchema_iface class		upload_ok synchronization parameter	
UltraLite C++ Component API	266	about (UltraLite C/C++)	444
UltraLite_SQLObject_iface class		upload_only synchronization parameter	
UltraLite C++ Component API	269	about (UltraLite C/C++)	445
UltraLite_StreamReader class		user authentication	
UltraLite C++ Component API	271	auth_parms synchronization parameter (UltraLite	
UltraLite_StreamReader_iface class		C/C++)	418
UltraLite C++ Component API	272	auth_status synchronization parameter (UltraLite	
UltraLite_StreamWriter class		C/C++)	419
UltraLite C++ Component API	275	auth_value synchronization parameter (UltraLite	
UltraLite_Table class		C/C++)	420
UltraLite C++ Component API	276	compiler directive for UltraLite	222
UltraLite_Table_iface class		embedded SQL UltraLite applications	85, 311,
UltraLite C++ Component API	278	373, 382	
UltraLite_TableSchema class		new_password synchronization parameter in	
UltraLite C++ Component API	285	UltraLite UltraLite C/C++	426
UltraLite_TableSchema_iface class		password synchronization parameter (UltraLite	
UltraLite C++ Component API	286	C/C++)	430
ULValue class		UltraLite C++ component development	35
UltraLite C++ Component API	291	UltraLite C/C++ applications	212
		UltraLite databases	212, 311, 315, 373

UltraLite embedded SQL	382
UltraLite Static C++ API	44
UltraLite Static C++ API applications	47
user_name synchronization parameter (UltraLite C/C++)	447
user_data synchronization parameter about (UltraLite C/C++)	446
user_name synchronization parameter about (UltraLite C/C++)	447
users authentication in the UltraLite C++ component	35
usm files UltraLite C++ component	15
UUIDToStr function UltraLite C++ Component API	243, 244
V	
values accessing in UltraLite C++ component	27
version synchronization parameter about (UltraLite C/C++)	448
Visual C++ Windows CE development	132
W	
WindowProc function ActiveSync	141, 376
Windows CE development for	132
platform requirements	132
synchronization for UltraLite	140
winsock.lib Windows CE applications	132
wizards publication creation in UltraLite static C++	167