



UniVerse

UCI Developer's Guide

Version 10.3
February, 2009

IBM Corporation
555 Bailey Avenue
San Jose, CA 95141

Licensed Materials – Property of IBM

© Copyright International Business Machines Corporation 2008, 2009. All rights reserved.

AIX, DB2, DB2 Universal Database, Distributed Relational Database Architecture, NUMA-Q, OS/2, OS/390, and OS/400, IBM Informix®, C-ISAM®, Foundation.2000™, IBM Informix® 4GL, IBM Informix® DataBlade® module, Client SDK™, Cloudscape™, Cloudsync™, IBM Informix® Connect, IBM Informix® Driver for JDBC, Dynamic Connect™, IBM Informix® Dynamic Scalable Architecture™ (DSA), IBM Informix® Dynamic Server™, IBM Informix® Enterprise Gateway Manager (Enterprise Gateway Manager), IBM Informix® Extended Parallel Server™, i.Financial Services™, J/Foundation™, MaxConnect™, Object Translator™, Red Brick® Decision Server™, IBM Informix® SE, IBM Informix® SQL, InformiXML™, RedBack®, SystemBuilder™, U2™, UniData®, UniVerse®, wIntegrate® are trademarks or registered trademarks of International Business Machines Corporation.

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries.

Windows, Windows NT, and Excel are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries.

UNIX is a registered trademark in the United States and other countries licensed exclusively through X/Open Company Limited.

Other company, product, and service names used in this publication may be trademarks or service marks of others.

This product includes cryptographic software written by Eric Young (eay@cryptosoft.com).

This product includes software written by Tim Hudson (tjh@cryptosoft.com).

Documentation Team: Claire Gustafson, Shelley Thompson, Anne Waite

US GOVERNMENT USERS RESTRICTED RIGHTS

Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Table of Contents

Preface

Organization of This Manual	viii
Documentation Conventions.	ix
Hungarian Naming Conventions	xi
Help	xii
UniVerse Documentation.	xiii
Related Documentation	xvi
API Documentation	xvii

Chapter 1

Introduction

What Is an SQL Call Interface?.	1-3
SQL Call Interface Versus Embedded SQL	1-3
Advantages of Call Interfaces.	1-4
Language Support	1-5
Operating Platforms	1-6
Compliance with the ODBC 2.0 Standard	1-7
Requirements for UCI Applications	1-8

Chapter 2

Getting Started

Installing UCI	2-3
On UNIX Systems	2-3
On Windows Systems	2-4
Version Compatibility	2-4
Creating and Running the Sample Application	2-6
Creating and Running Client Programs	2-8
UCI Administration	2-10
Maintaining the UCI Configuration File	2-10
Administering the UniRPC	2-10

Chapter 3

Configuring UCI

Configuring a Database Server for UCI	3-3
UniRPC	3-3
UniVerse NLS	3-4
Configuring a Client System for UCI	3-5
Configuration Parameters.	3-5
Editing the UCI Configuration File	3-8
Changing UCI Configuration File Parameters	3-10
Configuring UCI for an NLS-Enabled UniVerse Server	3-12

Chapter 4

Developing UCI Applications

Writing a UCI Application Program	4-3
Initializing Resources	4-4
Allocating the Environment	4-4
Allocating the Connection	4-5
Connecting to the Server	4-5
Allocating Statement Handles	4-9
Processing SQL Statements	4-10
Transaction Modes.	4-10
Function Calls	4-11
Executing an SQL Statement	4-11
Processing Output from SQL Statements	4-15
Checking for Errors	4-17
Freeing the SQL Statement Environment	4-18
Terminating the Connection	4-19
Transaction Processing	4-20
Nested Transactions	4-20
Transaction Isolation Levels	4-22
Handling Multivalued Columns	4-23
Setting the Data Model Mode	4-23
Dynamic Normalization and Associations	4-25

Chapter 5

Calling and Executing UniVerse Procedures

What Can You Call as a UniVerse Procedure?	5-3
Processing UniVerse Procedure Results	5-5
Print Result Set	5-5
Multicolumn Result Set	5-6
Affected-Row Count	5-6
Output Parameter Values	5-6
Processing Errors from UniVerse Procedures	5-7

Chapter 6

How to Write a UniVerse Procedure

Using UniVerse Paragraphs, Commands, and Procs as Procedures	6-3
Writing UniVerse BASIC Procedures	6-4
Parameters Used by a UniVerse BASIC Procedure	6-4
SQL Results Generated by a UniVerse BASIC Procedure	6-5
Using @HSTMT in a UniVerse BASIC Procedure to Generate SQL Results 6-7	
Using the @TMP File in a UniVerse BASIC Procedure.	6-9
Errors Generated by a UniVerse BASIC Procedure	6-12
Restrictions in UniVerse BASIC Procedures	6-15
Fetching Rows and Closing @HSTMT Within a Procedure	6-15
Hints for Debugging a Procedure.	6-16

Chapter 7

Data Types

Data Types and Data Type Coercion	7-3
C Data Types Supported	7-3
SQL Data Types Supported	7-9
Data Type Coercion	7-10

Chapter 8

UCI Functions

Function Call Summary	8-4
Variables	8-5
Search Patterns	8-7
Return Values	8-8
Error Codes	8-8
Use of Hungarian Naming Conventions	8-8
Functions	8-10
SQLAllocConnect	8-11
SQLAllocEnv	8-13
SQLAllocStmt	8-15
SQLBindCol.	8-17
SQLBindMvCol.	8-22
SQLBindMvParameter	8-25
SQLBindParameter	8-27
SQLCancel	8-32
SQLColAttributes	8-34
SQLColumns	8-41
SQLConnect.	8-45
SQLDataSources	8-49
SQLDescribeCol	8-52

SQLDisconnect	8-56
SQLError	8-58
SQLExecDirect	8-62
SQLExecute	8-66
SQLFetch	8-69
SQLFreeConnect	8-73
SQLFreeEnv	8-75
SQLFreeMem	8-77
SQLFreeStmt	8-78
SQLGetData	8-81
SQLGetFunctions	8-85
SQLGetInfo	8-89
SQLNumParams	8-98
SQLNumResultCols	8-100
SQLParamOptions	8-102
SQLPrepare	8-105
SQLRowCount	8-109
SQLSetConnectOption	8-111
SQLSetParam	8-117
SQLTables	8-120
SQLTransact	8-125
SQLUseCfgFile	8-129

Appendix A Error Codes

SQLSTATE Error Codes	A-2
UniVerse SQL Error Codes.	A-6
UniRPC Error Codes.	A-12

Appendix B The UCI Sample Program

Preface

This manual describes how to use UCI (Uni Call Interface). UCI is a C-language application programming interface that lets application developers create client programs that use SQL functions to access data in UniVerse and UniData databases. You should have a working knowledge of UniVerse or UniData, C, and SQL, and have an understanding of client/server protocols.

Organization of This Manual

This manual contains the following:

Chapter 1, “[Introduction](#),” describes UCI, its relationship to the ODBC 2.0 standard, and what you need to run it.

Chapter 2, “[Getting Started](#),” tells how to install UCI, how to run the sample application program, and how to compile, link, install, and execute a client application program that uses UCI.

Chapter 3, “[Configuring UCI](#),” tells how to modify the configuration file (*uci.config*) that fine-tunes the UCI process.

Chapter 4, “[Developing UCI Applications](#),” explains how to develop a UCI application.

Chapter 5, “[Calling and Executing UniVerse Procedures](#),” describes how to call and execute procedures stored on a UniVerse data source.

Chapter 6, “[How to Write a UniVerse Procedure](#),” describes how to write a UniVerse procedure.

Chapter 7, “[Data Types](#),” is the technical reference for data types.

Chapter 8, “[UCI Functions](#),” is the technical reference for UCI function calls.

Appendix A, “[Error Codes](#),” lists the SQLSTATE return codes and their meaning.

Appendix B, “[The UCI Sample Program](#),” contains *ucisample.c*, an annotated UCI client program.

The [Glossary](#) defines terms used in this manual.

Documentation Conventions

This manual uses the following conventions:

Convention	Usage
Bold	In syntax, bold indicates commands, function names, and options. In text, bold indicates keys to press, function names, menu selections, and MS-DOS commands.
UPPERCASE	In syntax, uppercase indicates database commands, keywords, and options; UniVerse BASIC statements and functions; and SQL statements and keywords. In text, uppercase also indicates database identifiers such as file names, account names, schema names, and Windows file names and paths.
<i>Italic</i>	In syntax, italic indicates information that you supply. In text, italic also indicates UNIX commands and options, file names, and paths.
Courier	Courier indicates examples of source code and system output.
Courier Bold	In examples, courier bold indicates characters the user types or keys the user presses (for example, <Return>).
[]	Brackets enclose optional items. Do not type the brackets unless indicated.
{ }	Braces enclose nonoptional items from which you must select at least one. Do not type the braces.
itemA itemB	A vertical bar separating items indicates that you can choose only one item. Do not type the vertical bar.
...	Three periods indicate that more of the same type of item can optionally follow.
I	Item mark. For example, the item mark (I) in the following string delimits elements 1 and 2, and elements 3 and 4: 1I2F3I4V5
F	Field mark. For example, the field mark (F) in the following string delimits elements FLD1 and VAL1: FLD1FVAL1VSUBV1SSUBV2

Documentation Conventions

Convention	Usage
V	Value mark. For example, the value mark (V) in the following string delimits elements VAL1 and SUBV1: FLD1FVAL1VSUBV1SSUBV2
S	Subvalue mark. For example, the subvalue mark (S) in the following string delimits elements SUBV1 and SUBV2: FLD1FVAL1VSUBV1SSUBV2
T	Text mark. For example, the text mark (T) in the following string delimits elements 4 and 5: 1F2S3V4T5

Documentation Conventions (Continued)

The following conventions are also used:

- Syntax definitions and examples are indented for ease in reading.
- All punctuation marks included in the syntax—for example, commas, parentheses, or quotation marks—are required unless otherwise indicated.
- Syntax lines that do not fit on one line in this manual are continued on subsequent lines. The continuation lines are indented. When entering syntax, type the entire syntax entry, including the continuation lines, on the same input line.

Hungarian Naming Conventions

As explained in Chapter 8, “[UCI Functions](#),” certain elements of the Hungarian naming convention are used as prefixes and tags in the detailed descriptions of UCI calls. Examples include:

Prefix or Tag	Description
<i>ipar</i>	<i>index parameter</i>
<i>pib</i>	<i>pointer to an index byte</i>
<i>rgb</i>	<i>range (array) of bytes</i>

UCI Prefixes and Tags

Help

To get Help about UCI, choose Programs -> IBM U2 -> UniDK -> UCI – Help from the **Start** menu.

UniVerse Documentation

UniVerse documentation includes the following:

UniVerse Installation Guide: Contains instructions for installing UniVerse 10.3.

UniVerse New Features Version 10.3: Describes enhancements and changes made in the UniVerse 10.3 release for all UniVerse products.

UniVerse BASIC: Contains comprehensive information about the UniVerse BASIC language. It includes reference pages for all UniVerse BASIC statements and functions. It is for experienced programmers.

UniVerse BASIC Commands Reference: Provides syntax, descriptions, and examples of all UniVerse BASIC commands and functions.

UniVerse BASIC Extensions: Describes the following extensions to UniVerse BASIC: UniVerse BASIC Socket API, Using CallHTTP, and Using WebSphere MQ with UniVerse.

UniVerse BASIC SQL Client Interface Guide: Describes how to use the BASIC SQL Client Interface (BCI), an interface to UniVerse and non-UniVerse databases from UniVerse BASIC. The BASIC SQL Client Interface uses ODBC-like function calls to execute SQL statements on local or remote database servers such as UniVerse, IBM, SYBASE, or INFORMIX. This book is for experienced SQL programmers.

Administering UniVerse: Describes tasks performed by UniVerse administrators, such as starting up and shutting down the system, system configuration and maintenance, system security, maintaining and transferring UniVerse accounts, maintaining peripherals, backing up and restoring files, and managing file and record locks, and network services. This book includes descriptions of how to use the UniVerse Admin program on a Windows client and how to use shell commands on UNIX systems to administer UniVerse.

Using UniAdmin: Describes the UniAdmin tool, which enables you to configure UniVerse, configure and manager servers and databases, and monitor UniVerse performance and locks.

UniVerse Security Features: Describes security features in UniVerse, including configuring SSL through UniAdmin, using SSL with the CallHttp and Socket interfaces, using SSL with UniObjects for Java, and automatic data encryption.

UniVerse Transaction Logging and Recovery: Describes the UniVerse transaction logging subsystem, including both transaction and warmstart logging and recovery. This book is for system administrators.

UniVerse System Description: Provides detailed and advanced information about UniVerse features and capabilities for experienced users. This book describes how to use UniVerse commands, work in a UniVerse environment, create a UniVerse database, and maintain UniVerse files.

UniVerse User Reference: Contains reference pages for all UniVerse commands, keywords, and user records, allowing experienced users to refer to syntax details quickly.

Guide to Retrieve: Describes Retrieve, the UniVerse query language that lets users select, sort, process, and display data in UniVerse files. This book is for users who are familiar with UniVerse.

Guide to ProVerb: Describes ProVerb, a UniVerse processor used by application developers to execute prestored procedures called procs. This book describes tasks such as relational data testing, arithmetic processing, and transfers to subroutines. It also includes reference pages for all ProVerb commands.

Guide to the UniVerse Editor: Describes in detail how to use the Editor, allowing users to modify UniVerse files or programs. This book also includes reference pages for all UniVerse Editor commands.

UniVerse NLS Guide: Describes how to use and manage UniVerse's National Language Support (NLS). This book is for users, programmers, and administrators.

UniVerse SQL Administration for DBAs: Describes administrative tasks typically performed by DBAs, such as maintaining database integrity and security, and creating and modifying databases. This book is for database administrators (DBAs) who are familiar with UniVerse.

UniVerse SQL User Guide: Describes how to use SQL functionality in UniVerse applications. This book is for application developers who are familiar with UniVerse.

UniVerse SQL Reference: Contains reference pages for all SQL statements and keywords, allowing experienced SQL users to refer to syntax details quickly. It includes the complete UniVerse SQL grammar in Backus Naur Form (BNF).

Related Documentation

The following documentation is also available:

UniVerse GCI Guide: Describes how to use the General Calling Interface (GCI) to call subroutines written in C, C++, or FORTRAN from BASIC programs. This book is for experienced programmers who are familiar with UniVerse.

UniVerse ODBC Guide: Describes how to install and configure a UniVerse ODBC server on a UniVerse host system. It also describes how to use UniVerse ODBC Config and how to install, configure, and use UniVerse ODBC drivers on client systems. This book is for experienced UniVerse developers who are familiar with SQL and ODBC.

UV/NET II Guide: Describes UV/Net II, the UniVerse transparent database networking facility that lets users access UniVerse files on remote systems. This book is for experienced UniVerse administrators.

UniVerse Guide for Pick Users: Describes UniVerse for new UniVerse users familiar with Pick-based systems.

Moving to UniVerse from PI/open: Describes how to prepare the PI/open environment before converting PI/open applications to run under UniVerse. This book includes step-by-step procedures for converting INFO/BASIC programs, accounts, and files. This book is for experienced PI/open users and does not assume detailed knowledge of UniVerse.

API Documentation

The following books document application programming interfaces (APIs) used for developing client applications that connect to UniVerse and UniData servers.

Administrative Supplement for APIs: Introduces IBM's seven common APIs, and provides important information that developers using any of the common APIs will need. It includes information about the UniRPC, the UCI Config Editor, the *ud_database* file, and device licensing.

UCI Developer's Guide: Describes how to use UCI (Uni Call Interface), an interface to UniVerse and UniData databases from C-based client programs. UCI uses ODBC-like function calls to execute SQL statements on local or remote UniVerse and UniData servers. This book is for experienced SQL programmers.

IBM JDBC Driver for UniData and UniVerse: Describes UniJDBC, an interface to UniData and UniVerse databases from JDBC applications. This book is for experienced programmers and application developers who are familiar with UniData and UniVerse, Java, JDBC, and who want to write JDBC applications that access these databases.

InterCall Developer's Guide: Describes how to use the InterCall API to access data on UniVerse and UniData systems from external programs. This book is for experienced programmers who are familiar with UniVerse or UniData.

UniObjects Developer's Guide: Describes UniObjects, an interface to UniVerse and UniData systems from Visual Basic. This book is for experienced programmers and application developers who are familiar with UniVerse or UniData, and with Visual Basic, and who want to write Visual Basic programs that access these databases.

UniObjects for Java Developer's Guide: Describes UniObjects for Java, an interface to UniVerse and UniData systems from Java. This book is for experienced programmers and application developers who are familiar with UniVerse or UniData, and with Java, and who want to write Java programs that access these databases.

UniObjects for .NET Developer's Guide: Describes UniObjects, an interface to UniVerse and UniData systems from .NET. This book is for experienced programmers and application developers who are familiar with UniVerse or UniData, and with .NET, and who want to write .NET programs that access these databases.

Using UniOLEDB: Describes how to use UniOLEDB, an interface to UniVerse and UniData systems for OLE DB consumers. This book is for experienced programmers and application developers who are familiar with UniVerse or UniData, and with OLE DB, and who want to write OLE DB programs that access these databases.

Introduction

What Is an SQL Call Interface?	1-3
SQL Call Interface Versus Embedded SQL	1-3
Advantages of Call Interfaces	1-4
Language Support	1-5
Operating Platforms	1-6
Compliance with the ODBC 2.0 Standard	1-7
Requirements for UCI Applications	1-8

This chapter provides an introduction to UCI (Uni Call Interface). UCI is a C-language application programming interface (API), enabling application programmers to write client programs that use SQL function calls to access data in UniVerse and UniData databases.

UCI is designed for use by third-party application developers, tools vendors, and end-user developers who want to write C-hosted, SQL-based client programs for use with tables and files in a database account.

What Is an SQL Call Interface?

UCI is referred to as an SQL call interface because it is an API that uses function calls to invoke dynamic SQL statements.

Dynamic SQL functionality lets a client application both generate and execute SQL statements at run time. Generally, each SQL statement is prepared before execution, with the database server generating a data access plan and a description of the result set; the statement can then be executed repeatedly using the same access path, reducing processing overhead. Another significant feature of dynamic SQL is the ability to include parameters in SQL statements. Parameters are like host variables in embedded SQL, with values assigned to the parameters before execution or retrieved from them after execution.

SQL Call Interface Versus Embedded SQL

An SQL call interface differs from an embedded SQL interface in how it invokes SQL. An application containing embedded SQL must first be passed through a precompiler to convert the embedded SQL statements into the language of the host program. The output from this precompilation is then compiled by the host language compiler, and the compiled code is bound to the database and executed.

An application using an SQL call interface requires neither precompilation nor binding. Instead, it calls upon a standard set of functions to execute SQL statements at execution time. Call interfaces are straightforward and easy to use for programmers familiar with function call libraries. Host variables and other artifacts of embedded SQL are not needed. Instead of passing the SQL statements through a precompiler, application programmers use the interface directly.

Advantages of Call Interfaces

The call interface approach enhances application portability since there is no need for a product-specific precompiler. Client applications can be distributed as compiled applications or run-time libraries instead of as source code that must be precompiled. Moreover, a call interface application does not need application-controlled global data areas such as SQLCA and SQLDA used in the embedded SQL approach. Instead, the call interface allocates and manages these structures, providing a *handle* with which the application can refer to them.

Language Support

UCI is targeted to application development in C, but the library is linkable with and works with client programs written in other languages, including C++.

UCI fully supports the UniVerse programmatic SQL language, as defined in *UCI Developer's Guide*.

Operating Platforms

The database server can be either on the same platform as the application or on a different platform accessible through either a TCP/IP or a LAN Manager network (network software must be installed even for local access). Neither UniVerse nor UniData needs to be installed on the client platform.

The network connection uses the UniRPC, a remote procedure call library.

Compliance with the ODBC 2.0 Standard

UCI is modelled on the ODBC (Open Database Connectivity) standard as defined in *Microsoft ODBC 2.0 Programmer's Reference and SDK Guide*. UCI is an API oriented to UniVerse and UniData; it is not a UNIX or Windows ODBC product. It models only the API side of the ODBC standard, not the driver/transport side. UCI is a look-alike ODBC interface that should reduce the learning curve, training costs, and development expenses for those familiar with ODBC specifications, or with the UniVerse BASIC SQL Client Interface (also modelled on the ODBC standard).

Requirements for UCI Applications

To run UCI applications, you need the following:

- On a UNIX server:
 - UniVerse Release 8.3.3 or later; or UniData Release 5.1 or later
 - TCP/IP
 - UniRPC daemon (*unirpcd*) running
- On a Windows server:
 - UniVerse Release 9.3.1 or later; or UniData Release 5.1 or later
 - TCP/IP, if connected to a UNIX client
 - TCP/IP or LAN Manager, if connected to a Windows client
 - UniRPC service (*unirpc*) running
- On a UNIX client:
 - TCP/IP
 - Required UCI files copied from the software development kit (SDK)
- On a Windows client:
 - TCP/IP, if connected to a UNIX server
 - TCP/IP or LAN Manager, if connected to a Windows server
 - Required UCI files copied from the software development kit (SDK)

To develop UCI applications, C-language development tools must be available on your development system.

Getting Started

Installing UCI.	2-3
On UNIX Systems	2-3
On Windows Systems	2-4
Version Compatibility	2-4
Creating and Running the Sample Application	2-6
Creating and Running Client Programs	2-8
UCI Administration	2-10
Maintaining the UCI Configuration File	2-10
Administering the UniRPC	2-10

This chapter explains how to:

- Install UCI
- Create and run the sample application
- Create and run client application programs developed using UCI
- Perform the necessary system administration

Installing UCI

The installation of UCI is different on UNIX and Windows platforms.

On UNIX Systems

The UCI software development kit (SDK) is included on the UniVerse installation media for UNIX systems. It is installed from the UCI group as part of the standard UniVerse installation.

The installation process creates a directory called *ucisdk* in the *unishared* directory, whose path is stored in the file */.unishared*. The *ucisdk* directory contains the following files:

File	Description
<i>UCI.h</i>	A C header file used when compiling UCI application programs.
<i>UCI.a</i>	A library file used when linking UCI application programs.
<i>ucimsg.text</i>	A message text file used when running UCI applications.
<i>uci.config</i>	A configuration file used when running UCI applications.
<i>ucisample.c</i>	C source code for a sample UCI program <i>ucisample</i> . Program code for <i>ucisample.c</i> is in Appendix B, “ The UCI Sample Program. ”
<i>Make.UCI</i>	A make file for building <i>ucisample</i> . <i>Make.UCI</i> varies from platform to platform. The <i>Make.UCI</i> supplied is customized for your platform.
version	A text file containing the current version of UCI.

ucisdk Directory Files

On Windows Systems

UCI is available for 32-bit Windows only. It is one of several APIs in the UniDK (Uni Development Kit). The UniDK is installed using the standard Microsoft Windows installation procedure. The following UniDK files are used for UCI development:

File	Description
include\UCI.h	A C header file used when compiling UCI application programs.
lib\uci.lib	A library file used when linking UCI application programs.
bin\uci.dll	A DLL used when running UCI applications.
bin\unirpc32.dll	A DLL used when running UCI applications.
redist\platform\shared\ucimsg.text	A message text file used when running UCI applications.
samples\platform\uci\uci.config	A configuration file used when running UCI applications.
samples\platform\uci\ucisample.c	C source code for a sample UCI program <i>ucisample</i> . Program code for <i>ucisample.c</i> is in Appendix B, “ The UCI Sample Program .”

UniDK Files for UCI Development

platform is either i386 or ALPHA.

Version Compatibility

New versions of the UCI server components in the *unishared* directory are backward-compatible with earlier versions, so you can always upgrade the *unishared* directory. However, if you upgrade *unishared* in a new location, you can revert to the older *unishared* directory by doing the following:

1. Shut down the database.

2. If you have done any of the following since you last used the older *unishared* directory:

- Uninstalled the database without reinstalling in the same directory.
- Upgraded the database in a different directory
- Installed a new instance of the database

Copy the following files from the newer *unishared* directory to the older one:

- *unishared\sharedby*
- *unishared\unirpc\unirpcservices*

Make sure these files have the same permissions and ownership as before.

3. **On UNIX systems:** Update the file *./unishared* to contain the absolute pathname of the older *unishared* directory.

On Windows platforms: Do the following:

- Update the Registry Value
HKEY_LOCAL_MACHINE\SOFTWARE\ibm\unishared\path to contain the absolute path of the older *unishared* directory.
- Update the Registry Value
HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\unirpc\ImagePath to contain the following path:
unishared\unirpc\unirpcd.exe
unishared is the full path of the older *unishared* directory.
- Copy the file *unishared\unirpc\unirpc32.dll.bak* from the old directory to the *Windows\system32* directory, then rename it *unirpc32.dll*.
- Restart your PC.

4. Restart the database.

Creating and Running the Sample Application

The easiest way to create and run the sample application program is to do it on a system where the database is installed, with the client connecting to *localuv*. You cannot set up the data on a nonserver Windows system, because it is necessary to invoke the database to run the sample.

To create and run the sample program *ucisample*, do the following:

1. On a UNIX server, change directories to the *ucisdk* directory and invoke UniVerse, creating a database account if none exists.

On a Windows server, change directories to
UNIDK\SAMPLES\platform\UCI. *platform* can be i386 or ALPHA.

2. To create the 10 database demonstration files and populate them, at the database prompt enter the following command on the server:

MAKE.DEMO.FILES

These files are used by the *ucisample* program.

3. On a UNIX server, exit the database and enter the following command:

make -f Make.UCI ucisample

Note: The system must contain development tools such as a C compiler, linker, *make* utility, and the like. It must also contain a TCP/IP library.

On a Windows server, exit the database and enter the following command:

nmake ucisample

Note: *nmake* requires Microsoft Visual C/C++ V2, V4, V5, or the equivalent.

4. If the make was successful, enter the following command on the client to run the sample program:

ucisample

5. At the prompt, enter:

localuv



6. At the prompt, enter the location of the demo data files on the server. You can specify the location as one of the following:
 - The name of a schema
 - The name of a database account
 - The full path of a schema directory
7. At the prompt, enter a user name that is valid on the server.
8. At the prompt, enter the user's password. The password is not echoed to the screen.

The sample program uses UCI to issue SQL statements against the files created in step 2. As the program executes, it informs you of its progress.

Creating and Running Client Programs

You can run your client application on any platform similar to the platform where you created it. UniVerse need not be present on the system on which the client application is running.

On a UNIX server, the UniRPC daemon (*unirpcd*) must be running. On a Windows server, the *unirpc* service must be running. The *unirpcservices* file must have an entry for *uvserver* or *udserver*. These conditions must be satisfied even if the application is connecting to the local database. The procedures for administering the UniRPC are described in the *Administrative Supplement for Client APIs*.

To create and run your own client application programs, complete the following steps:

1. Use *ucisample.c* and *Make.UCI* or *Makefile* as examples.

On a UNIX client, to work in a directory other than *ucisdk* in the *unishared* directory, copy *ucisample.c* and *Make.UCI* to the other directory, and modify *Make.UCI* to find the *UCI.h* include file and the *UCI.a* archive file. Now create your own UCI program and compile and link it.

On a Windows client, to work in a directory other than *\UNIDK\SAMPLES*, copy *ucisample.c* and *Makefile* to that directory and modify *Makefile* to find the *UCI.h* include file and the *UCI.lib* library. Now create your own UCI program and compile and link it.

2. To run your program on another system, copy the executable to an appropriate directory on the other system. If your program is to run on a Windows system, see the Redistribution Notes shipped with the UniDK for details on distributing UCI-based applications.
3. Copy the UCI configuration file and *ucimsg.text* to an appropriate directory on the client system, such as the */etc* directory. These files must be in one of the following directories:
 - Your current working directory.
 - The UV account directory. On UNIX systems, this directory is pointed to by *./uvhome*. On Windows systems, the directory is listed in the Windows Registry.
 - On UNIX systems, the */etc* directory. On Windows systems, one of the directories specified in the PATH variable.

4. Before running the application, edit the UCI configuration file to define the data source to which the application will connect. Use the UCI Config Editor to create data source definitions in the UCI configuration file (for details see *Administrative Supplement for Client APIs*).

The supplied UCI configuration file includes two data source definitions, *localuv* and *localud*, for the local database. If your application needs to connect to some other database, you must edit the UCI configuration file. Chapter 3, “[Configuring UCI](#),” provides more information about this configuration file.

UCI Administration

Once UCI has been installed, it needs little administration. The only major tasks are:

- Maintaining the UCI configuration file (on the client)
- Administering the UniRPC (on the server)

Maintaining the UCI Configuration File

On the client, system administration consists of maintaining the appropriate entries in the UCI configuration file, described in Chapter 3, “[Configuring UCI](#).” Changes to this file should be relatively infrequent, and the system administrator can maintain it using the UCI Config Editor or a text editor.

The client searches for the UCI configuration file (and the *ucimsg.text* file) in the following places:

1. The current working directory.
2. The UV account directory.
3. On a UNIX system, the */etc* directory. On a Windows platform, each directory specified in the PATH environment variable.

Administering the UniRPC

On the server the UniRPC handles requests from client machines. The UniRPC is required even if the server and client machines are the same. The UniRPC uses TCP/IP or LAN Manager transport layer software to communicate between the client and the server.

In particular, the UniRPC daemon (*unirpcd*) on UNIX systems, and the *unirpc* service on Windows platforms, receive SQLConnect requests and start the appropriate server processes to support each UCI application. On UNIX each UCI application has two supporting processes (*uvserver* and *uvsrvhelpd*) on the server while the application is connected. On Windows platforms, a helper thread runs as part of the *uvserver* process. The *uvserver* process uses the same amount of system resources as a local database user.

Before any UCI client applications can run, the administrator of the database server must ensure that the UniRPC daemon or service is running on the server. The UNIX server machine must be running Release 8.3.3 or later of UniVerse. The Windows server machine must be running Release 9.3.1 or later.

See *Administrative Supplement for Client APIs* for more information about the UniRPC, including how to do the following:

- Start and stop the UniRPC daemon or service manually
- Start the UniRPC daemon or service automatically
- On UNIX, add nodes to and remove nodes from the network
- Change the number of the UniRPC port



Note: Any change you make to the UniRPC service or the UniRPC daemon affects all databases that use it.

Administrative Supplement for Client APIs also describes the structure and function of the `unirpcservices` file in the `unirpc` directory. The `unirpcservices` file contains entries for `uvserver` and `udserver`.

Configuring UCI

Configuring a Database Server for UCI.	3-3
UniRPC	3-3
UniVerse NLS	3-4
Configuring a Client System for UCI	3-5
Configuration Parameters.	3-5
Editing the UCI Configuration File.	3-8
Changing UCI Configuration File Parameters	3-10
Configuring UCI for an NLS-Enabled UniVerse Server	3-12

This chapter describes how to configure both the client and the server systems for UCI.

Configuring a Database Server for UCI

The process of configuring a server system is minimal.

- As of Release 8.3.3, any UniVerse system on a UNIX platform can be a server for UCI client application programs.
- As of Release 9.3.1, any UniVerse system on a Windows platform can be a server for UCI client application programs.
- As of Release 5.1, any UniData system on a UNIX or Windows platform can be a server for UCI client application programs.

UniRPC

The UniVerse server (*uvserver*) uses the UniRPC facility (remote procedure call), which is installed with UniVerse. To make UniVerse available as a server, the UniRPC daemon (*unirpcd*) must be running on UNIX systems, or the *unirpc* service must be running on Windows systems.

On UNIX systems, the UniRPC services file, *unirpcservices*, on the server must contain an entry similar to the following:

```
uvserver /usr/ibm/uv/bin/uvsrvd * TCP/IP 0 3600
```

On Windows systems, the entry would be:

```
uvserver C:\IBM\UV\bin\uvsrvd.exe * TCP/IP 0 3600
```

When the client system requests a connection to a service on the server, the local UniRPC daemon or service uses the *unirpcservices* file to verify that the client can start the requested service, which in this case is *uvserver*. Once the daemon or service is started, UCI clients can connect to the database server.

For more information about the UniRPC, see the *Administrative Supplement for Client APIs*.

UniVerse NLS

If UniVerse is running with NLS enabled, you must install any character maps needed by clients. If you need to modify existing maps or derive appropriate new maps to install, use the UniVerse NLS menus, described in the *UniVerse NLS Guide*. The easiest way to ensure that client programs use that map is to see that the client's UCI configuration file contains the name of the new map.

Configuring a Client System for UCI

Various parameters in the UCI configuration file on the client system control the operation of UCI.

The following sections deal with those parameters of interest to UCI clients. Do not change any of the other parameters in the UCI configuration file.

Configuration Parameters

Configuration parameters of interest to UCI developers are described in the following table.



Warning: You can change the values of *MAPERROR*, *MAXFETCHBUFF*, and *MAXFETCHCOLS*. If the UniVerse server to which you are connecting has *NLS* enabled, you can also change the values of the *NLS* and *NLSLC* parameters. Changing other parameters can make UCI unusable.

Parameter	Description	Default
AUTOINC	Produces an <code>SQLColAttributes</code> report if the column is an auto-increment column.	No
CASE	Produces an <code>SQLColAttributes</code> report if the column is case-sensitive.	Yes
DBMSTYPE	Specifies the type of database you want to access (UNIDATA, UNIVERSE, or any other database type, such as DB2).	<i>none</i>
DESCB4EXEC	Indicates if the database's describe operation is legal before executing the SQL statement (for internal use only).	Yes
DSPSIZE	Produces an <code>SQLColAttributes</code> report showing the column display size.	Yes
HOST	Specifies the name of the server machine or its network IP address.	<i>none</i>

Configuration Parameters

Parameter	Description	Default
MAPERROR	Maps UniVerse error codes to standard ODBC SQLSTATE error codes. Whenever the server returns one of the mapped codes as an error condition, UCI sets the SQLSTATE variable equal to the five-character code defined in the ODBC standard.	List
MARKERNAME	Indicates if the database uses names for parameter markers. If not, the ? (question mark) is the marker character.	No
MAXFETCHBUFF	Controls the maximum buffer size on the server to hold data rows. The server usually fills this buffer with as many rows as possible before sending data to the client. If any single row exceeds the length of MAXFETCHBUFF, SQLFetch fails, and you should increase the value of this parameter.	8192 bytes
MAXFETCHCOLS	Controls the maximum number of column values the server can put in the buffer before sending data to the client. If the number of columns in the result set exceeds the number specified by MAXFETCHCOLS, SQLFetch fails, and you should increase the value of this parameter.	400 column values
NETWORK	Specifies the network used to access the data source (TCP/IP or LAN).	<i>none</i>
NLSLCALL	Specifies all components of a locale.	<i>none</i>
NLSLCCOLLATE	Specifies the name of a locale whose sort order to use.	<i>none</i>
NLSLCCTYPE	Specifies the name of a locale whose character type to use.	<i>none</i>
NSLCMONETARY	Specifies the name of a locale whose monetary convention to use.	<i>none</i>
NSLCNUMERIC	Specifies the name of a locale whose numeric convention to use.	<i>none</i>
NLSLCTIME	Specifies the name of a locale whose time convention to use.	<i>none</i>

Configuration Parameters (Continued)

Parameter	Description	Default
NLSLOCALE	Specifies all components of a locale.	<i>none</i>
NLSMAP	Specifies the name of the server's NLS map for the connection. For a client to connect to the server successfully, the server must be able to locate the specified map, which must also be installed in the server's shared memory segment.	<i>none</i>
NULLABLE	Produces an SQLDescribeCol and SQLColAttributes report if the column is nullable.	Yes
SEARCH	Produces an SQLColAttributes report if the column is searchable.	Yes
SERVICE	Specifies the name of the server process for the DBMSTYPE you specified. For UniData, specify <i>udserver</i> ; for UniVerse, specify <i>uvserver</i> .	<i>none</i>
TXBEHAVIOR	Defines default autocommit/manual-commit transaction behavior. Normally, UniVerse is autocommit by default.	1
TXCOMMIT	Database SQL statement for committing a transaction (for internal use only).	No
TXROLL	Database SQL statement for rolling back a transaction (for internal use only).	No
TXSTART	Database SQL statement for starting a transaction (for internal use only).	No
TYPENAME	Produces an SQLColAttributes report showing the name of the SQL TYPE for the column.	Yes
UNSIGNED	Produces an SQLColAttributes report if the column is UNSIGNED.	No
UPDATE	Produces an SQLColAttributes report if the column is updatable.	Yes

Configuration Parameters (Continued)

The following parameters are not used by UCI. They control the UniVerse BASIC SQL Client Interface, which allows data interchange between a UniVerse client BASIC program and a non-UniVerse or UniVerse database.

DATEFETCH	EODCODE	MAXVARCHAR	SMINTPREC
DATEFORM	FLOATPREC	PRECISION	SQLTYPE
DATEPREC	INTPREC	REALPREC	SSPPORTNUMBER
DBLPREC	MAXCHAR	SCALE	USETGITX



Warning: Do not define these parameters for any data source that points to a UniVerse or UniData database. If you do, the results may be unpredictable.

Editing the UCI Configuration File

To create or modify data source definitions, edit the UCI configuration file.

On UNIX client systems running UniVerse

Use the UniVerse System Administration menus or any text editor to edit the UCI configuration file. The UniVerse System Administration menus are described in *UniVerse BASIC SQL Client Interface Guide*.

On UNIX client systems not running UniVerse

Use any text editor to edit the UCI configuration file.

On Windows client systems

Use the UCI Config Editor or any text editor to edit the UCI configuration file. For information about the UCI Config Editor, see the *Administrative Supplement for Client APIs*.

Default UCI Configuration File

On UNIX systems

The default UCI configuration file shipped with the database looks like this:

```
[ODBC DATA SOURCES]
<localuv>
DBMSTYPE = UNIVERSE
NETWORK = TCP/IP
SERVICE = uvserver
HOST = localhost
```

On Windows systems

The default UCI configuration file shipped with the database looks like this:

```
[ODBC DATA SOURCES]
<localuv>
DBMSTYPE = UNIVERSE
NETWORK = TCP/IP
SERVICE = uvserver
HOST = localhost

<localud>
DBMSTYPE = UNIDATA
NETWORK = TCP/IP
SERVICE = udserver
HOST = localhost
```



Warning: *On Windows systems, do not change the HOST parameters of the <localuv> and <localud> entries.*

This default UCI configuration file lets you access a database on the same hardware platform as the one on which your application is running.

Adding Data Source Definitions to the UCI Configuration File

You can add as many data source entries as you want, each with a different data source name.

To access a remote database on a different platform, add an entry to the configuration file for that database. For example, if the remote system you want to access is named *hq1*, make up a data source name such as *corp* and change the UCI configuration file as follows:

```
[ODBC DATA SOURCES]
<localuv>
DBMSTYPE = UNIVERSE
NETWORK = TCP/IP
SERVICE = uvserver
HOST = localhost

<corp>
DBMSTYPE = UNIVERSE
NETWORK = TCP/IP
SERVICE = uvserver
HOST = hq1
```



Note: *The spaces surrounding the equal signs are required.*

Changing UCI Configuration File Parameters

Two parameters you might want to change are MAXFETCHBUFF and MAXFETCHCOLS. Use these parameters to increase the amount of data in each buffer sent from the server to the client. This will improve performance by reducing the number of data transfers between server and client.

MAXFETCHBUFF specifies the size of the buffer the server uses to hold data rows before sending them to the client. MAXFETCHCOLS specifies the number of column values the server can put in the buffer before sending them to the client. For example, if MAXFETCHCOLS is set to 100 column values and you do a SELECT of 40 columns, no more than two rows can be sent in any buffer, because the total number of column values in two rows is 80. Three rows would contain 120 column values, which exceeds the value of MAXFETCHCOLS.

You can change these parameters for specific data sources or for all database connections. Using the sample configuration file shown previously, you might add entries for MAXFETCHBUFF and MAXFETCHCOLS as shown below to change the *internal default* for those parameters to 16000 and 600, respectively:

```
[ODBC DATA SOURCES]
<localuv>
DBMSTYPE = UNIVERSE
NETWORK = TCP/IP
SERVICE = uvserver
HOST = localhost

<corp>
DBMSTYPE = UNIVERSE
NETWORK = TCP/IP
SERVICE = uvserver
HOST = hq1

[UNIVERSE]
MAXFETCHBUFF = 16000
MAXFETCHCOLS = 600
```

To make the data source *corp* use larger buffers, make the following changes:

```
[ODBC DATA SOURCES]
<localuv>
DBMSTYPE = UNIVERSE
NETWORK = TCP/IP
SERVICE = uvserver
HOST = localhost

<corp>
DBMSTYPE = UNIVERSE
NETWORK = TCP/IP
SERVICE = uvserver
HOST = hq1
MAXFETCHBUFF = 20000
MAXFETCHCOLS = 800

[UNIVERSE]
MAXFETCHBUFF = 16000
MAXFETCHCOLS = 600
```

In this situation, you have set the default for connections to UniVerse to 16000 and 600, but when you connect to the data source *corp*, the local settings of 20000 and 800 override the defaults.

Configuring UCI for an NLS-Enabled UniVerse Server

NLS (National Language Support) is fully documented in *UniVerse NLS Guide*. For information about connecting to an NLS-enabled server, see Chapter 4, “[Developing UCI Applications](#).”

If clients need to override the default server map names and locale settings, you can change the UCI configuration file to contain this information:

- For each data source
- For all UniVerse server connections

NLS users should note that the configuration file is in ASCII format. When you specify NLS and locale settings in the configuration file, you need not make changes to your programs to let client programs work with an NLS-enabled server.

Server Map

Use the NLSMAP parameter to specify the server map to use.

Server Locale

Use the following parameters to specify a locale’s components:

NLSLCTIME
NLSLCNUMERIC
NLSLCMONETARY
NLSLCCTYPE
NLSLCCOLLATE

Use the NLSLOCALE parameter to specify all of a locale’s components.

Use the NLSLCALL parameter to specify a slash-separated list of locale identifiers, as set up in the server’s NLS.LC tables. The syntax for NLSLCALL is:

NLSLCALL = *value1/value2/value3/value4/value5*

For example, you could specify:

```
NLSLOCALE = DE-GERMAN
```

Or you could specify:

```
NLSLCALL = NL-DUTCH/NL-DUTCH/DEFAULT/NL-DUTCH/NL-DUTCH
```

This sets all components of the locale for this connection to those indicated by the entry in the NLS.LC table with ID = NL-DUTCH, except for the LCMONETARY entry, which is loaded from the NLS.LC.MONETARY table for the DEFAULT entry.

If there is more than one entry in the NLSLCALL entry, all entries must be nonempty and must represent valid entries in the appropriate NLS.LC.*category* table.

You can also change only a single component of the locale:

```
NLSLCCOLLATE = NO-NORWEGIAN
```

This forces the server's sort order to be Norwegian.

NLSLCCOLLATE is the most important locale parameter because it affects the order in which rows are returned to the application.

Developing UCI Applications

Writing a UCI Application Program	4-3
Initializing Resources	4-4
Allocating the Environment	4-4
Allocating the Connection	4-5
Connecting to the Server	4-5
Allocating Statement Handles	4-9
Processing SQL Statements	4-10
Transaction Modes	4-10
Function Calls	4-11
Executing an SQL Statement	4-11
Processing Output from SQL Statements	4-15
Checking for Errors	4-17
Freeing the SQL Statement Environment	4-18
Terminating the Connection	4-19
Transaction Processing	4-20
Nested Transactions	4-20
Transaction Isolation Levels	4-22
Handling Multivalued Columns	4-23
Setting the Data Model Mode	4-23
Dynamic Normalization and Associations	4-25

This chapter describes the steps to develop a UCI application program. It covers how to code the three major sections of a program—initializing, processing, and closing—and then discusses related topics such as how to handle the database’s multivalued columns and how to check for and handle errors.

Writing a UCI Application Program

An application has three major phases, executed in the following sequence:

Phase	Description
Initialization	Connecting to the server and allocating and initializing resources in preparation for SQL statement processing.
Processing	Either passing SQL statements to UCI to retrieve and modify the data, or calling and executing procedures stored on the data source.
Termination	Freeing allocated resources, mainly data buffers identified by unique handles, and disconnecting from the server.

Application Phases

In addition to these phases, a program must deal with error returns, which can occur at any point in its execution. It can also request information about the database and its accessible tables and columns.



***Note:** In this chapter and those following, certain elements of the Hungarian naming convention are used when describing the elements of UCI calls (see [Use of Hungarian Naming Conventions](#) in Chapter 8, “*UCI Functions*.”).*

An annotated sample application, *ucisample.c*, is in Appendix B, “[The UCI Sample Program](#).”

Initializing Resources

Before you can perform any actual processing, you must establish the necessary connections. These are the four steps associated with this task:

1. Allocating an environment object (**SQLAllocEnv**)
2. Allocating a connection object (**SQLAllocConnect**)
3. Connecting to the data source (**SQLConnect**)
4. Allocating statement handles (**SQLAllocStmt**)

These steps dynamically allocate objects to be used by the client interface (UCI) for storing essential data between calls, create a handle (a pointer to a structure) for these areas, and return that handle to the application program. Subsequently the application program can return that handle to the client interface, when necessary, as a parameter of a call. For each initialization step there is a corresponding step in the termination process that frees these objects and handles.

Allocating the Environment

SQLAllocEnv, the first call in an application program, allocates an environment and returns a handle for it. The handle points to the data area that contains information about the state of UCI, including a list of the connection handles owned by the application. Some of this information is defined in *UCI.h*, a file included in the source modules.

The environment is of use only to UCI software. The client application program has no need of it, and its only job is to store the returned handle and to release it at the end.

An application must allocate an environment before it can do anything else. Each application can have only one environment handle.

Allocating the Connection

The second step in initializing an application is to pass back the environment handle just received and acquire a connection handle. The connection handle points to an area containing information for the connection managed by UCI. This information includes general status information, transaction status, and diagnostic information.

As with the environment handle, an application's only responsibility in regard to the connection handle is to store the returned handle before connecting to the data source and to release the handle before terminating. Each application can have multiple connection handles.

Connecting to the Server

The third step in initialization is to issue an **SQLConnect** call to connect to a data source that will handle database operations for the client. A data source, as defined in UCI, is an entry in the UCI configuration file (see Chapter 3, [“Configuring UCI”](#)). This entry describes such things as the DBMS type (which for UCI is always UNIVERSE or UNIDATA), the network connection (TCP/IP or LAN Manager), the name of the required RPC server (*uvserver*), and the host (either *localhost*¹, if the server is on the same platform as the client, or the name or IP address of some remote host).

Before issuing an SQLConnect call, an application can establish certain conditions for the connection by issuing one or more **SQLSetConnectOption** calls. The SQLSetConnectOption call can be used to specify the default transaction isolation level, specify a data model (first normal form or nonfirst-normal form), specify the NLS map table and locale information to use, or supply the user ID and password for the connection, if required.

Use SQLSetConnectOption calls to specify the user name (SQL_OS_UID) and password (SQL_OS_PWD) for logging in to a remote database server. On all systems but Windows NT 3.51, if the host specified for this DSN is either *localhost* or the TCP/IP loopback address (127.0.0.1), the user name and password are not required and are ignored if specified. On Windows NT 3.51 systems, the user name and password are always required, so you must specify *localpc* as the DSN (for information about adding the *localpc* entry to the UCI configuration file, see Editing the UCI Configuration File in Chapter 3, [“Configuring UCI”](#)).

You must provide several pieces of information to SQLConnect: the connection pointer that was just returned by the **SQLAllocConnect** call, a data source name (DSN), and the name of the SQL schema or database account containing the data.

An application can have more than one connection, for connecting to more than one schema or account or to more than one database server.

To close a connection, call SQLDisconnect.

1. *localpc* on Windows NT 3.51 systems.

Connecting to a UniVerse Server with NLS Enabled

NLS (National Language Support) is fully documented in the *UniVerse NLS Guide*.

When a UCI program connects to an NLS-enabled UniVerse server, the map and locale values the server uses depend on the settings of the parameters in the server's *uvconfig* and UCI configuration files, as well as values from the client's operating system and its UCI configuration file. All these values can be explicitly set by the client.

The UniVerse server honors the following configurable parameters in its *uvconfig* file:

Parameter	Description
NLSMODE	Switches NLS mode on or off. A value of 1 indicates NLS is on, a value of 0 indicates NLS is off.
NLSDEFSRVMAP	Specifies the default map to be used for passing string arguments to and from client programs. Used if the client does not assign an explicit map.
NLSLCMODE	Switches locale support on or off. A value of 1 enables locales for the whole UniVerse system. The value is ignored if NLSMODE is set to 0. A value of 0 turns off locales even if NLSMODE is set to 1.
NLSDEFSRVLC	Specifies the default locale for the server, which is used by all client programs accessing the server.

Configuration Parameters

When the client application starts, it determines the default map and locale values to send to the server as follows:

1. It gets a map and locale from the client's operating system.

On UNIX systems:

- UCI gets the map from the UV_UCI_CHARMAP environment variable, if found.
- All five locale categories are first set to the value in the UVLANG environment variable, if found.

On Windows systems:

- UCI gets the map through GetACP(), which returns the CodePage and prefixes it with WIN:. CodePage is an integer like 1252 or 1200 or 932.
- UCI gets locale information through GetThreadLocaleString(), which returns the ThreadLocale and prefixes it with WIN:. ThreadLocale is an integer like 0409 or 0809 or 0C07.

2. It reads the UCI configuration file, if found, and sequentially replaces values set by step 1 with values set by the UCI configuration file.

Client programs can use the SQLSetConnectOption call to override any of the server's default map and locale values.

If the specified mapping or locale information is incorrect, SQLConnect returns an error and does not connect to the server.

Matching Client to Server

Certain combinations of clients and servers may not be able to transfer data reliably because of a mismatch in the character mapping, locale settings, or both at the client end.

UniVerse Release 9.4 (or Later) Client and Release 9.4 (or Later) Server

The following table shows which combinations of server map specification and locale specification are allowed depending on the client type and the NLS status of the server.

NLS State of the Server	Is Client NLSMAP Set?	Are Any Client Locale Settings Set?	Action
ON	Yes	Yes	NLSMAP and NLSLOCALE are used if the ID is valid.
	Yes	No	NLSMAP is used if the ID is valid. NLS DEFAULT locale is used.
	No	Yes	NLS DEFAULT character map is used. NLSLOCALE is used if the ID is valid.
	No	No	NLS defaults are used for both.
OFF	Yes	Yes	Connection is rejected.
	Yes	No	Connection is rejected.
	No	Yes	Connection is rejected.
	No	No	Connection succeeds.

NLS Map and Locale Settings

UniVerse Release 9.3 (or Earlier) Client and Release 9.4 (or Later) Server:

UniVerse releases before 9.4 do not support UniVerse NLS. Therefore, any Release 9.3 (or earlier) client cannot request a map or locale; it uses the server's current map and locale settings. If these are the NLS defaults, the results returned are the same as those from a 9.3 or earlier server.

UniVerse Release 9.4 (or Later) Client and Release 9.3 (or Earlier) Server:

Because UniVerse releases before 9.4 do not support UniVerse NLS, Release 9.4 clients can connect to Release 9.3 (or earlier) servers only if the client does not request NLS options.

Allocating Statement Handles

Allocating statement handles is either the last step of the initialization phase or the first step of the SQL statement processing phase. An application must allocate a statement handle before executing any statements. Each statement handle is associated with a specific connection handle. Within an application, UCI allows a virtually unlimited number of statement handles.

Use the **SQLAllocStmt** function to allocate a statement handle. A statement handle points to the data area containing information about an SQL statement managed by UCI. This information includes dynamic arguments, cursor data, bindings, result values, status information, and diagnostic information. A statement handle is a variable of type **HSTMT**.

You can release statement handles by calling **SQLFreeStmt**.

Processing SQL Statements

Processing SQL statements is the second major phase in an application. This phase is the heart of the application, where all database operations occur.

Transaction Modes

Two transaction modes are supported: *autocommit* and *manual-commit*. By default, the database is in autocommit mode, but you can put it into manual-commit mode by calling **SQLTransact** with the `SQL_BEGIN_TRANSACTION` flag.

Autocommit Mode

In autocommit mode, each SQL statement is treated as a separate and complete transaction, and the server commits one transaction per statement. If no explicit **SQLTransact** call is issued, all statements are processed in autocommit mode. Data definition language (DDL) statements cannot be executed inside a transaction and must be executed in autocommit mode. All **SQLConnect** and **SQLDisconnect** calls must also be performed in autocommit mode.

Manual-Commit Mode

In manual-commit mode, a transaction begins with an **SQLTransact** call with an *fType* of `SQL_BEGIN_TRANSACTION`. If another transaction is already active, this transaction becomes a *nested* transaction.

The final step in manual-commit mode is to either commit or roll back the transaction with another call to **SQLTransact**, this time with an *fType* of `SQL_COMMIT` or `SQL_ROLLBACK`.

For more details about transaction processing, see [“Transaction Processing”](#) on page 20.

Function Calls

SQL statement processing can involve a number of function calls, as shown in the following figure. This figure illustrates all the steps in SQL statement processing except error checking. However, certain steps are optional. For example, instead of releasing the statement handle after each SQL statement is processed and then allocating a new handle for the next SQL statement, you could unbind the columns and parameters associated with the handle and then reuse the handle for the next SQL statement.

Statement processing is divided into the following steps:

1. Start a transaction (optional). If you do not start a transaction, each succeeding SQL statement is processed as a separate transaction.
2. Process one or more SQL statements. For each statement do the following:
 - Submit the statement for either direct or prepared execution.
 - Analyze the result set (if any) produced, assign the storage necessary to hold the results, and then fetch the results row by row.
 - Check for errors and, if any occurred, obtain the error codes and take appropriate action.
 - Free the SQL statement environment.
3. Terminate the transaction by either committing it or rolling it back (optional).

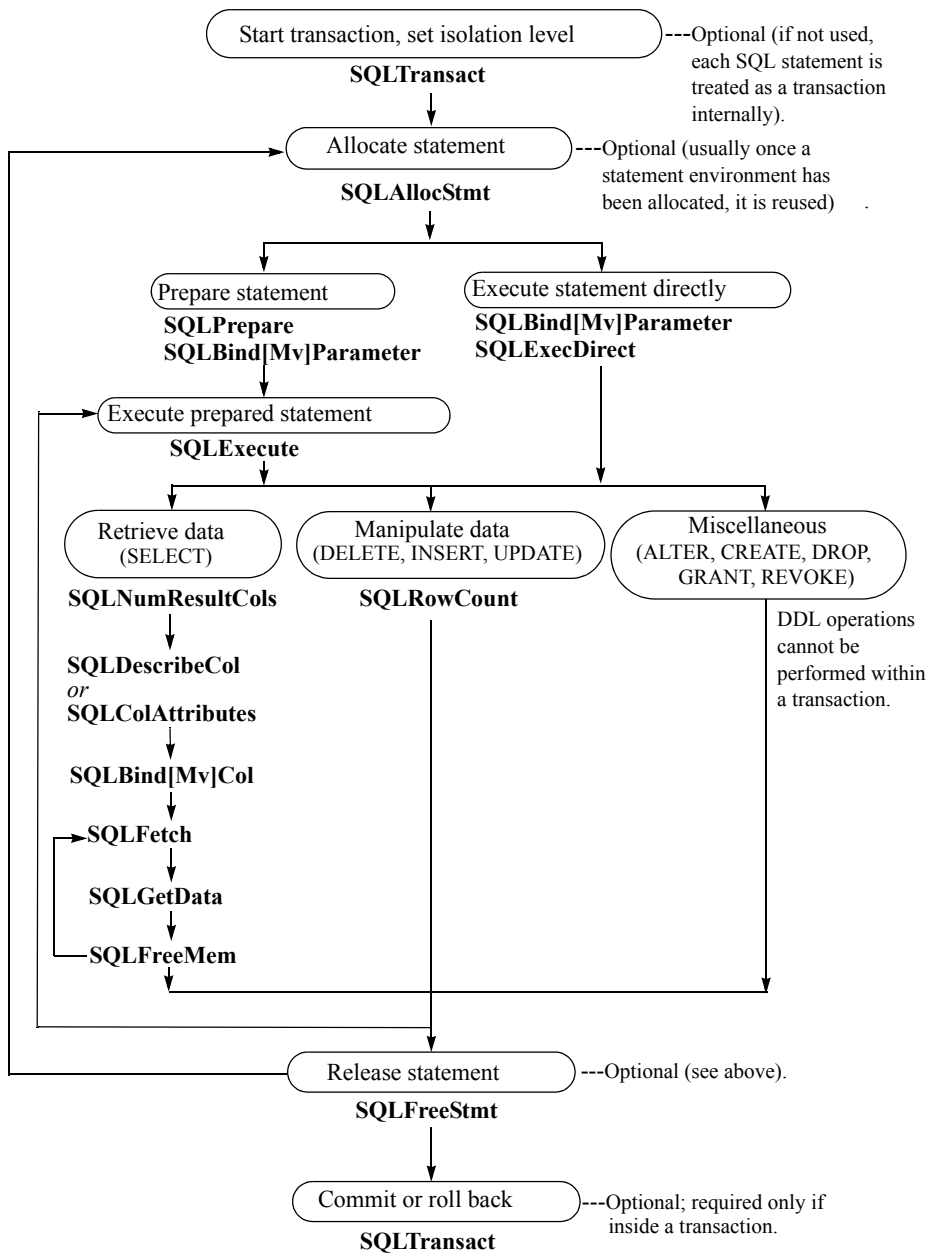
Executing an SQL Statement

Using an existing statement handle, the application can submit SQL statements for execution. There are two ways to do this: executing a statement directly, or preparing a statement first and then executing it.

Executing an SQL Statement Directly

Executing an SQL statement directly is the simplest and most efficient approach if the statement is to be executed only once. Direct execution is similar to execute immediate in embedded SQL.

SQLExecDirect parses and binds an SQL statement to an *hstmt* and then executes it. Whenever an application issues an **SQLExecDirect** call, UCI does the following:



SQL Statement Processing Steps

1. Passes the SQL statement to the server for query plan generation
2. Retrieves information about result set columns from the server if the operation was a SELECT
3. Gets the current input parameter values, converts them, and sends them to the data source (see [“Using Parameter Markers in SQL Statements”](#) on page 14).
4. Requests that the server execute the statement with the parameters given
5. Gets output parameters from the data source and stores them in the designated variables
6. Returns errors, if any

Preparing and Executing an SQL Statement

Instead of using a call to `SQLExecDirect` to do everything, you can parse and bind a statement to an *hstmt* using an **SQLPrepare** call and then execute it with one or more calls to `SQLExecute`.

There are two main advantages to this approach:

1. Greater efficiency for statements that are to be executed many times. The data source produces an access plan for executing the SQL statement, and then uses that same plan for each iteration of the statement.
2. The application can obtain information about the result set before actually executing the statement.

When an application issues an `SQLPrepare` call, UCI does the following:

3. Passes the SQL statement for parsing
4. Retrieves information about the result set columns from the server if the operation was a SELECT

The subsequent calls to `SQLExecute`:

1. Get the current parameter values, convert them, and send them to the data source (see [“Using Parameter Markers in SQL Statements”](#) on page 14)
2. Request that the server execute the SQL statement
3. Get output parameters from the data source and store them in the designated variables
4. Return errors, if any

Using Parameter Markers in SQL Statements

Application variables are associated with parameter markers by a process called *parameter binding*. If an SQL statement contains input parameter markers, their values must be retrieved from the application at execution time. Parameter markers are commonly used for such tasks as inserting multiple rows of data into an SQL table.

Parameter markers are represented by a ? (question mark) and indicate places in the SQL statement where application variables are to be substituted when the statement is executed. Markers are numbered from left to right, starting with 1.

The **SQLBindParameter** and **SQLBindMvParameter** functions bind parameters. Both functions accept the number of the parameter marker, the C data type of the variable, the SQL data type of the parameter, and (for **SQLBindParameter**) a pointer to the buffer for the variable, its length, and whether to use it for input, output, or both.

Only the pointer (*rgbValue*) to the application variable is passed when **SQLBindParameter** is called, but the data in the variable is not read until the statement is executed. In this way the application can modify the data in a bound parameter variable and reexecute the statement multiple times, each time with a new value.

For example, if an application is inserting new rows of data into a simple four-column SUPPLIER table, the INSERT statement might look like this:

```
INSERT INTO SUPPLIER (COMPANY, ADDRESS, PHONE,  
CONTACT_NAME) VALUES (?, ?, ?, ?)
```

In this example, there are four parameter markers, and the application must call **SQLBindParameter** once for each marker.

Then, when the application issues an **SQLExecDirect** or **SQLExecute** call to execute the statement, UCI:

1. Ascertains that the application has called **SQLBindParameter** for each parameter marker
2. Gets the current value of each input parameter from its storage area, and converts the data if necessary
3. Passes the parameter values to the data source
4. Gets output parameters from the data source and stores them in the designated variables



This parameter information is retained following execution of the statement and is released only after the application issues an `SQLFreeStmt` call with an `SQL_RESET_PARAMS` or `SQL_DROP` option.

Note: An `SQLSetParam` function is provided for compatibility with ODBC 1.0 and the UniVerse BASIC SQL Client Interface. It binds an application buffer to a parameter marker in an SQL statement, and is essentially a front end to the `SQLBindParameter` call. Also, an `SQLBindMvParameter` function is included as a database extension to handle parameter markers associated with multivalued columns (see “[Handling Multivalued Columns](#)” on page 23 for more information). It is usable with singlevalued columns as well.

Processing Output from SQL Statements

SELECT statements and some called procedures return a set of one or more data rows called a result set. The SQL data manipulation language (DML) statements DELETE, INSERT, and UPDATE, and some called procedures, do not return result sets but only a count of the number of rows affected by the statement. Generally the application does nothing, except in the case of an error return.

SQL data definition language (DDL) statements—ALTER TABLE, SCHEMA, CREATE TABLE, CREATE VIEW, CREATE INDEX, CREATE TRIGGER, DROP SCHEMA, DROP TABLE, DROP VIEW, DROP INDEX, DROP TRIGGER, GRANT, and REVOKE—do not query or modify data and need no further processing, except to check for and handle any error returns that might result from their execution.

A result set comprises one or more rows of data obtained by a SELECT statement or called procedure. The application must retrieve the data one row at a time to process it. When a result set is produced, UCI opens a *cursor* into that result set. A cursor is a pointer to the next row of data that a fetch will return from the result set.

Analyzing and retrieving a result set includes the following steps:

1. Analyzing the result set (optional)
2. Binding application variables to columns (optional)
3. Fetching each row of data, one row at a time, putting the column results into the bound application variables
4. Retrieving data from unbound columns (optional)

Analyzing the Result Set

If the SELECT statement is hard-coded, there is no need to analyze the result set because the application already knows the structure of the result set and data type of each column.

If the SELECT statement is not hard-coded but instead is generated at run time (for example, entered ad hoc by a user), or is a SELECT *, the application needs to know how many columns there are and their data types (as well as the names of the columns).

An application obtains this information by calling `SQLNumResultCols`, then `SQLDescribeCol` or `SQLColAttributes`, either after preparing the statement or after executing it. `SQLNumResultCols` returns the number of columns in the result set, and `SQLDescribeCol` and `SQLColAttributes` return information about those columns.

If the SQL operation is a DELETE, INSERT, or UPDATE, an application may need to know how many rows were affected by the operation. Get this information by calling **`SQLRowCount`**.

Binding Application Variables to Columns

An application calls `SQLBindCol` for each singlevalued column and calls `SQLBindMvCol` for each multivalued column in the result set (however, you can use `SQLBindMvCol` on all columns, whether singlevalued or multivalued). The *fCType* argument of the call determines the type of conversion to be performed on the column's data, and the *rgbValue* (*pCArray* in the case of `SQLBindMvCol`) tells the subsequent `SQLFetch` where to store the converted data.

Fetching Rows

`SQLFetch` reads each row of data, one row at a time, and places the content of each column into the application variable to which it was bound by `SQLBindCol` (or `SQLBindMvCol`). A cursor mechanism keeps track of the current position in the rows. Each time an application calls `SQLFetch`, the cursor moves to the next row, the row's data is retrieved and converted, and the result for each bound column is placed into its assigned storage variable. This process continues until there is no more data to retrieve and `SQL_NO_DATA_FOUND` is returned. For unbound columns or columns bound to variables that are too small, the application can issue an **`SQLGetData`** call to get the remainder of the data.

Retrieving Data from Unbound Columns

Generally, columns to be retrieved are bound. However, there are times when you want to retrieve an unbound column, for example, when you want to deal with the columns individually, or when you are interested in only one column of a table. The application program must first call `SQLFetch` to position the cursor at the next row, then call `SQLGetData` to get the data from a specific column in that row. You can use `SQLGetData` on unbound columns and on columns bound by `SQLBindCol`, but not on a column bound by `SQLBindMvCol`, because `SQLBindMvCol` allocates enough storage automatically.

Checking for Errors

Error conditions can result from executing any SQL statement and must be handled by the application.

Return codes range from `SQL_SUCCESS` (the function completed successfully), to `SQL_ERROR` (the function failed totally).

Along with these return codes are `SQLSTATE`s, which are alphanumeric strings of five characters that further define the warning or error condition indicated by the return code. An application can interrogate the `SQLSTATE` by calling **`SQLError`**, providing more detailed error information from the server.

Not all return codes provide additional `SQLSTATE` information, so first check the return code to see if additional diagnostic information is available. Generally, when an error occurs in an application, the error subroutine displays or prints the name of the function, the return code, the content of `SQLSTATE`, the database error code, and the diagnostic text.

Freeing the SQL Statement Environment

To terminate processing for a statement, issue a call to `SQLFreeStmt`. You can do any of the following:

- Unbind all bound columns and parameters associated with the statement
- Reset parameter markers associated with the statement
- Drop the statement handle and release its associated resources

After you have unbound its columns and parameters, the statement handle is available for reuse, unless you choose to drop it.

Terminating the Connection

One of the last steps an application does is to close the connection to the database. This is done through an **SQLDisconnect** call. Because you cannot issue an SQLDisconnect call if a transaction exists or is active, an application must commit or roll back all active transactions before it issues the call.

Transaction Processing

By default, a client process and connected server processes are initially in autocommit mode, which means that each SQL statement is treated as a separate and complete transaction, and the server commits one transaction per statement.

Use the **SQLTransact** function to enter manual-commit mode and to control transaction behavior and mark the beginning and end of each transaction or subtransaction. Every SQLTransact issued on a client sends a request to the connected server process to begin a transaction or subtransaction on the server. The nesting level is stored in the client so that it knows how many transaction levels exist on the server processes.

A commit of an unnested transaction writes all modified data to the database, releases all locks acquired during the transaction, terminates the transaction, and returns to autocommit mode. If the transaction is nested, any data written is internally committed and made available to the higher (parent) transaction.

If the current transaction is not nested, a rollback discards any changes made during the transaction, and then terminates the transaction. If the transaction is nested, only those changes made by the nested transaction are discarded.

Nested Transactions

In manual-commit mode, transactions can be nested, for example, an application can begin a subtransaction when another transaction or subtransaction is active. Because only one transaction can be active at a time, the subtransaction becomes the current active transaction while its parent transaction becomes temporarily inactive but continues to exist. When a subtransaction completes, it is committed to its parent transaction; it does not commit to the database until the top-level (topmost) transaction commits. If a higher-level transaction or subtransaction rolls back, all subtransactions beneath it also roll back.

The number of nesting levels is restricted to 65,000, but in practice is limited by available memory.

Transaction nesting requires that the transaction mode be manual-commit so that a transaction and its subtransactions must either *all* commit to the database, or else *none* of them commit and the database remains unchanged.

Nested Transactions and SELECT Statements

When a client program requests execution of a SELECT statement on a server, data is fetched interactively between client and server. Because of this interaction, more than one SELECT statement can *exist* at one time. However, since only one SELECT statement can be *active* at one time, the active designation switches between the *existing* statements.

In manual-commit mode, the SELECT statement does not begin a new subtransaction. Instead it is executed as part of the active transaction or subtransaction on the server process.

A call to SQLFetch must be executed at the same isolation level as its corresponding SQLExecute or SQLExecDirect call.

Nested Transactions and DELETE, INSERT, and UPDATE Statements

If the statement is a DELETE, INSERT, or UPDATE, a subtransaction is always begun by the server. The statement is executed, and if it completes successfully, the subtransaction is committed; otherwise it is rolled back.

Nested Transactions and Called Procedures

A called procedure does not begin a transaction.

Nested Transactions and DDL Statements

SQL DDL statements are not permitted within a transaction because their effects on the database cannot be rolled back.

Transaction Isolation Levels

Every transaction or subtransaction runs at a particular transaction isolation level. The possible isolation levels are:

BASIC Isolation Level		SQLTransact <i>ftype</i> Modifier
0	No isolation	Not applicable
1	Read uncommitted	SQL_TXN_READ_UNCOMMITTED
2	Read committed	SQL_TXN_READ_COMMITTED
3	Repeatable read	SQL_TXN_REPEATABLE_READ
4	Serializable	SQL_TXN_SERIALIZABLE

Isolation Levels

A UCI client program has an initial default isolation level of 0 (however, you cannot *explicitly* start a transaction at this isolation level). You can change this to a different default value by issuing an SQLSetConnectOption call. Also, in manual-commit mode, a client can specify an isolation level for a specific transaction when issuing the SQLTransact call that begins the transaction.

There are two additional rules:

- If the transaction is an SQL DML statement, the transaction isolation level used is the higher of:
 - The minimum transaction isolation level required by the statement. The minimum is 0 for SQL SELECT statements, and 2 for SQL DELETE, INSERT, and UPDATE statements.
 - The current default transaction isolation level for the user.
- A subtransaction’s isolation level cannot be lower than that of its parent transaction. Consequently, isolation levels can increase as the nesting level increases, but they cannot decrease. If you attempt to set a subtransaction’s isolation level lower than its parent’s isolation level, the subtransaction inherits the parent’s isolation level.

Handling Multivalued Columns

There are two ways to handle a database table or file that contains multivalued columns:

- As a nonfirst-normal-form (NF²) data structure
- As a set of first-normal-form (1NF) tables that can be joined as can any set of related tables

You choose which data model you want to use by setting the data model mode. In addition, UniVerse provides a means of dynamically normalizing a nonfirst-normal-form database. This lets you work in NF² mode while accessing the data as if it were normalized.

Setting the Data Model Mode

In UCI, how tables with multivalued columns are treated depends on the mode set through the `SQL_DATA_MODEL` *fOption* of the **SQLSetConnectOption** call. This mode is in effect for the duration of the connection.

Setting the *fOption* of `SQLSetConnectOption` to `SQL_DATA_MODEL` and the *vParam* to `SQL_1NF_MODE_OFF` lets UCI “see” and work with all columns of a base table, whether singlevalued or multivalued; this is called NF² (nonfirst-normal-form) mode. Setting *vParam* to `SQL_1NF_MODE_ON` lets UCI “see” and work only with a base table’s singlevalued columns. The base table’s multivalued columns are accessible as separate virtual tables related to the base table. This is called 1NF mode.

For application programmers familiar with UniVerse and its NF² structure, NF² mode is the logical choice. For others who prefer to work with normalized (1NF) tables, operating in 1NF mode is the suggested approach.

Programming for NF² Mode

To treat a table with multivalued columns as an NF² structure, which is the standard UniVerse approach, a UCI client program calls `SQLBindMvParameter` and `SQLBindMvCol`. These two functions, which are extensions to ODBC calls, handle multivalued columns.

SQLBindMvParameter binds an array representing multivalued data to a parameter marker in an SQL statement. SQLBindMvCol does the opposite, turning a multivalued dynamic array into a C array.

Programming for 1NF Mode

Because traditional SQL users usually deal exclusively with first-normal-form tables, 1NF mode enables a client SQL program to treat a UniVerse NF² table as if it were a set of normalized first-normal-form tables.

Conceptually, a UniVerse NF² table containing some singlevalued columns, some associations of multivalued columns, and some unassociated multivalued columns, becomes, in 1NF mode, a set of related tables. This set of tables comprises:

- One table representing all of the singlevalued columns in the base table.
- A normalized table for each of the associations of multivalued columns, and for each unassociated multivalued column. Each association comprises the primary key of the base table plus all multivalued columns of the association. Each unassociated multivalued column comprises the primary key of the base table and the column itself.

When running in 1NF mode:

- All tables appear to be first normal form and to contain only singlevalued columns. References to the name of a base table (*tablename*) access only the singlevalued columns in that table. You can access associated and unassociated multivalued columns (in NF² base tables) only by using dynamic normalization (see [“Dynamic Normalization and Associations”](#) on page 25).
- `SELECT * FROM tablename` retrieves data from all singlevalued columns in the base table but not from any multivalued columns.
- If no column list is specified for `INSERT INTO tablename`, values must be supplied for all of the base table’s singlevalued columns but not for its multivalued columns.

Dynamic Normalization and Associations

Any set of related multivalued columns in a UniVerse table or file can be defined as an association. Dynamic normalization allows SQL statements to access the primary keys of an NF² base table and the columns of any association within it as a 1NF table. In other words, a separate association row is generated for each set of values in the association.

Dynamic Normalization in 1NF Mode

In 1NF mode, a UCI client program sees a base table as having only its primary key plus any singlevalued columns. The program accesses a base table's multivalued columns by dynamically normalizing the table. To access an association of multivalued columns, UniVerse SQL uses the construct *tablename_association*. To access an unassociated multivalued column, it uses the construct *tablename_mvcolumnname*. A *tablename_association* virtual table is defined in UV_TABLES (TABLE_TYPE is ASSOCIATION), and contains in its COLUMNS column the names of the base table's primary key columns and the names of the association's columns. Unassociated multivalued columns are not defined in UV_TABLES.

Dynamic Normalization in NF² Mode

In NF² mode, a UCI client program can also use *tablename_association* to normalize an association of multivalued columns. Also the SELECT statement can explode an association using the UNNEST clause.

An SQL DML statement (SELECT, INSERT, UPDATE, or DELETE) can include *tablename_association* anywhere that *tablename* is valid, but *tablename_association* cannot be used in the following SQL DDL statements: CREATE TABLE, CREATE INDEX, CREATE TRIGGER, ALTER TABLE, DROP TABLE, DROP VIEW, DROP INDEX, DROP TRIGGER, GRANT, or REVOKE.

The primary keys of a dynamically normalized association are always the primary key values of the base table followed by the association key values, with the keys separated by text marks. If the association does not have association keys, use the @ASSOC_ROW keyword to provide a set of virtual association key values.

Dynamic Normalization and DML Statements

Using the *tablename_association* and *tablename_mvcolumnname* constructs to access individual association rows in an NF² association is called *dynamic normalization*.

A DML statement that uses dynamic normalization can name only columns in the association, for example, the multivalued associated columns and the primary key columns of the base table.

Data read from or written to a dynamically normalized table must be singlevalued.

SELECT Statements

Data selected from a dynamically normalized table is presented to the client as singlevalued.

SELECT * FROM *tablename_association* gets all the association columns in one of the following ways:

- In the order specified in the @SELECT phrase in the dictionary
- If there is no @SELECT phrase, in the order specified in the @ phrase in the dictionary
- If there is no @ phrase, in the order specified by the CREATE TABLE statement that created the base table

Selection criteria in the WHERE clause are used to select association rows. WHEN clauses are not allowed with dynamic normalization because in effect the multivalued columns have been normalized into singlevalued columns.

INSERT and UPDATE Statements

Data written to *tablename_association* must be singlevalued.

If an INSERT statement does not specify a list of columns into which data is to be inserted, values must be supplied for all association columns (including the base table's primary key columns), in one of the following ways:

- In the order specified in the @INSERT phrase in the dictionary
- If there is no @INSERT phrase, in the order specified in the @ phrase in the dictionary

- If there is no @ phrase, in the order specified by the CREATE TABLE statement that created the base table

New association rows are inserted according to the positioning rule (INSERT FIRST, INSERT LAST, etc.) specified by the ASSOC clause of the CREATE TABLE or ALTER TABLE statement. Existing association rows cannot be assumed to be in sorted order.

Criteria in the WHERE clause of an UPDATE statement are used to select the association rows to be updated. WHEN clauses are not allowed with dynamic normalization.

DELETE Statements

DELETE FROM *tablename_association* deletes all association rows that meet the WHERE criteria.

Dynamic Normalization and Referential Integrity

Because of the interrelationships between the base table and the normalized *tablename_association* tables created from it by dynamic normalization, referential integrity must be maintained.

For example, if you delete a row in base table *tablename*, the change cascades to the related association rows in *tablename_association*. Likewise, if you update a primary key value in *tablename* to a different value, the change also cascades to the related association rows in *tablename_association*. If an INSERT statement on *tablename_association* tries to create a primary key that does not exist in the base table, or if a DELETE statement on *tablename_association* tries to delete a nonexistent primary key, the action is disallowed. This phenomenon is sometimes called *implicit* referential integrity because there is no code in the CREATE TABLE statement that explicitly causes it.

Dynamic Normalization and DDL Statements

The ALTER TABLE, DROP SCHEMA, DROP TABLE, DROP VIEW, GRANT, and REVOKE statements function normally when issued by a UCI program running in 1NF mode and behave as they do in NF² mode.

Calling and Executing UniVerse Procedures

What Can You Call as a UniVerse Procedure?	5-3
Processing UniVerse Procedure Results.	5-5
Print Result Set	5-5
Multicolumn Result Set	5-6
Affected-Row Count	5-6
Output Parameter Values	5-6
Processing Errors from UniVerse Procedures	5-7

This chapter describes how to call and execute procedures stored on a UniVerse data source.

Client programs can call and execute procedures that are stored on a database server. Procedures can accept and return parameter values and return results to the calling program.

Procedures let developers predefine database actions on the server. Procedures can provide a simple interface to users, insulating them from the names of tables and columns as well as from the syntax of SQL. Procedures can enforce additional database rules beyond simple referential integrity and constraints. Such rules need not be coded into each application that references the data, providing consistency and easy maintenance.

Procedures can provide a significant performance improvement in a client/server environment. Applications often have many steps, where the result from one step becomes the input for the next. If you run such an application from a client, it can take a lot of network traffic to perform each step and get results from the server. If you run the same program as a procedure, all the intermediate work occurs on the server; the client simply calls the procedure and receives a result.

What Can You Call as a UniVerse Procedure?

Typically you call a UniVerse BASIC subroutine as a procedure. You can also call a UniVerse BASIC program, a paragraph or stored sentence, a proc (ProVerb), a UniVerse command, or a remote command. You can call any of your existing programs, subroutines, and most of your existing paragraphs, stored sentences, and procs as procedures. You can call almost any UniVerse command as a procedure.

To call a UniVerse procedure, use `SQLExecDirect` or `SQLExecute` to execute a `CALL` statement. There are two formats of the `CALL` statement, one for calling UniVerse BASIC subroutines and the other for calling paragraphs, sentences, commands, programs, and procs.

If you call a UniVerse BASIC subroutine, you use the following `CALL` statement syntax, which lets you pass a comma-separated list of parameters within parentheses as arguments to the subroutine:

```
CALL procedure [(parameter [,parameter] ... )]
```

Parameters can be literals or parameter markers. The number and order of parameters must correspond to the number and order of arguments expected by the subroutine.

For example, to call subroutine `SUBX`, which requires a file name and a field name as arguments, you can use `SQLExecDirect` to execute a call statement such as:

```
CALL SUBX ('MYFILE','MYFIELD')
```

Or you could bind parameter number 1 to a program variable, load the desired field name into that variable, and execute:

```
CALL SUBX ('MYFILE',?)
```

The second format for the `CALL` statement is used to call a UniVerse BASIC program or a Universe command that accepts a string of arguments after the verb. In this case you use the standard UniVerse syntax after the procedure name, which lets you specify keywords, literals, and other tokens as part of the command line. You cannot use parameter markers with this syntax. You do not use parentheses, nor do you separate arguments with commas:

```
CALL procedure [argument [argument] ... ]
```

For example, to obtain a listing of the first three records in MYFILE, call the UniVerse LIST command by executing:

```
CALL LIST MYFILE SAMPLE 3
```

Processing UniVerse Procedure Results

The output of a procedure call, returned to the client application if the procedure executes successfully, consists of an SQL result and (optionally) output parameter values. The type and contents of these results are, of course, determined by the procedure itself.

An SQL result is either a set of fetchable rows (similar to what is returned by a SELECT statement) or a count of affected rows (similar to what is returned by an UPDATE statement). Usually the client application is written with the knowledge of what kind of results are produced by any procedure it calls, but if the client application does not know the nature of the procedure it is calling, the first thing it should do after executing the procedure is to call `SQLNumResultCols` to determine whether there is a fetchable result set. If there are any result columns the application can use `SQLColAttributes`, `SQLBindCol`, and `SQLFetch` to retrieve the results in the usual way.



Note: Information about the SQL result of a `CALL` statement is not available until after the statement has been executed. Therefore, if you `SQLPrepare` a `CALL` statement and then want to use `SQLNumResultCols`, `SQLColAttributes`, or `SQLRowCount`, you must first `SQLExecute` the statement. Otherwise the `SQLNumResultCols` (and so forth.) call receives a function sequence error (`SQLSTATE = S1010`).

Every call to a UniVerse procedure produces one of the following SQL results:

- Print result set
- Multicolumn result set
- Affected-row count

Print Result Set

One very common UniVerse result set is called a *print result set*. This is a one-column result set (`SQLNumResultCols` returns 1) whose rows are the lines of (screen) output produced by the called program, paragraph, command, or proc. The client application should use `SQLBindCol` to bind the one output column to a program variable, then use `SQLFetch` to return each print line into that variable.

Multicolumn Result Set

If the called procedure is a UniVerse BASIC subroutine containing SQL SELECT statements, the result set is called a *multicolumn result set* (SQLNumResultCols returns a positive integer). This result set comprises the fetchable rows produced by the last SELECT issued by the procedure before it exited. The client application should bind each output column to a program variable, then fetch the rows of output into those variables.

Affected-Row Count

If there are no result columns (SQLNumResultCols returns 0), the application can find out how many rows were affected in the database by calling SQLRowCount.

Output Parameter Values

In addition to an SQL result, some procedures return output in one or more output parameters. Before a client application calls such a procedure, it must use SQLBindParameter to indicate which parameters are output parameters and to assign a variable location for each. Then, after the procedure returns, the assigned variables contain the output values supplied by the procedure.

Processing Errors from UniVerse Procedures

The client application should always check the status of the `SQLExecute` or `SQLExecDirect` function used to execute a procedure call. If this status indicates an error, the application should use the `SQLError` function to obtain the `SQLSTATE`, UniVerse error code, and error message text that describe the error.

Calls to some UniVerse procedures return a status of `SUCCESS` even though the procedure encountered some kind of error. This is true for many procedures which produce a print result set (paragraphs, commands, procs, and some UniVerse BASIC programs). The client application might have to examine the contents of the print result set or display it for a user, in order to determine whether the procedure executed correctly. For example, suppose a client issues the following call:

```
CALL CREATE.INDEX MYFILE BADF
```

where `BADF` is not a valid field name in `MYFILE`. Execution of this call returns `SUCCESS` status, and the print result set contains the following error message produced by the UniVerse server when it tried to execute the `CREATE.INDEX` command:

```
Cannot find field name BADF in file dictionary or VOC,  
no index created.
```

How to Write a UniVerse Procedure

Using UniVerse Paragraphs, Commands, and Procs as Procedures	6-3
Writing UniVerse BASIC Procedures	6-4
Parameters Used by a UniVerse BASIC Procedure	6-4
SQL Results Generated by a UniVerse BASIC Procedure	6-5
Using @HSTMT in a UniVerse BASIC Procedure to Generate SQL Results	6-7
Using the @TMP File in a UniVerse BASIC Procedure	6-9
Errors Generated by a UniVerse BASIC Procedure.	6-12
Restrictions in UniVerse BASIC Procedures.	6-15
Fetching Rows and Closing @HSTMT Within a Procedure	6-15
Hints for Debugging a Procedure	6-16

A UniVerse procedure is a program that runs on a UniVerse server and can be called by UCI and BCI client applications. Client applications call a procedure by executing an SQL CALL statement. A UniVerse procedure can be any of the following:

- A UniVerse command
- A remote command
- A paragraph or stored sentence
- A proc (ProVerb)
- A UniVerse BASIC program
- A UniVerse BASIC subroutine

UniVerse BASIC programs, stored sentences and paragraphs, commands, and ProVerb procs that are defined in the VOC can always be called as procedures. UniVerse BASIC programs and subroutines should be locally, normally, or globally cataloged, although it is also possible to call a UniVerse BASIC program directly if the source code is stored in the BP file.

This chapter discusses the rules for using paragraphs, commands, and procs as procedures. It also discusses how to write UniVerse BASIC procedures including input and output parameters, result set generation, and the types of errors that can be produced by a UniVerse BASIC procedure.

Using UniVerse Paragraphs, Commands, and Procs as Procedures

You can call most UniVerse paragraphs, commands, and procs as procedures, as long as they conform to the following rules:

- If user input is required (if a paragraph contains the <<...>> syntax for inline prompting, for example), the input must be supplied by DATA statements.
- The paragraph, command, or proc cannot invoke a UniVerse menu.
- The paragraph, command, or proc cannot invoke any of the following UniVerse commands:

ABORT	MAKE	SREFORMAT
ABORT.LOGIN	MESSAGE	T.BCK
ANALYZE.SHM	NOTIFY	T.DUMP
AUTOLOGOUT	PASSWD	T.EOD
CALL	PHANTOM	T.FWD
CHDIR	Q	T.LOAD
CLEAN.ACCOUNT	QUIT	TRDLBL
GET.STACK	RADIX	T.READ
LO	RAID	T.REW
LOGON	REFORMAT	T.UNLOAD
LOGOUT	SAVE.STACK	T.WEOF
LOGTO	SET.REMOTE.ID	T.WTLBL
LOGTO.ABORT	SP.EDIT	UVFIXFILE
MAIL	SP.TAPE	VI

When a UniVerse paragraph, command, or proc is called as a procedure, all output lines that would ordinarily be sent to the terminal screen are stored in a special print file. These output lines make up what is called a *print result set*. After the procedure has finished executing, the calling client application can fetch the contents of the print result set, one line at a time, and process or display this output.



Note: The special print file used to store a print result set does not affect the behavior of print-capturing commands, such as *COMO* or *SPOOL*, that might be invoked by the paragraph, command, or proc.

Writing UniVerse BASIC Procedures

The most flexible and powerful UniVerse procedures are written as UniVerse BASIC programs, usually subroutines.

UniVerse BASIC procedures should be compiled and cataloged (locally, normally, or globally). If a UniVerse BASIC procedure is uncataloged, it can be called if it is in the BP directory of the account to which the client application is connected.

The writer of a UniVerse BASIC procedure should specify its characteristics so that client application programmers know how to call the procedure and what results it will return. These characteristics should include:

- The number of parameters to be used when calling the procedure
- The definition of each parameter as input, input/output, or output
- The nature of data to be supplied in input and input/output parameters
- The type of SQL result generated (print result set, multicolumn result set, or affected-row count)
- For a multicolumn result set, how many columns are returned
- The name, data type, and so forth, of each column in a multicolumn result set
- The types of SQL errors that may be generated
- For each error type, what SQLSTATE and error code are returned

Parameters Used by a UniVerse BASIC Procedure

The SUBROUTINE statement at the beginning of a UniVerse BASIC subroutine procedure determines how many input and output parameters it requires. The calling client application program must supply the same number of parameters (or parameter markers for output parameters) in the same order as they are expected by the procedure.

For example, a UniVerse BASIC procedure that takes one input parameter (employee number) and returns one output parameter (person's name) might be coded roughly as follows:

```
SUBROUTINE GETNAME (EMPNO, PERSON)
OPEN "EMPS" TO EMPS ELSE...
READ INFO FROM EMPS, EMPNO ELSE...
PERSON = INFO<1>
RETURN
```

A client application would call this procedure with program logic such as the following:

1. SQLBindParameter: Define parameter marker 1 as an output parameter which is bound to variable NAME.
2. SQLExecDirect: CALL GETNAME(4765,?)
3. Check status for error.
4. If no error, the name of employee 4765 is now in NAME.

***Note:** A UniVerse BASIC procedure need not define any parameters. An application that calls a procedure with no parameters should not specify any parameter values or parameter markers in its call.*

SQL Results Generated by a UniVerse BASIC Procedure

Every call to a UniVerse procedure returns one of the following types of SQL result:

- Print result set
- Multicolumn result set
- Affected-row count

This section discusses how the programming of a UniVerse BASIC procedure determines which type of SQL result it produces.

All output lines that would normally be sent to the terminal screen during the execution of a procedure are stored in a special print file; in the case of a UniVerse BASIC procedure, this would, of course, include any PRINT statements issued by the UniVerse BASIC program. The contents of this special print file will become a one-column print result set unless the procedure overrides this default behavior and forces one of the other types of SQL result.



The functionality of client/server procedure calls is greatly enhanced by having the ability to write procedures that generate multicolumn result sets or affected-row counts instead of print result sets. Some of the advantages are:

- If a multicolumn result set is generated, output results are delivered into separate program variables in the calling client. There is no need for the client to scan each output line and extract individual items of information.
- The full power of the SQL query language and query optimizer can be used in a procedure. For example:
 - Output rows can be generated from SQL joins, subqueries, unions, and grouping queries.
 - Output columns can be defined using SQL functions and expressions.
 - Multivalued data can be dynamically normalized and returned as singlevalued data.
- INSERT, UPDATE, and DELETE statements can be used in a procedure to modify the database, returning an affected-row count to the caller.
- Data definition statements such as CREATE TABLE, ALTER TABLE, CREATE VIEW, and GRANT can be executed within a procedure.
- The power of SQL can be combined with the flexibility of UniVerse BASIC to perform almost any desired function in a callable UniVerse procedure. This centralizes complex business logic, simplifies the writing of client applications, and reduces network traffic in a client/server environment.

Procedures generate multicolumn result sets and affected-row counts by executing SQL statements using the `@HSTMT` variable. These are discussed in the following two sections.



Note: *@HSTMT is the only variable that can be used to generate a multicolumn result set or an affected-row count. Other variables can be allocated and used within a procedure, but their results are strictly internal to the procedure.*

Using @HSTMT in a UniVerse BASIC Procedure to Generate SQL Results

UniVerse BASIC procedures running on a UniVerse server can use the preallocated variable @HSTMT to execute programmatic SQL statements. If any SQL statements are executed in this way, the results from the last such statement to be executed become the SQL result that is returned to the calling client application. This result, which can be either a multicolumn result set, an affected-row count, or an SQL error, overrides the default print result set.

The following sample server and client programs show how to use procedures to simplify a client program's access to the numbers and names of employees in various departments. The procedures use a table called EMPS, whose key column is EMPNUM and whose data columns are EMPNAME and DEPNUM.

Procedure

This UniVerse BASIC subroutine, SHOWDEPT, uses the @HSTMT variable to execute a SELECT statement on the server. The SELECT statement returns a multicolumn result set containing employee numbers and names from the EMPS table.

```
SUBROUTINE SHOWDEPT(DEPT)
$INCLUDE UNIVERSE.INCLUDE ODBC.H
SELSTMT = "SELECT EMPNUM, EMPNAME FROM EMPS WHERE DEPNUM=:DEPT
ST = SQLExecDirect(@HSTMT, SELSTMT)
RETURN
```

Client Program

The following fragment of a BCI client program, LIST.EMPLOYEES, calls the SHOWDEPT subroutine as a procedure (the same could be done with a UCI client program):

```
.
.
.
PRINT "ENTER DEPT NUMBER"
INPUT DEPTNO
ST=SQLBindParameter(HSTMT, 1, SQL.B.BASIC, SQL.INTEGER, 4, 0,
DEPTNO)
ST=SQLExecDirect(HSTMT, "CALL SHOWDEPT(?)")
ST=SQLBindCol(HSTMT, 1, SQL.B.NUMBER, EMPNO)
ST=SQLBindCol(HSTMT, 2, SQL.B.CHAR, NAME)
LOOP
```



```

        WHILE SQL.SUCCESS = SQLFetch(HSTMT) DO
        PRINT EMPNO '4R' : " " : NAME
    REPEAT
    .
    .
    .

```

Sample Output

When the client program runs, output such as the following appears on the terminal screen:

```

>RUN BP LIST.EMPLOYEES
ENTER DEPT NUMBER
?123
4765 John Smith
2109 Mary Jones
 365 Bill Gale
.
.
.

```

Procedure

This UniVerse BASIC subroutine, FIXDEPT, uses the @HSTMT variable to execute an UPDATE statement on the server, which changes the department number in the EMPS table for all employees in a particular department:

```

SUBROUTINE FIXDEPT(OLDDEPT,NEWDEPT)
$INCLUDE UNIVERSE.INCLUDE ODBC.H
UPDSTMT = "UPDATE EMPS SET DEPNUM = ":NEWDEPT
UPDSTMT := " WHERE DEPNUM = ":OLDDEPT
ST=SQLExecDirect(@HSTMT, UPDSTMT)
RETURN

```

Client Program

The following fragment of a BCI client program, CHANGE.DEPT, calls the FIXDEPT subroutine as a procedure (the same could be done with a UCI client program):

```

.
.
.
PRINT "ENTER OLD DEPT NUMBER: " : ; INPUT OLD
PRINT "ENTER NEW DEPT NUMBER: " : ; INPUT NEW
ST = SQLExecDirect(HSTMT, "CALL FIXDEPT(":OLD:",":NEW:")")
IF ST = 0 THEN

```

```

        ST = SQLRowCount(HSTMT,ROWS)
        PRINT "Department number ":OLD:" has been changed to ":NEW:
        PRINT " for ":ROWS:" employees."
    END ELSE
        PRINT "The EMPS table could not be updated."
    END
.
.
.

```

Sample Output

When the client program runs, output such as the following bold appears on the terminal screen:

```

>RUN BP CHANGE.DEPT
ENTER OLD DEPT NUMBER: ?901
ENTER NEW DEPT NUMBER: ?987
Department number 901 has been changed to 987 for 45 employees.

```

Using the @TMP File in a UniVerse BASIC Procedure

It is relatively easy for a procedure to produce a multicolumn result set when the data to be returned is already in an existing file, as shown in the examples above. But there are situations in which you want a procedure to return multicolumn output that is created on the fly, from a variety of sources, perhaps using complex calculations. It might be much easier to generate this data by programming in UniVerse BASIC than by using some complex SQL join or union with SQL expressions. To accommodate this kind of situation, UniVerse BASIC procedures can use a virtual file called @TMP.

The general mechanism for using @TMP consists of three steps:

1. Generate the desired data as a dynamic array (referred to below as DARRAY), using field marks as “row” separators and text marks as “column” separators.
2. Save the dynamic array as a select list.
3. Execute an SQL SELECT from @TMP, using the select list as input.

When the SQL SELECT is executed, the virtual @TMP file appears to have a number of rows equal to the number of “rows” in DARRAY. The SQL SELECT can reference virtual fields in @TMP named F1, F2, F3, ..., F23, which represent up to 23 text-mark-separated “columns” in DARRAY. The @TMP file also appears to have an @ID field containing the entire contents of each “row” in DARRAY (the length of each “row” is not subject to the 255-character limit usually associated with @ID in UniVerse files).

The virtual @TMP file can be used in any SQL SELECT statement, including joins and unions. @TMP cannot be referenced with INSERT, UPDATE, or DELETE statements, however.

The use of @TMP is illustrated in the following example. A client application calls a UniVerse BASIC procedure to obtain a list of employees whose department is located in New Hampshire, along with their department number and zip code, sorted by department number. The EMPS table does not indicate which state and zip code each department is located in; this information is determined from a list in the procedure program itself.

Procedure

This UniVerse BASIC subroutine FINDEMPS builds a dynamic array consisting of department number, zip code, and employee name for each employee who works in a specified state. It then saves this dynamic array in select list 9, and uses the @HSTMT variable to execute an SQL SELECT from the virtual @TMP file specifying select list 9 as the source of the data. The SELECT statement contains an ORDER BY clause to sort the output by department number.

```
SUBROUTINE FINDEMPS(INSTATE) ; * Returns dept, zip code, name
sorted
      by dept
$INCLUDE UNIVERSE.INCLUDE ODBC.H
DARRAY = ""
OPEN "EMPS" TO FVAR ELSE PRINT "OPEN ERROR" ; RETURN
SELECT FVAR
LOOP
READNEXT EMPNUM THEN
  READ EMPREC FROM FVAR,EMPNUM ELSE PRINT "READ ERROR" ; RETURN
  NAME = EMPREC<1> ; * EMPREC field 1 contains employee name
  DEPT = EMPREC<2> ; * EMPREC field 2 contains department number
  GOSUB GETSTATE ; * GETSTATE (not shown) returns STATE & ZIP for
this
      DEPT
IF STATE = INSTATE THEN
  IF DARRAY <> "" THEN DARRAY := @FM
  DARRAY := DEPT:@TM:ZIP:@TM:NAME ;* Add 1 "row" with 3
```

```

"columns" to
        DARRAY
    END
END ELSE EXIT
REPEAT
SELECT DARRAY TO 9 ; * Save DARRAY in select list 9
ST=SQLExecDirect(@HSTMT, "SELECT F1,F2,F3 FROM @TMP SLIST 9 ORDER
BY 1")
RETURN

```

Client Program

The following fragment of a BCI client program EMPS.IN.STATE calls the FINDEMPS subroutine as a procedure (the same could be done with a UCI client program):

```

.
.
.
PRINT "ENTER STATE: ": ; INPUT SSS
ST = SQLExecDirect(HSTMT, "CALL FINDEMPS('":SSS:"'")
IF ST = 0
THEN
    ST = SQLBindCol(HSTMT, 1, SQL.B.NUMBER, DEPTNO)
    ST = SQLBindCol(HSTMT, 2, SQL.B.NUMBER, ZIPCODE)
    ST = SQLBindCol(HSTMT, 3, SQL.B.CHAR, EMPNAME)
    LOOP
        WHILE SQL.SUCCESS = SQLFetch(HSTMT) DO
            PRINT DEPTNO '4R' : " ":ZIPCODE '5R%5' : " ":EMPNAME
        REPEAT
    END
.
.
.

```

Sample Output

When the client program runs, output such as the following appears on the terminal screen:

```

>RUN BP EMPS.IN.STATE
ENTER STATE: ?NH
529 03062 Ann Gale
529 03062 Fred Pickle
987 03431 John Kraneman
989 03101 Edgar Poe
.
.
.

```

Errors Generated by a UniVerse BASIC Procedure

When a client application calls a procedure, several types of output results can be returned to the caller. But a procedure can also generate an SQL error instead of normal output results. If an error is generated, the calling client application should detect this by testing the status returned from its `SQLExecDirect` or `SQLExecute` function call, getting `SQL ERROR (-1)` instead of `SQL SUCCESS (0)`.

A UniVerse BASIC procedure can generate an SQL error either indirectly (by issuing an SQL statement that causes an error) or directly (by using the UniVerse BASIC `SetDiagnostics` function).

If the last SQL statement issued (using `@HSTMT`) within the procedure before it returns to the caller encountered an error, that error condition is passed back to the calling client application, as shown in the following example.

Procedure

This procedure `ADDEMP` can be called to add a new employee to the `EMPS` table:

```
SUBROUTINE ADDEMP (NEWNUM, NEWNAME, NEWDEPT)
$INCLUDE UNIVERSE.INCLUDE ODBC.H
INSSTMT = "INSERT INTO EMPS VALUES (":NEWNUM
INSSTMT := ", '":NEWNAME: "','":NEWDEPT:");"
ST=SQLExecDirect (@HSTMT, INSSTMT)
RETURN
```

Client Program

The following fragment of a BCI client program `NEW.EMPLOYEE` calls the `ADDEMP` subroutine as a procedure, providing information about a new employee but erroneously assigning him an existing employee number (the same could be done with a UCI client program):

```
.
.
.
EMPNO = 2109
FIRSTLAST = "Cheng Du"
DEPNO = 123
CALLSTMT = "CALL ADDEMP (":EMPNO
CALLSTMT := ", '":FIRSTLAST
CALLSTMT := "','":DEPNO:");"
PRINT "The CALL statement is: ":CALLSTMT
ST = SQLExecDirect (HSTMT, CALLSTMT)
IF ST <> 0 THEN
```

```

        ERST =
        SQLError(SQL.NULL.HENV,SQL.NULL.HDBC,HSTMT,STATE,CODE,MSG)
        PRINT "SQLSTATE = ":STATE:", UniVerse error code = ":CODE:",
            Error text ="
        PRINT MSG
    END
    .
    .
    .

```

Sample Output

When the client program runs, output such as the following appears on the terminal screen:

```

>RUN BP NEW.EMPLOYEE
The CALL statement is: CALL ADDEMP (2109,'Cheng Du',123);
SQLSTATE = S1000, UniVerse error code = 950060,
    Error text = [IBM] [SQL Client] [UNIVERSE]UniVerse/SQL:
    Attempt to insert duplicate record "2109" is illegal.

```

A procedure can force an error condition to be returned by using the UniVerse BASIC SetDiagnostics function. This function sets a procedure-error condition and stores error text (supplied by the procedure) in the SQL diagnostics area associated with @HSTMT. The error condition remains in effect until the next programmatic SQL statement, or SQLClearDiagnostics, is issued. In particular, the error condition will be detected by the calling client application if the procedure returns before issuing another SQL statement.

The use of SetDiagnostics to generate a procedure error condition is illustrated in the following example.

Procedure

This procedure DELEMP can be called to delete an employee from the EMPS table:

```

SUBROUTINE DELEMP(OLDNUM)
    OPEN "EMPS" TO FVAR ELSE PRINT "OPEN ERROR" ; RETURN
    READU REC FROM FVAR,OLDNUM THEN
        DELETE FVAR,OLDNUM
    END ELSE
        JUNK = SetDiagnostics("Employee ":OLDNUM:" does not exist")
    END
    RETURN

```

Client Program

The following fragment of a BCI client program RESIGNATION calls the DELEMP subroutine as a procedure, asking it to delete an employee but providing an incorrect employee number (the same could be done with a UCI client program):

```
.
.
.
EMPNO = 555
ST = SQLExecDirect(HSTMT, "CALL DELEMP (":EMPNO:")")
IF ST <> 0 THEN
  ERST =
  SQLError(SQL.NULL.HENV,SQL.NULL.HDBC,HSTMT,STATE,CODE,MSG)
  PRINT "SQLSTATE = ":STATE:", UniVerse error code = ":CODE:",
      Error text ="
  PRINT MSG
END
.
.
.
```

Sample Output

When the client program runs, output such as the following appears on the terminal screen:

```
>RUN BP RESIGNATION
SQLSTATE = S1000, UniVerse error code = 950681, Error text =
[IBM] [SQL Client] [UNIVERSE]Employee 555 does not exist
```

Restrictions in UniVerse BASIC Procedures

Several restrictions must be observed when writing a UniVerse BASIC procedure:

- A procedure must not invoke any of the UniVerse commands listed in [“Using UniVerse Paragraphs, Commands, and Procs as Procedures”](#) on page 3.
- A procedure must not pause for user input; for example, if any INPUT statements are executed, the input must be provided by DATA statements.
- A procedure must not execute any (nested) procedure CALL statements using the @HSTMT variable. Nested procedure calls are allowed only if a different variable is used.

Fetching Rows and Closing @HSTMT Within a Procedure

If a UniVerse BASIC procedure executes an SQL SELECT using @HSTMT, the procedure can process the results itself (just like any other UniVerse BASIC program) using any of the following function calls:

- SQLNumResultCols
- SQLDescribeCol
- SQLColAttributes
- SQLBindCol
- SQLFetch

If a procedure fetches some of the rows in a SELECT's result set and then returns to the calling client application, the remaining rows (but not the fetched rows) are available for the client to fetch.

If a procedure executes an SQL SELECT, fetches some rows and decides not to return the remaining rows to the client, it should close the @HSTMT variable:

```
ST = SQLFreeStmt (@HSTMT, SQL.CLOSE)
```

It is also necessary to close @HSTMT if the procedure wants to execute another SQL statement using @HSTMT. Closing @HSTMT discards any pending results and reinitializes the cursor associated with @HSTMT.

At the time a procedure exits, if @HSTMT has been closed and not reused, and if SetDiagnostics has not been issued, then a print result set is returned to the caller. If the procedure executes no PRINT statements, the print result set contains no rows.

Hints for Debugging a Procedure

If a procedure does not produce the expected results, try the following:

- Ensure that both the procedure and the calling client application check the status returned by each SQL Client Interface function call (SQLExecDirect, SQLFetch, and so on).

- Comment out the SQL Client Interface function calls in the procedure, or close @HSTMT before exiting, so that the print results are returned to the client; if necessary, add diagnostic PRINT statements to the procedure program.
- Debug UniVerse BASIC programs and subroutines by running them directly, before calling them from a client application.

Data Types

Data Types and Data Type Coercion.	7-3
C Data Types Supported	7-3
SQL Data Types Supported	7-9
Data Type Coercion	7-10

This chapter is a reference for the data types supported by UCI.

Data Types and Data Type Coercion

UCI lets you specify how the application program converts data from the database. This section covers the C data types and the SQL data types supported by UCI, and the data coercion performed during data conversion.

In most instances of retrieving data from the data source and storing it in a C structure, the SQL data type source is compatible with the C data type, and no data coercion (conversion) is required. For instance, an SQL_CHAR data type can be stored directly into a C string. However, if the SQL data type of the source is not compatible with the C data type, the data is coerced (converted) into a comparable form. For instance, if an SQL VARCHAR value is stored in a numeric C field such as SQL_C_FLOAT, UCI tries to convert the source data to numeric.

C Data Types Supported

UCI supports all core C data types and some extended C data types from ODBC 2.0, as shown in the following table. These application data types are used in **SQLBindCol**, **SQLBindMvCol**, **SQLGetData**, **SQLBindParameter**, **SQLBindMvParameter**, and **SQLSetParam**.

UCI Definition	C Data Type	Comments
SQL_C_CHAR	unsigned char	The distinction between SQL_C_CHAR and SQL_C_STRING is that the SQL_C_CHAR data type is presumed to be a null-terminated string while SQL_C_STRING is not. The native database STRING type corresponds to the SQL_C_STRING data type.
SQL_C_TINYINT	char	
SQL_C_STINYINT	char	
SQL_C_UTINYINT	unsigned char	
SQL_C_SHORT	short	
SQL_C_SSHORT	short	

Supported C Data Types

UCI Definition	C Data Type	Comments
SQL_C_USHORT	unsigned short	
SQL_C_LONG	int	32 bits
SQL_C_SLONG	int	32 bits
SQL_C_ULONG	int	32 bits
SQL_C_FLOAT	float	
SQL_C_DOUBLE	double	
SQL_C_STRING	struct { UDWORD <i>len</i> ; UCHAR* <i>text</i> ; }	See Comments for SQL_C_CHAR.
SQL_C_TIME	struct { UWORD <i>hour</i> ; UWORD <i>minute</i> ; UWORD <i>second</i> ; }	
Supported C Data Types (Continued)		

UCI Definition	C Data Type	Comments
SQL_C_DATE	<pre>struct { SWORD <i>year</i>; UWORD <i>month</i>; UWORD <i>day</i>; }</pre>	
C_ARRAY	<pre>struct { UWORD <i>cDcount</i>; UWORD <i>cStorage</i>; SWORD <i>fCType</i>; SWORD <i>fSqlType</i>; SWORD <i>fParamType</i>; SWORD <i>ibscale</i>; UDWORD <i>cbColDef</i>; UCIDATUM Data[1]; }</pre>	A special data type reserved for use with multivalued columns in the SQLBindMvParameter and SQLBindMvCol calls. It cannot be used with SQLBindParameter and SQLBindCol.
UCI_DATUM	<pre>struct { SDWORD <i>fIndicator</i>; union <i>uValue</i>; }</pre>	Outlined here only to detail the SQL_C_ARRAY structure. See C Data Type Representation of Multivalued Columns on page 8 for details of the <i>uValue</i> union. UCI_DATUM is analogous to the SQL_C_STRING data type, but, unlike SQL_C_STRING, it can be used for an arbitrary C data type through its union <i>uValue</i>. UCI_DATUM cannot be used directly in any UCI calls.

Supported C Data Types (Continued)

The unsupported C data types are:

- SQL_C_BIT
- SQL_C_BINARY
- SQL_C_TIMESTAMP
- SQL_C_BOOKMARK

C Data Types and Database Internal/External Formats

UniVerse and UniData maintain data in a specific format and provide mapping functions known as conversions to convert this internal format into the external format expected by the application.

By default, UCI server returns, for a bound column, a stripped external format. For example, if a money column has a conversion code of MD2\$, the database internally stores the value \$4.50 as the integer value 450. UCI returns this value to the application as 4.50, that is, the correct value numerically, but stripped of all text formatting such as currency symbols.

Also by default, dates and times are returned as C structures that preserve the full informational content of those data types. An application can obtain dates and times in internal format by coercing them as integers (refer to [“Data Type Coercion”](#) on page 10) so that it can manipulate them arithmetically.

When converting data to C data types, be aware that the database supports string math and can operate on numbers that cannot be mapped into standard C data types.

UniVerse and UniData store all data as text strings, and any attempt to convert database numerics that exceed the limits of a C numeric data type (as specified by the *fCType* parameter in an SQLBindCol call) will fail when fetching data from the server. However, numerics can be bound as SQL_C_STRING or SQL_C_CHAR to reduce the possibility of conversion failure.

Empty Strings

Data of all types frequently contains empty strings. If a column contains an empty string (that is, the whole field in a singlevalued column is an empty string, or a singlevalue in a multivalued column is an empty string), the value is returned as follows:

Data Type	Value Returned
CHAR STRING	Zero-length string
LONG SHORT BYTE DOUBLE FLOAT	0
DATE	SQL_BAD_DATA 0 year, 0 month, 0 day
TIME	SQL_BAD_DATA 0 hour, 0 minute, 0 second

Empty String Return Values

As the table shows, an empty string maps to 0 for numeric data types and to a zero-length string for nonnumeric data types. Because an empty string in a DATE or TIME field cannot be mapped logically to a reasonable date or time, zeros are returned along with a return value of SQL_SUCCESS, and the *pcbValue* of SQLBindCol and SQLGetData is set to SQL_BAD_DATA.

You may want your client program to return empty string data from the data source as null values, and to convert null values to empty strings when inserting or updating data on the data source. To do this, do the following:

- Add an X-descriptor called @EMPTY.NULL to the dictionary of the table or file. The only data in the descriptor should be an X in field 1.
- In your client program, set the SQL_EMPTY_NULL option of the **SQLSetConnectOption** function in your client program to SQL_EMPTY_NULL_ON.

C Data Type Representation of Multivalued Columns

UCI supports a structure called `C_ARRAY`, which holds data read into a bound multivalued column and writes it back through a parameter for a multivalued column. This structure provides a natural C data type mapping of a dynamic array:

```
typedef struct tagC_ARRAY
{
    UWORD cDcount;      /* count of the number of values in Data
array */
    UWORD cStorage; /* size of Data array for which memory was
allocated */
    SWORD fCType;      /* the C data type pointed to by Data array
*/
    SWORD fSqlType;    /* the SQL data type of the columns for
parameters */
    SWORD fParamType; /* input only */
    SWORD ibScale;     /* not currently used by the database*/
    UDWORD cbColDef;   /* not currently used by the database*/
    UCI_DATUM Data[1]; /* array of UCI_DATUMs, one for each value
*/
} C_ARRAY

typedef struct tagUCI_DATUM
{
    SDWORD fIndicator; /* set to SQL_NULL_DATA to indicate null
value */
    union
    {
        double    dbl;      /* for SQL_C_DOUBLE */
        float     flt;      /* for SQL_C_FLOAT */
        SCHAR     sbyte;    /* for SQL_C_TINYINT and
SQL_C_STINYINT */
        UCHAR     ubyte;    /* for SQL_C_UTINYINT */
        SWORD     sword;    /* for SQL_C_SHORT and SQL_C_SSHORT
*/
        UWORD     uword;    /* for SQL_C_USHORT */
        SDWORD    sdword;   /* for SQL_C_LONG and SQL_C_SLONG */
        UDWORD    udword;   /* for SQL_C_ULONG */
        STRING    string;   /* for SQL_C_STRING and SQL_C_CHAR +
length */
        TIME_STRUCT time;   /* for SQL_C_TIME */
        DATE_STRUCT date;   /* for SQL_C_DATE */
    } uValue
} UCI_DATUM;
```

SQL Data Types Supported

UCI recognizes all minimum and core SQL data types from ODBC 2.0, plus dates and times, as shown in the following table.

SQL Data Type	C Application Type	UniVerse SQL Data Type
SQL_CHAR	SQL_C_STRING	CHAR[ACTER]
SQL_VARCHAR	SQL_C_STRING	VARCHAR[ACTER]
SQL_DECIMAL	SQL_C_DOUBLE	DEC[IMAL]
SQL_NUMERIC	SQL_C_DOUBLE	NUMERIC
SQL_INTEGER	SQL_C_SLONG	INT[EGER]
SQL_SMALLINT	SQL_C_SLONG	INT[EGER]
SQL_REAL	SQL_C_FLOAT	REAL
SQL_FLOAT	SQL_C_DOUBLE	FLOAT
SQL_DOUBLE	SQL_C_DOUBLE	DOUBLE PRECISION
SQL_DATE	SQL_C_DATE	DATE
SQL_TIME	SQL_C_TIME	TIME

SQL Data Types

A column with a conversion code beginning with D is assumed to be an SQL_DATE type. A column with a conversion code beginning with MT is assumed to be an SQL_TIME type.

Data Type Coercion

Data returned by the server can be coerced into other data types, as shown in the following table. UCI tries to perform all data type coercions sensibly, except for date-to-time and time-to-date, which cannot be performed in any logical way.

SQL Data Type	Target C Data Type	Comments
SQL_CHAR, SQL_VARCHAR	SQL_C_CHAR, SQL_C_STRING	Application is responsible if the data contains the number 0.
	SQL_C_TINYINT	Converted to numeric if possible.
	SQL_C_SHORT	Converted to numeric if possible.
	SQL_C_LONG	Converted to numeric if possible.
	SQL_C_DOUBLE	Converted to numeric if possible.
	SQL_C_FLOAT	Converted to numeric if possible.
SQL_DATE	SQL_C_SHORT	Returns internal format.
	SQL_C_LONG	Returns internal format.
	SQL_C_CHAR	Returns a null-terminated string in ISO format.
	SQL_C_STRING	Returns a string of length 10 in ISO format.
SQL_TIME	SQL_C_SHORT	Returns UniVerse internal format of time.
	SQL_C_LONG	Returns UniVerse internal format of time.
	SQL_C_CHAR	Returns a null-terminated string in ISO format.
	SQL_C_STRING	Returns a string of length 8 in ISO format.
SQL_SMALLINT, SQL_INTEGER	SQL_C_CHAR	Returns a null-terminated string.
	SQL_C_STRING	Returns a string with length.

Data Type Coercions

SQL Data Type	Target C Data Type	Comments
SQL_REAL, SQL_FLOAT, SQL_DECIMAL, SQL_DOUBLE, SQL_NUMERIC	SQL_C_CHAR	Returns a null-terminated string.
	SQL_C_STRING	Returns a string with length.

Data Type Coercions (Continued)

Parameter Coercion

If a numeric C-type parameter is bound to a database column of type SQL_DATE or SQL_TIME, the numeric value is coerced to a date as the number of days before or since December 31, 1967, or to a time as the number of seconds since midnight, respectively. This is the reverse conversion to that performed when a numeric C data type is bound to a column of type SQL_DATE or SQL_TIME for fetching data.

UCI supports SQL_C_DATE in the range 0001-01-01 through 12-31-9999, and SQL_C_TIME in the range 00:00:00 through 23:59:59. UCI does not perform range checking on parameters for columns of numeric data types; that is, UCI lets you store a value in an SQL_REAL column from a bound SQL_C_DOUBLE parameter that exceeds the range of an SQL_C_FLOAT type. Also, UCI does not check the length of bound parameters of types SQL_CHAR and SQL_VARCHAR because UCI takes full advantage of the database's ability to store data of any length in any column.

Generally, you should use a data type of SQL_C_DOUBLE for approximate numerics. If precision is less important than memory, you can use SQL_C_FLOAT in the range from 1.17549e-38 through 3.402823e+38.

Coercing approximate numeric SQL data types into integer C data types is legal, but in cases where the approximate numeric contains a fractional part, UCI truncates the fractional part and returns the integer part. It also returns SQL_SUCCESS_WITH_INFO, indicating that one or more columns of data were truncated. If the integer part of the approximate number is too large to fit into the designated C data type, UCI returns SQL_ERROR.

Precision, Scale, and Display Size Definitions

The calls SQLDescribeCol and SQLColAttributes allow an application to determine the database values for precision, scale, display size, and other qualities. UCI always ignores scale and precision when binding parameters and columns.

When binding columns, a data truncation error is issued after an SQLFetch call if, for example, the server returns a value of 257 for a column bound to an SQL_C_TINYINT. But that sort of error is based on the actual data returned for a particular row, not on the column's precision and scale.

Tables

The following table shows the precision, scale, and display size for supported SQL data types for columns in UniVerse and UniData tables:

SQL Data Type	Precision	Scale ^a	Display Size ^b
SQL_CHAR	From the definition. If not defined, precision is 1.	0	Same as precision.
SQL_VARCHAR	From the definition. If not defined, precision is 254.	0	10
SQL_SMALLINT	5	0	10
SQL_INTEGER	10	0	10
SQL_REAL	7	0	10
SQL_FLOAT	15	0	16
SQL_DOUBLE	15	0	30
SQL_DECIMAL, SQL_NUMERIC	From the definition. If not defined, precision is 9.	<i>See footnote 1</i>	<i>precision + 2</i>
SQL_DATE	10 (yyyy-mm-dd)	0	11
SQL_TIME	8 (hh:mm:ss)	0	8

a. The scale of an SQL_DECIMAL or SQL_NUMERIC data type comes from the column's definition; if it is not defined, the scale is 0.

b. Any FORMAT specification overrides the defaults shown in the table.

UniVerse Files

The following table shows the precision, scale, and display size for supported SQL data types for fields in database files:

SQL Data Type	Precision	Scale ^a	Display Size ^b
SQL_CHAR	From SQLTYPE or FORMAT	0	10
SQL_VARCHAR	From SQLTYPE or FORMAT	0	10
SQL_SMALLINT	5	0	10
SQL_INTEGER	10	0	10
SQL_REAL	7	0	10
SQL_FLOAT	15	0	10
SQL_DOUBLE	15	0	10
SQL_DECIMAL, SQL_NUMERIC	From SQLTYPE or FORMAT	<i>See footnote 1</i>	10
SQL_DATE	10 (<i>yyyy-mm-dd</i>)	0	10
SQL_TIME	8 (<i>hh:mm:ss</i>)	0	10

- a. The scale of an SQL_DECIMAL or SQL_NUMERIC data type comes from the column's definition; if it is not defined, the scale is 0.
- b. Any FORMAT specification overrides the defaults shown in the table.

Expressions

The following table shows the precision, scale, and display size for supported SQL data types for expressions in database tables and files. By default, expressions use only the following data types:

SQL Data Type	Precision	Scale	Display Size ^a
SQL_VARCHAR	Computed. If not computed, precision is 254.	0	Computed. If not computed, 254.
SQL_INTEGER	10	0	11
SQL_DOUBLE	15	0	22
SQL_DATE	10	0	10
SQL_TIME	8	0	8

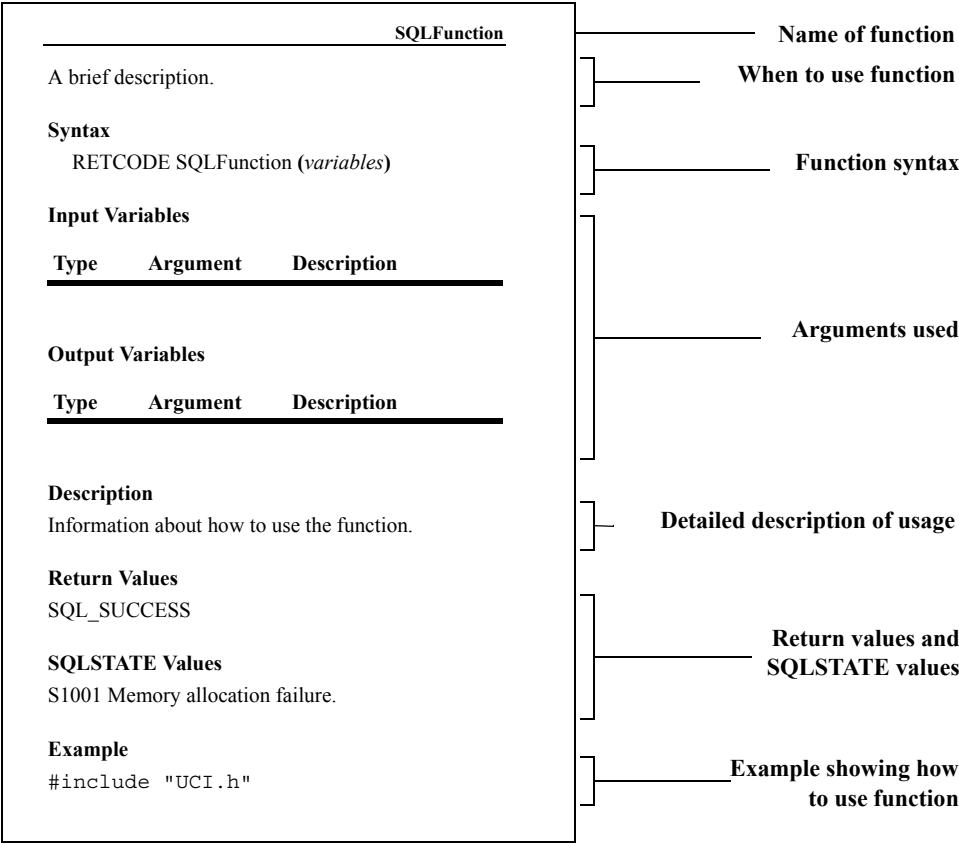
a. Any FORMAT specification overrides the defaults shown in the table.

UCI Functions

Function Call Summary	8-4
Variables	8-5
Search Patterns	8-6
Return Values	8-7
Error Codes	8-7
Use of Hungarian Naming Conventions	8-8
Functions	8-10
SQLAllocConnect	8-11
SQLAllocEnv	8-13
SQLAllocStmt	8-15
SQLBindCol	8-17
SQLBindMvCol	8-22
SQLBindMvParameter	8-25
SQLBindParameter	8-27
SQLCancel	8-32
SQLColAttributes	8-34
SQLColumns	8-40
SQLConnect	8-44
SQLDataSources	8-48
SQLDescribeCol	8-51
SQLDisconnect	8-54
SQLError	8-56
SQLExecDirect	8-59
SQLExecute	8-63
SQLFetch	8-65
SQLFreeConnect	8-68
SQLFreeEnv	8-70

SQLFreeMem	8-72
SQLFreeStmt	8-73
SQLGetData	8-75
SQLGetFunctions	8-79
SQLGetInfo	8-83
SQLNumParams	8-91
SQLNumResultCols	8-93
SQLParamOptions	8-95
SQLPrepare	8-98
SQLRowCount	8-102
SQLSetConnectOption	8-104
SQLSetParam	8-110
SQLTables	8-112
SQLTransact	8-116
SQLUseCfgFile	8-120

This chapter is a reference for UCI function calls, listed in alphabetical order. The following diagram illustrates a typical function reference page.



Function Call Summary

The following table lists UCI ODBC function calls according to how they are used.

Use	Functions
Initializing	SQLAllocConnect SQLAllocEnv SQLAllocStmt SQLConnect SQLPrepare SQLSetConnectOption SQLUseCfgFile
Exchanging data	SQLBindCol SQLBindMvCol SQLBindMvParameter SQLBindParameter SQLColAttributes SQLColumns SQLDataSources SQLDescribeCol SQLExecDirect SQLExecute SQLFetch SQLGetData SQLGetFunctions SQLGetInfo SQLNumParams SQLNumResultCols SQLParamOptions SQLRowCount SQLSetParam SQLTables SQLTransact

Functions and Their Uses

Use	Functions
Memory management	SQLFreeMem
Processing errors	SQLError
Disconnecting	SQLCancel SQLDisconnect SQLFreeConnect SQLFreeEnv SQLFreeStmt

Functions and Their Uses (Continued)

Variables

In the following syntax the variable *henv* is the environment handle returned from `SQLAllocEnv`, and the variable *phdbc* is a pointer to where the connection handle is to be stored. Names of return variables, input variables, and output variables are user-defined.

RETCODE `SQLAllocConnect` (*henv*, *phdbc*)

All calls use handles that represent a pointer to an underlying data structure. The data structures are defined by the *UCI.h* include file. Handles form a hierarchy as follows:

1. The application allocates an environment (of data type HENV).
2. Using that environment, one or more connections (of data type HDBC) are established.
3. Once a connection has been established, one or more statements (of data type HSTMT) can be allocated. Each statement is associated with only one connection.

Call arguments are summarized in the following table.

Argument	Comment
HDBC	(void *)
HENV	(void *)
HSTMT	(void *)

Arguments in UCI Calls

Argument	Comment
PTR	(void *)
RETCODE	Always 32 bits
SDWORD	Always 32 bits
SWORD	Always 16 bits
UCHAR	Always 8 bits
UDWORD	Always 32 bits
UWORD	Always 16 bits

Arguments in UCI Calls (Continued)

In the syntax section of each function, variable descriptions are divided into two groups: input variables and output variables. An input variable is an argument that you must supply to the function for use in its execution. An output variable represents data returned by the function; however, you must still provide a value for it in the call (usually a pointer to where the data is to be stored).

Search Patterns

Information returned by a function can, in some cases, be controlled by a search pattern that you pass as an argument to that function. For example, `SQLColumns` returns a result set describing the columns from the tables specified in the search pattern. Besides the standard alphanumeric characters, you can use the following characters as wildcards:

Character	Description
_	An underscore in a pattern represents any single character.
%	A percent sign in a pattern represents a sequence of zero or more characters.
\	A backslash is an escape character, which is placed before the _ or % to indicate that the _ or % represents itself in the search pattern and is not a wildcard.

Wildcard Characters

As an example, to cause SQLColumns to return the columns from all tables that are named REF_TBL*x*, use the search pattern REF_TBL_. The first underscore, which is preceded by a backslash, is interpreted as a literal backslash, whereas the second underscore is interpreted as any single character. Note that using a search pattern of % represents an empty pointer and, in this example, returns all tables.

Return Values

UCI functions return a value to the *status* variable. Return values are the following:

Return Value	Meaning
SQL_SUCCESS	Function call completed successfully.
SQL_SUCCESS_WITH_INFO	Function call completed successfully with a possible nonfatal error. Your program can call <code>SQLError</code> to get information about the error.
SQL_ERROR	Function call failed. Your program can call <code>SQLError</code> to get information about the error.
SQL_INVALID_HANDLE	Function call failed because one of the three handles (environment, connection, or SQL statement) is invalid.
SQL_NO_DATA_FOUND	All rows from the result set have been retrieved.

Return Values

Error Codes

Any UCI function call can generate errors. Use the `SQLError` function after any other function call for which the returned status indicates an error condition. UCI follows the guidelines dictated by the Microsoft ODBC specification in returning these error codes. For a list of UCI function error codes, see [SQLError](#) later in this chapter, Appendix A, “[Error Codes](#)” for more detail.

Use of Hungarian Naming Conventions

In the function syntax that follows, some elements of the Hungarian naming convention are used as prefixes to variable names, and some variable names also include a tag after the prefix.

The prefixes are:

Prefix	Meaning
<i>c</i>	count of
<i>h</i>	handle to
<i>i</i>	index of
<i>p</i>	pointer to
<i>rg</i>	range (array) of

The tags are:

Tag	Meaning
<i>b</i>	byte
<i>col</i>	column (of a result set)
<i>dbc</i>	database connection
<i>env</i>	environment
<i>f</i>	flag; unsigned integer
<i>par</i>	parameter
<i>row</i>	row (or a result set)
<i>stmt</i>	statement
<i>sz</i>	null-terminated string
<i>v</i>	value of unspecified type

For example, *hdbc* is a *handle* for a *database connection*, *ipar* is an *index parameter*, *pib* is a *pointer* to an *index byte*, and *rgb* is a *range array* of *bytes*.

Functions

UCI function calls are presented on the following pages in alphabetical order. Each function is described in terms of syntax, input and output variables, description, return values, and SQLSTATE values. Some functions also have an example.

Programmatic SQL statements are case-sensitive. You must code all SQL statement names (such as CREATE TABLE, SELECT, and INSERT), SQL keywords (such as INTEGER, WHERE, FROM, and GROUP BY), and database-specific keywords (such as ROWUNIQUE and UNNEST) in uppercase letters. You must code identifiers such as table and column names to match the format of the identifier as originally defined.



Note: An asterisk (*) following a Type entry in a table of input or output variables indicates that the argument is the address of a variable that is a pointer. A double asterisk (**) indicates that the argument is a pointer to a pointer.

SQLAllocConnect

SQLAllocConnect allocates memory for a connection handle within the environment identified by *henv*. You must issue a call to SQLAllocConnect before you try to connect to a server.

Syntax

RETCODE SQLAllocConnect (*henv*, *phdbc*)

Input Variable

The following table describes the input variable:

Type	Argument	Description
HENV	<i>henv</i>	Environment handle returned in an SQLAllocEnv call.

SQLAllocConnect Input Variable

Output Variable

The following table describes the output variable:

Type	Argument	Description
HDBC *	<i>phdbc</i>	Pointer to where the connection handle is stored. If an error is returned, <i>phdbc</i> is set to SQL_NULL_HDBC.

SQLAllocConnect Output Variable

Description

Use this function to create a connection environment to connect to a particular data source. SQLAllocConnect stores the environment handle in *phdbc*.

One environment can have several connection handles, one for each data source.

Return Values

SQL_SUCCESS
SQL_ERROR
SQL_INVALID_HANDLE

SQLSTATE Values

The following table describes the SQLSTATE values:

SQLSTATE	Description
S1001	Memory allocation failure.
S1009	<i>phdbc</i> is a null pointer.

SQLSTATE Values

SQLAllocEnv

SQLAllocEnv allocates memory for an environment handle and initializes the interface for use by the client application. This must be the first call issued before any other UCI function.

Syntax

RETCODE SQLAllocEnv (*phenv*)

Output Variable

The following table describes the output variable:

Type	Argument	Description
HENV *	<i>phenv</i>	Pointer to where the environment handle is stored.

SQLAllocEnv Output Variable

Description

Use this function to allocate memory for an environment. The address is stored in *phenv*.

You cannot allocate more than one environment.

Return Values

SQL_SUCCESS
SQL_ERROR

SQLSTATE Values

No SQLSTATE can be returned on an error, because there is no valid *henv* for the **SQLError** call. If the call fails, the failure is caused by one of the following:



- Memory allocation failed.
- The *phenv* argument is 0.
- The application already allocated an environment handle.

Note: *Only one environment handle is permitted to be active at one time.*

SQLAllocStmt

SQLAllocStmt allocates memory for a statement handle and associates the statement handle with the connection specified by *hdbc*.

Syntax

RETCODE SQLAllocStmt (*hdbc*, *phstmt*)

Input Variable

The following table describes the input variable.

Type	Argument	Description
HDBC	<i>hdbc</i>	Connection handle.

SQLAllocStmt Input Variable

Output Variable

The following table describes the output variable.

Type	Argument	Description
HSTMT *	<i>phstmt</i>	Pointer to where the statement handle is stored.

SQLAllocStmt Output Variable

Description

A statement handle represents a single SQL statement and holds all information that UCI needs to describe results, return data rows, and so forth.

An application should not call `SQLAllocStmt` with a pointer to a valid statement environment, because UCI will overwrite the pointer with the address of the newly allocated environment, causing the memory allocated by the previous HSTMT handle to be lost.

Return Values

`SQL_SUCCESS`
`SQL_ERROR`
`SQL_INVALID_HANDLE`

SQLSTATE Values

The following table describes the SQLSTATE values.

Value	Description
S1009	<i>phstmt</i> argument was null.
08003	No connection has been established.
SQLAllocStmt SQLSTATE Values	

SQLBindCol

SQLBindCol assigns storage for loading data from a column in a result set and specifies any data conversion to be performed.

Syntax

RETCODE SQLBindCol (*hstmt*, *icol*, *fCType*, *rgbValue*, *cbValueMax*, *pcbValue*)

Input Variables

The following table describes the input variables.

Type	Argument	Description
HSTMT	<i>hstmt</i>	Statement handle.
UWORD	<i>icol</i>	Column number of the result set, numbered left to right starting at 1. This value must be from 1 through the number of columns returned in an operation.
SWORD	<i>fCType</i>	C data type into which to convert the incoming data. See C Data Types Supported in Chapter 7, “Data Types,” for a complete list of valid C data types.
PTR	<i>rgbValue</i>	Pointer to the storage area allocated to hold the result set. For an SQL_C_STRING data type, this should be the address of the structure’s text member, and <i>pcbValue</i> should be the address of the length part of the structure.
SDWORD	<i>cbValueMax</i>	Maximum length of the <i>rgbValue</i> buffer. For character data, this must include space for the null terminator.
SDWORD *	<i>pcbValue</i>	If the cell value is null, this will contain SQL_NULL_DATA. For character data, this contains the number of bytes available to return. If this is greater than or equal to <i>cbValueMax</i>, the data returned is truncated to <i>cbValueMax</i> – 1 bytes.

SQLBindCol Input Variables

Type	Argument	Description
		If an SQL_DATE or SQL_TIME column contains an empty string, SQL_SUCCESS is returned, but <i>pcbValue</i> is set to SQL_BAD_DATA.
		For binary data, this contains the number of bytes available to return. If this is greater than <i>cbValueMax</i> , the data returned is truncated to <i>cbValueMax</i> bytes.
		For all other data types, this contains the size of the application data type specified.
SQLBindCol Input Variables (Continued)		

Description

Use this function to tell UCI where to return the results of an SQLFetch call. SQLBindCol defines where data values retrieved from the database by SQLFetch are to be stored in the application and specifies the data conversion (*fCType*) to be performed on the fetched data.

SQLBindCol is designed for use on singlevalued data primarily. If you use it for a column and at SQLFetch time that column is found to contain multivalues, only the first value is returned, coerced into the requested data type. A return code of SQL_SUCCESS_WITH_INFO is also returned with an SQLSTATE of IM981 to indicate that the multivalued data was truncated to the first value. Successive calls to SQLGetData fetch successive values for that column. However, this approach is much less efficient than using SQLBindMvCol, which is the recommended method. Using both binding methods for the same *hstmt* is permitted, and may in fact be necessary to deal with those queries that generate a mix of single-valued and multivalued data.



Note: *Issuing this call does not fetch data from the database, but only performs the setup for SQLFetch. SQLBindCol has no effect until SQLFetch is used.*

Normally you call SQLBindCol once for each column of data in the result set. Issuing SQLFetch moves data from the result set at the data source to the variables specified in the SQLBindCol call, overwriting any existing contents.

Data is converted from the data source to the data type requested by the SQLBindCol call, if possible. If data cannot be converted to *fCType*, an error occurs and the column is not bound. See [C Data Types Supported](#) in Chapter 7, “Data Types,” for information about data conversion types.

Values are returned only for bound columns when a call to SQLFetch is issued. Unbound columns are ignored and are not accessible unless you call SQLGetData.

For example, if a SELECT statement returns three columns, but you called SQLBindCol for only two columns, data from the third column is accessible to your program only by using SQLGetData on the column. If you bind more variables than there are columns in the result set, an error is returned. If you bind no columns and SQLFetch is issued, the cursor advances to the next row of results and no program variables are loaded with data.

Do not use SQLBindCol with SQL statements that do not produce result sets.



Note: Be careful when executing a new SQL statement with a statement handle that already has columns bound with SQLBindCol. If you do not use the SQLFreeStmt call with the SQL_UNBIND option first, UCI assumes that the previous column bindings are still in effect. If the new SQL statement generates fewer columns than the previous SQL statement, the new SQL statement fails with an SQLSTATE of S1002, indicating that the wrong number of columns were bound. This error might also lead to data conversion errors if the columns for the new SQL statement cannot be converted according to the previous bindings.

Return Values

SQL_SUCCESS
SQL_ERROR
SQL_INVALID_HANDLE

SQLSTATE Values

The following table describes the SQLSTATE values.

SQLSTATE	Description
S1000	General error for which no specific SQLSTATE code has been defined.
S1001	Memory allocation failure.
S1002	Illegal column number. The value of <i>icol</i> is greater than the number of columns in the result set or is less than 1.
S1003	The <i>fCType</i> argument is not a recognized data type.
S1009	<i>rgbValue</i> is a null pointer.
S1090	The value of <i>cbValueMax</i> is less than 0.

SQLBindCol SQLSTATE Values

Example

This program fragment determines the number of columns generated from the execution of an SQL statement and, if there are results, binds up to 10 columns to a column array.



Note: The code to allocate environments, connections, and the like is not shown here.

```
#define MAXCOLS 10
#define COLUMN_WIDTH 132

#include <stdio.h>
#include "UCI.h"

struct column
{
    char column_buffer [COLUMN_WIDTH];
    SDWORD column_outlen;
};

SWORD numcols;
```

```

HSTMT hstmt;
struct column columns[10];    /* Max of 10 columns */

main()
{
    int indexs;
    /* All allocation, connection, etc., code goes here */
    SQLExecDirect ( hstmt, "SELECT * FROM MYTABLE");
    /* Get the number of columns produced. If there are any,
     * bind them all to character strings in the column
     * array. */
    SQLNumResultCols (hstmt, &numcols);

    if (numcols)
    {
        for (indexs = 1; indexs <= MAXCOLS; indexs ++)
        {
            SQLBindCol(hstmt, indexs,
                        SQL_C_CHAR,
                        &columns[indexs].column_buffer,
                        COLUMN_WIDTH,
                        &columns[indexs].column_outlen);
        }
    }
}

```

SQLBindMvCol

SQLBindMvCol is a database-specific extension of the SQLBindCol function. It simplifies the fetching of multivalued data by normalizing it into arrays of C program variables allocated by UCI. UCI allocates storage based on the number of values returned, so you do not need to know how much storage to allocate in advance.

Syntax

RETCODE SQLBindMvCol (*hstmt*, *icol*, *fCType*, *pCArray*)

Input Variables

The following table describes the input variables.

Type	Argument	Description
HSTMT	<i>hstmt</i>	Statement handle.
UWORD	<i>icol</i>	Column number of the result set, numbered left to right starting at 1.
SWORD	<i>fCType</i>	The data type for storing the data, in the UCI_DATUM union.
C_ARRAY **	<i>pCArray</i>	Address of a pointer to an array where the returned data is to be stored following an SQLFetch call.

SQLBindMvCol Input Variables

Description

This extension to SQLBindCol allows a simple model to be used in dealing with multivalued columns. UCI allocates storage for these values and returns addresses to the application in the form of an array.

The user application specifies the application data type into which to convert the data. UCI allocates a data structure for each value it encounters in a particular column, and returns to the application the address of that array of values. As the application need not be concerned with allocating storage before fetching a row of data, there is no *cbValueMax* parameter in this call.

You can also use SQLBindMvCol with singlevalued data whenever you want UCI to allocate storage.

When the contents of the attribute evaluate to an empty string, a subsequent SQLFetch returns a C_ARRAY structure whose *cDcount* field is 0, rather than returning one value whose content is the empty string.

The SQLBindMvCol function allocates memory as necessary to hold multivalued data, and returns a pointer to the allocated memory. The program is responsible for freeing the allocated memory with the SQLFreeMem function.

Return Values

SQL_SUCCESS
SQL_ERROR
SQL_INVALID_HANDLE

SQLSTATE Values

The following table describes the SQLSTATE values.

SQLSTATE	Description
S1000	General error for which no specific SQLSTATE code has been defined.
S1001	Memory allocation failure.
S1002	Illegal column number. The value of <i>icol</i> specified is greater than the number of columns in the result set.
S1003	A data type in the array is not a recognized data type.

SQLBindMvCol SQLSTATE Values

Example

The following fragment of pseudocode shows how to use this call to print some results. The example assumes a two-column result set, with the first column being single-valued, and the second column being a multivalued column containing integers.

```
#include "UCI.h"

UCHAR      collbuff[100];
HSTMT      hstmt;
SDWORD     collsize;
UWORD      nv;
RETCODE     status;
C_ARRAY     *pCArray;
UCI_DATUM  *ud;
SQLExecDirect (hstmt, "SELECT COL1, COL2 FROM MYTABLE");
SQLBindCol (hstmt, 1, SQL_C_CHAR, collbuff, 100, &collsize);
SQLBindMvCol (hstmt, 2, SQL_C_INTEGER, &pCArray);
while ((status = SQLFetch(hstmt)) == SQL_SUCCESS)
{
    printf(" %s\n ", collbuff);
    nv = pCArray->cDcount;
    ud = pCArray->Data;
    while (nv--)
    {
        if (ud->fIndicator == SQL_NULL_DATA)
        {
            printf("\t NULL \n");
        }
        else if (ud->fIndicator == SQL_BAD_DATA)
        {
            printf("\t Data could not be converted \n");
        }
        else
        {
            printf("\t %d\n", ud->uValue.int);
        }
        ud++;
    }
}
SQLFreeMem (*pCArray);
```

SQLBindMvParameter

SQLBindMvParameter is a database-specific extension of the SQLBindParameter function. It allows an application to write a multivalued column from an array of C variables (which is the form read by SQLFetch after SQLBindMvCol has been called).

Syntax

RETCODE SQLBindMvParameter (*hstmt*, *ipar*, *pCArray*)

Input Variables

The following table describes the input variables.

Type	Argument	Description
HSTMT	<i>hstmt</i>	Statement handle.
UWORD	<i>ipar</i>	Parameter number, ordered sequentially from the right starting at 1.
C_ARRAY*	<i>pCArray</i>	Pointer to an array that specifies the number of values, the data types, an array of pointers to the data, and an array of indicator/length values. <i>fParamType</i> is always SQL_PARAM_INPUT, no matter what value is used in the array. Multivalued output and input/output parameters are not supported.

SQLBindMvParameter Input Variables

Description

This function allows data to be used in the form returned by SQLBindCol anywhere in the SQL grammar that a parameter marker can be used. The array of data of type *fCType* is processed by UCI into a dynamic array in a form that the database can use internally—the reverse of how a multivalued dynamic array is processed into a C array by SQLBindCol.

The *pCArray* argument is the address of an array of C_ARRAY structures that you must manage in your application program. If the memory is allocated from system memory with the *malloc* command, be sure that you free that memory when you no longer need it.

For further information, refer to **SQLBindParameter**.

Return Values

SQL_SUCCESS
SQL_ERROR
SQL_INVALID_HANDLE

SQLSTATE Values

The following table describes the SQLSTATE values.

SQLSTATE	Description
IM977	Multivalued parameter markers can only be SQL_PARAM_INPUT.
S1000	General error for which no specific SQLSTATE code has been defined.
S1001	Memory allocation failure.
S1003	The argument <i>fCType</i> is not a recognized data type.
S1093	<i>ipar</i> was less than 1 or greater than the number of parameters in the SQL statement.

SQLBindMvParameter SQLSTATE Values

SQLBindParameter

SQLBindParameter binds an application buffer to a parameter marker in an SQL statement. It is functionally similar to the SQLSetParam call in the ODBC 1.0 specifications. SQLSetParam has also been provided in UCI for compatibility.

Syntax

RETCODE SQLBindParameter (*hstmt*, *ipar*, *fParamType*, *fCType*, *fSqlType*, *cbColDef*, *ibScale*, *rgbValue*, *cbValueMax*, *pcbValue*)

Input Variables

The following table describes the input variables.

Type	Argument	Description
HSTMT	<i>hstmt</i>	Statement handle.
UWORD	<i>ipar</i>	Parameter number, ordered sequentially from left to right starting at 1.
SWORD	<i>fParamType</i>	Can be one of the following: SQL_PARAM_INPUT Use for parameters in an SQL statement that does not call a procedure, or for input parameters in a procedure call. SQL_PARAM_OUTPUT Use for parameters that mark the return value of a procedure or an output parameter in a procedure. SQL_PARAM_INPUT_OUTPUT Use for an input/output parameter in a procedure.
SWORD	<i>fCType</i>	C data type from which to convert the incoming data. See C Data Types Supported in Chapter 7, “Data Types,” for a complete list of valid C data types supported.
SWORD	<i>fSqlType</i>	SQL data type of the parameter. For more information, see “The fSqlType Parameter” on page 29.

SQLBindParameter Input Variables

Type	Argument	Description
UDWORD	<i>cbColDef</i>	Not currently used, but reserved for precision of the column or expression of the associated parameter marker. You must set it to SQL_UV_DEFAULT_PARAMETER. For more information, see “The cbColDef and ibScale Parameters” on page 29.
SWORD	<i>ibScale</i>	Not currently used, but reserved for scale of the column or expression of the associated parameter marker. You must set it to SQL_UV_DEFAULT_PARAMETER. For more information, see “The cbColDef and ibScale Parameters” on page 29.
PTR	<i>rgbValue</i>	Pointer to the buffer for the parameter’s data. If you are using SQLParamOptions, <i>rgbValue</i> points to an array of data values.
SDWORD	<i>cbValueMax</i>	Maximum length of the <i>rgbValue</i> buffer.
SDWORD *	<i>pcbValue</i>	Pointer to the buffer holding the parameter’s length. If you are using SQLParamOptions, <i>pcbValue</i> points to an array of parameter lengths. For more information, see “The pcbValue Parameter” on page 30.

SQLBindParameter Input Variables (Continued)

Description

Use this function when parameter markers (represented by the ? character) are used as part of the SQL statement syntax. This call identifies the program variables that are used to hold values for each parameter marker in the statement. When you issue an SQLExecDirect or an SQLExecute call, UCI extracts the value now in place for each marker, checks for any data conversion errors, and delivers the values to the server, where they are inserted into the SQL statement. The statement is then executed.

You need to call SQLBindParameter only once for each marker. From that point on UCI remembers where to find each marker and what its characteristics are.

*The **fSqlType** Parameter*

For the database, if *fSqlType* is set to either SQL_DATE or SQL_TIME, the parameter is used as follows:

fSqlType Parameter	Description
SQL_DATE	Any <i>fCType</i> value is permitted with SQL_DATE except for SQL_C_TIME. However, if SQL_C_CHAR or SQL_C_STRING is specified, the data literal must be in the form <i>yyyy-mm-dd</i> , as specified in the ODBC 2.0 specification. This removes ambiguities related to European date formats.
SQL_TIME	Any <i>fCType</i> value is permitted with SQL_TIME except for SQL_C_DATE. However, if SQL_C_CHAR or SQL_C_STRING is specified, the time literal must be in the form <i>hh:mm:ss</i> , as specified in the ODBC 2.0 specification.

***fSqlType** Parameters*

Generally, *fSqlType* is used to ensure that the data presented to UCI for the parameter marker is compatible with the data type of the marker. For example, if the application specifies a numeric SQL type for a marker, and the data presented at execution is a text string rather than a numeric string, UCI returns SQLSTATE 22005.

*The **cbColDef** and **ibScale** Parameters*

According to the ODBC 2.0 specification, *cbColDef* contains the precision of the parameter marker, and *ibScale* contains the scale of the marker. Both of these depend on the value loaded into *fSqlType*, and *ibScale* is relevant only for DECIMAL and NUMERIC SQL types. As of Release 8.3.3, these fields are ignored, and you should set these arguments to SQL_UV_DEFAULT_PARAMETER.

The pcbValue Parameter

The *pcbValue* parameter has several different meanings, as shown in the following table:

<i>fCType</i>	<i>pcbValue Description</i>
All	If <i>pcbValue</i> points to a location that contains the constant SQL_NULL_DATA, the value that will be used for the parameter is the null value.
SQL_C_CHAR or SQL_C_STRING	If the <i>pcbValue</i> pointer is 0, or the location it points to is 0, <i>rgbValue</i> is interpreted as the address of a null-terminated character string. In this case, data up to the first 0x00 byte is sent to the server. If <i>pcbValue</i> points to a valid program variable, that variable should contain the length of the data pointed to by <i>rgbValue</i>. This value is valid only if <i>fParamType</i> is SQL_PARAM_INPUT or SQL_PARAM_INPUT_OUTPUT. If <i>fParamType</i> is SQL_PARAM_OUTPUT or SQL_PARAM_INPUT_OUTPUT, the <i>pcbValue</i> cannot be 0. Note: UCI ignores the contents of <i>pcbValue</i> if it does not point to a location containing SQL_NULL_DATA and <i>fCType</i> is not SQL_C_CHAR or SQL_C_STRING.
SQL_NTS	The <i>rgbValue</i> is a null-terminated string.

pcbValue Parameters

Return Values

SQL_SUCCESS
SQL_ERROR
SQL_INVALID_HANDLE

SQLSTATE Values

The following table describes the SQLSTATE values.

SQLSTATE	Description
S1000	General error for which no specific SQLSTATE code has been defined.
S1001	Memory allocation failure.
S1003	The <i>fCType</i> argument is not a recognized data type.
S1090	The value of <i>cbValueMax</i> is less than 0.
S1093	<i>ipar</i> is less than 1 or greater than the number of parameters in the SQL statement, or <i>fParamType</i> is not SQL_PARAM_INPUT, or <i>cbColDef</i> or <i>ibScale</i> is not SQL_UV_DEFAULT_PARAMETER.
07006	The <i>fCType</i> data type cannot be converted to the <i>fSqlType</i> data type.

SQLBind Parameter SQLSTATE Values

SQLCancel

SQLCancel cancels the processing of the current SQL statement and discards any pending results. SQLCancel is equivalent to SQLFreeStmt with the SQL_CLOSE option specified.

Syntax

RETCODE SQLCancel (*hstmt*)

Input Variable

The following table describes the input variable.

Type	Argument	Description
HSTMT	<i>hstmt</i>	Statement handle.

SQLCancel Input Variable

Description

This function closes any open cursor for the statement handle supplied and discards pending results at the data source. *hstmt* can be reopened by executing it again, using the same or different parameters.

SQLCancel can also be used to cancel a long-running SQLExecute or SQLExecDirect operation on an *hstmt*. To do this, an application must use signal handlers to trap the ^C (Ctrl-C) interrupt from the terminal. Issuing an SQLCancel request from the signal handler interrupts the server's operation and returns an SQLSTATE of S1008 to the execute request. In the application interrupt handler, the only legal operation is to cancel the executing *hstmt*. Virtually all other attempted functions fail and will return an SQLSTATE of S1010 (Function sequence error) to the application. An attempt to cancel an *hstmt* not currently executing also causes an S1010 error return.

Return Values

SQL_SUCCESS
SQL_ERROR
SQL_INVALID_HANDLE

SQLSTATE Values

The following table describes the SQLSTATE values.

SQLSTATE	Description
S1000	General error for which no specific SQLSTATE code has been defined.
S1001	Memory allocation failure.
S1010	Function sequence error. An attempt was made to cancel an <i>hstmt</i> while another <i>hstmt</i> was still executing.

SQLCancel SQLSTATE Values

SQLColAttributes

SQLColAttributes returns more extensive column attribute information than SQLDescribeCol.

Syntax

RETCODE SQLColAttributes (*hstmt*, *icol*, *fDescType*, *rgbDesc*, *cbDescMax*, *pcbDesc*, *pfDesc*)

Input Variables

The following table describes the input variables.

Type	Argument	Description
HSTMT	<i>hstmt</i>	Statement handle.
UWORD	<i>icol</i>	Column number to describe, numbered sequentially from left to right starting at 1.
UWORD	<i>fDescType</i>	A valid descriptor type. Refer to “Description.”
SWORD	<i>cbDescMax</i>	Maximum length of the <i>rgbDesc</i> descriptor area.

SQLColAttributes Input Variables

Output Variables

The following table describes the output variables.

Type	Argument	Description
PTR	<i>rgbDesc</i>	Pointer to storage for character strings returned as results.
SWORD *	<i>pcbDesc</i>	Pointer to the location used to hold the total number of bytes available to return in <i>rgbDesc</i> .
SDWORD *	<i>pfDesc</i>	Pointer to the location used to hold the description information for numeric descriptor types.

SQLColAttributes Output Variables

Description

Depending on the attribute requested, SQLColAttributes can return the result as either a character string or an integer value.

You can call SQLColAttributes only after the statement has been prepared by either SQLPrepare or SQLExecDirect; before either of these two calls, the information is not available. If the statement is an SQL procedure call, column information is not available until after the statement is executed. Integer information is returned in *pfDesc* as a 32-bit value.

All other formats are returned in *rgbDesc* (the use of which depends on *fDescType*). The following table shows where each result is returned.

If <i>fDescType</i> is...	Information is returned in...	Description
SQL_COLUMN_AUTO_INCREMENT	<i>pfDesc</i>	TRUE if the values in the column are automatically incremented, otherwise FALSE
SQL_COLUMN_CASE_SENSITIVE	<i>pfDesc</i>	TRUE for character data. FALSE for all other.
SQL_COLUMN_CONVERSION	<i>rgbDesc</i>	The CONV entry for this column in the file dictionary.
SQL_COLUMN_COUNT	<i>pfDesc</i>	Number of columns in the result set. The <i>icol</i> argument must be a valid column number in the result set.
SQL_COLUMN_DISPLAY_SIZE	<i>pfDesc</i>	See Precision, Scale, and Display Size Definitions in Chapter 7, “Data Types,” for details.
SQL_COLUMN_FORMAT	<i>rgbDesc</i>	The FMT entry for this column in the file dictionary.
SQL_COLUMN_LABEL	<i>rgbDesc</i>	Column heading. If COL.HDG, DISPLAYLIKE, or DISPLAYNAME is used in the query, the descriptor contains the column heading, otherwise the descriptor contains the column name. If the column has no heading or name, an empty string is returned.

If <i>fDescType</i> is...	Information is returned in...	Description
SQL_COLUMN_LENGTH	<i>pfDesc</i>	The amount of data transferred using SQLFetch. See Precision, Scale, and Display Size Definitions in Chapter 7, “Data Types,” for details.
SQL_COLUMN_MULTI_VALUED	<i>pfDesc</i>	TRUE if this is a multivalued column, otherwise FALSE.
SQL_COLUMN_NAME	<i>rgbDesc</i>	Name of the column. If the column is an expression, an empty string is returned.
SQL_COLUMN_NULLABLE	<i>pfDesc</i>	SQL_NULLABLE if the column can contain nulls, otherwise SQL_NO_NULLS.
SQL_COLUMN_PRECISION	<i>pfDesc</i>	See Precision, Scale, and Display Size Definitions in Chapter 7, for details.
SQL_COLUMN_PRINT_RESULT	<i>pfDesc</i>	Can be either TRUE or FALSE, indicating that the column is or is not a one-column PRINT result set from a called procedure. See Processing UniVerse Procedure Results in Chapter 5, “Calling and Executing UniVerse Procedures,” for details.
SQL_COLUMN_SCALE	<i>pfDesc</i>	For a file, always 0; for a table column that is DECIMAL or NUMERIC, the scale is taken from the column definition in the dictionary.

If <i>fDescType</i> is...	Information is returned in...	Description
SQL_COLUMN_SEARCHABLE	<i>pfDesc</i>	Always SQL_SEARCHABLE.
SQL_COLUMN_TABLE_NAME	<i>rgbDesc</i>	Name of the table to which the column belongs. If the column is an expression, an empty string is returned.
SQL_COLUMN_TYPE	<i>pfDesc</i>	A number representing the column's SQL data type.
SQL_COLUMN_TYPE_NAME	<i>rgbDesc</i>	Name of the column's data type.
SQL_COLUMN_UNSIGNED	<i>pfDesc</i>	TRUE for nonnumeric data types, otherwise FALSE.
SQL_COLUMN_UPDATABLE	<i>pfDesc</i>	As of Release 9, any expressions or computed columns return SQL_ATTR_READONLY, and stored data columns return SQL_ATTR_WRITE.



Note: *SQL_COLUMN_CONVERSION*, *SQL_COLUMN_FORMAT*, and *SQL_COLUMN_MULTI_VALUED* are specific to the database; the remainder are part of standard ODBC.

The values returned for some of these column attributes are of limited use to database applications. For example, in databases constrained to fixed-length columns, the precision of a column is typically of fundamental importance, and can be viewed as an internal constraint on the data stored in that column. The database does not enforce such constraints, so although a column may be defined as CHAR(30), the database does not prohibit entry of more than 30 characters. Likewise, the attributes SQL_COLUMN_DISPLAY_SIZE, SQL_COLUMN_PRECISION, SQL_COLUMN_SCALE, and SQL_COLUMN_LENGTH are only approximations and do not place constraints on the data that the application can insert.

Return Values

SQL_SUCCESS
SQL_ERROR
SQL_INVALID_HANDLE
SQL_SUCCESS_WITH_INFO

SQLSTATE Values

The following table describes the SQLSTATE values.

SQLSTATE	Description
S1000	General error for which there is no specific SQLSTATE code defined.
S1001	Memory allocation failure.
S1002	Illegal column number. The value of <i>icol</i> is less than 1 or is greater than the number of columns in the result set.
S1009	<i>rgbDesc</i> or <i>pcbDesc</i> is null, or the result returned will be an integer and <i>pfDesc</i> is null.
S1010	Function sequence error. SQLColAttributes was called before calling either SQLPrepare or SQLExecDirect. In the case of a procedure call statement, SQLColAttributes was called before calling either SQLExecute or SQLExecDirect.
S1090	The value of <i>cbDescMax</i> is less than 0.
01004	The <i>rgbDesc</i> buffer was too small. The <i>pcbDesc</i> parameter holds the length of the untruncated value. The string in <i>rgbDesc</i> is truncated to <i>cbDescMax</i> – 1 bytes. SQL_SUCCESS_WITH_INFO is returned as the status code.
24000	<i>hstmt</i> has no result set pending. There are no columns to describe.

SQLColAttributes SQLSTATE Values

SQLColumns

SQLColumns returns a result set listing the columns matching the search patterns.

Syntax

RETCODE SQLColumns (*hstmt*, *szTableQualifier*, *cbTableQualifier*, *szTableOwner*, *cbTableOwner*, *szTableName*, *cbTableName*, *szColumnName*, *cbColumnName*)

Input Variables

The following table describes the input variables.

Type	Argument	Description
HSTMT	<i>hstmt</i>	Statement handle.
UCHAR *	<i>szTableQualifier</i>	Qualifier (schema) name search pattern.
SWORD	<i>cbTableQualifier</i>	Length of <i>szTableQualifier</i> .
UCHAR *	<i>szTableOwner</i>	Table owner number search pattern.
SWORD	<i>cbTableOwner</i>	Length of <i>szTableOwner</i> .
UCHAR *	<i>szTableName</i>	Table name search pattern.
SWORD	<i>cbTableName</i>	Length of <i>szTableName</i> .
UCHAR *	<i>szColumnName</i>	Column name search pattern.
SWORD	<i>cbColumnName</i>	Length of <i>szColumnName</i> .

SQLColumns Input Variables



Description

This function returns a result set in *hstmt* as a cursor of 13 columns describing those columns found by the search pattern (refer to **SQLTables**). As with **SQLTables**, the search is done on the SQL catalog. This is a standard result set that can be accessed with **SQLFetch**. The ability to obtain descriptions of columns does not imply that a user has any privileges on those columns.

*If the application is running in 1NF mode, **szTableOwner** and **szColumnName** are ignored, and a null value is returned for the owner.*

Note: ***szTableOwner** is the user ID of the person who created the table. **SQLColumns** accepts the **TableOwner** search pattern as a character string, but the character string must equate to an integer value and must not contain wildcards.*

The result set contains 13 columns:

	NF ² Mode	1NF Mode
TABLE_SCHEMA	CHAR(18)	CHAR(18)
OWNER	INTEGER	VARCHAR ^a
TABLE_NAME	CHAR(18)	CHAR(18)
COLUMN_NAME	CHAR(18)	CHAR(18)
DATA_TYPE_NULL ^b	VARCHAR	VARCHAR
TYPE_NAME	CHAR(18)	CHAR(18)
NUMERIC_PRECISION	INTEGER	INTEGER
CHAR_MAX_LENGTH	INTEGER	INTEGER
NUMERIC_SCALE	INTEGER	INTEGER
NUMERIC_PREC_RADIX	INTEGER	INTEGER
NULLABLE_UV	VARCHAR	VARCHAR
REMARKS	CHAR(254)	CHAR(254)
MULTI_VALUE ^c	CHAR(1)	VARCHAR

SQLColumns Result Set

a. In 1NF mode, OWNER is always NULL.

- b. DATA_TYPE_NULL is always NULL.
- c. In 1NF mode, MULTI_VALUE_S.

The application is responsible for binding variables for the output columns and fetching the results using SQLFetch. The result set contains one column in addition to those columns listed in the ODBC 2.0 interface description. This is the MULTI_VALUE column, which returns S for single-valued columns and M for multivalued columns.

If no search criteria are specified, the SQL statement executed by SQLColumns is:

```
SELECT A.TABLE_SCHEMA, OWNER, A.TABLE_NAME, COLUMN_NAME,
      NULL COL.HDG 'Data Type' AS DATA_TYPE_NULL,
      DATA_TYPE COL.HDG 'Type Name' AS TYPE_NAME,
      NUMERIC_PRECISION,
      CHAR_MAX_LENGTH, NUMERIC_SCALE, NUMERIC_PREC_RADIX,
      EVAL B.'IF NULLABLE="NO" THEN 0 ELSE 1' COL.HDG 'Nullable'
      AS NULLABLE_UV, B.REMARKS, MULTI_VALUE
FROM UNNEST UV_TABLES ON COLUMNS A, UV_COLUMNS B
WHERE A.TABLE_SCHEMA = B.TABLE_SCHEMA
AND A.TABLE_NAME = B.TABLE_NAME
AND A.COLUMNS = B.COLUMN_NAME
ORDER BY 1, 2, 3;
```

In SQL_1NF_MODE_ON mode, the SQL statement executed by SQLColumns is:

```
SELECT TABLE_SCHEMA, NULL COL.HDG 'Owner' AS OWNER, TABLE_NAME,
      COLUMN_NAME, NULL COL.HDG 'Data Type' AS DATA_TYPE_NULL,
      DATA_TYPE COL.HDG 'Type Name' AS TYPE_NAME,
      NUMERIC_PRECISION,
      CHAR_MAX_LENGTH, NUMERIC_SCALE, NUMERIC_PREC_RADIX,
      EVAL 'IF NULLABLE="NO" THEN 0 ELSE 1' COL.HDG 'Nullable'
      AS NULLABLE_UV, REMARKS, 'S' COL.HDG 'Single/Multivalued'
      AS MULTI_VALUE_S
FROM UV_COLUMNS
WHERE MULTI_VALUE = 'S'
ORDER BY 1, 3;
```

If search criteria are specified, they are added as part of the SQL WHERE clause.

Return Values

```
SQL_SUCCESS
SQL_ERROR
SQL_INVALID_HANDLE
SQL_SUCCESS_WITH_INFO
```

SQLSTATE Values

The following table describes the SQLColumns SQLSTATE values.

SQLSTATE	Description
S1000	General error for which no specific SQLSTATE code has been defined.
S1001	Memory allocation failure.
S1008	Cancelled. Execution of the statement was stopped by an SQLCancel call.
S1010	Function sequence error. The <i>hstmt</i> specified is currently executing an SQL statement.
S1C00	The table owner field was not numeric.
24000	Invalid cursor state. Results are still pending from the previous SQL statement. Use SQLCancel to clear the results.
42000	Syntax error or access violation. This can happen for a variety of reasons. The native error code returned by the SQLError call indicates the specific database error that occurred.

SQLColumns SQLSTATE Values

SQLConnect

SQLConnect connects to a data source, which can be either a local or a remote UniVerse database. You cannot use SQLConnect inside a transaction.

Syntax

RETCODE SQLConnect (*hdbc*, *szDSN*, *cbDSN*, *szSchema*, *cbSchema*)

Input Variables

The following table describes the input variables.

Type	Argument	Description
HDBC	<i>hdbc</i>	Connection handle.
UCHAR *	<i>szDSN</i>	Pointer to a data source name (see “Description”).
SWORD	<i>cbDSN</i>	Length of the <i>szDSN</i> string.
UCHAR *	<i>szSchema</i>	Pointer to a schema name or account to log on to.
SWORD	<i>cbSchema</i>	Length of the <i>szSchema</i> string.

Description

The server uses the supplied data source name (*szDSN*) as a key to the UCI configuration file *uci.config*, which maps the name to a specific database account or schema on a specific system. A skeleton version of this file, shipped with UCI, allows connection to the local host using the name *localuv*. To add remote database entries, the system administrator must edit this configuration file. For more information about the UCI configuration file, see Chapter 3, “[Configuring UCI](#).”

The account identifier string must be one of the following:

- The schema name in the UV_SCHEMA table to which the server will attach itself

- An account name in the UV.ACCOUNT file
- A full path to define the directory to which the server will attach itself

The account identifier (*szSchema*) must point to a directory that has been set up to run UniVerse.

If the string does not begin with / (slash) or, on Windows systems, \ (backslash), both the UV_SCHEMA table and the UV.ACCOUNT file are examined. If the name is unambiguous (that is, it is defined in only one file or has the same definition in both files), it is used. If it is ambiguous, it is rejected.

Within an environment, UCI supports multiple connections to the same source as well as to different sources.

You can also specify certain connect time options with the SQLSetConnectOption call, and these take effect for the duration of the connection only.

Before issuing a call to SQLConnect, use SQLSetConnectOption calls to specify the user name (SQL_OS_UID) and password (SQL_OS_PWD) for logging in to a remote database server. On all systems but Windows NT 3.51, if the host specified for this DSN is either *localhost* or the TCP/IP loopback address (127.0.0.1), the user name and password are not required and are ignored if specified. On Windows NT 3.51 systems the user name and password are always required, so you must specify *localpc* as the DSN (for information about adding the *localpc* entry to the UCI configuration file, see [Editing the UCI Configuration File](#) in Chapter 3, “[Configuring UCI](#)”).

If the DSN is not the local host, the client passes the requested user name, password, and schema/account name through to the server. The server verifies the user name/password combination with the operating system and if that is valid, verifies that the requested schema is a valid schema or valid account on the server. Finally, the NLS map and locale settings, if set, are sent to the server. If any of these steps fails, an error is returned, indicating that the server rejected the connection request.

You must establish all connections before you can start a transaction.

Return Values

SQL_SUCCESS
SQL_ERROR
SQL_INVALID_HANDLE

SQLSTATE Values

The following table describes the SQLConnect SQLSTATE values.

SQLSTATE	Description
IM002	The specified data source was not found in the UCI configuration file.
IM980	A user password is required to connect to this data source.
IM982	A user identification is required to connect to this data source. This user must be found in the password file at the server.
IM984	UCI does not allow connections to data sources other than UniVerse.
IM987	Bad MAPERROR statement. A malformed MAPERROR statement was found in the UCI configuration file.
IM997	An illegal option was found in the UCI configuration file.
IM999	A network type other than TCP/IP or LAN Manager is specified for the data source.
S1000	General error for which no SQLSTATE code has been defined.
S1001	Memory allocation failure.
S1090	<i>szSchema</i> is 0 or <i>cbSchema</i> is less than or equal to 0.
08001	The connection could not be established. See “Error Codes” for more information.
08002	The <i>hdbc</i> used already has an active connection in place.
08004	The server rejected the connection. See “Error Codes.”
08S01	The communication link failed during the function.

SQLConnect SQLSTATE Values

Error Codes

An SQLSTATE return of 08001 or 08004 indicates that, for one of several reasons, the connection to the server could not be established. In such cases, further information can be obtained by issuing a call to `SQLError` and examining the native error code parameter. The most common reasons for a connect failure are as follows:

Code	Description
80011	The user name specified could not be found in the server system's password file.
81002	The server name specified in the data source was not found in the <i>unirpcservices</i> file on the server.
81011	The host specified in the <i>uci.config</i> file for the data source could not be found on the network.
81013	The <i>unirpcd</i> daemon on the UNIX server, or the <i>unirpc</i> service on the Windows server, could not open the <i>unirpcservices</i> file in the server's <i>unishared</i> directory.
81014	The service requested by the client could not be located or run by the server. Check the data source entry in the <i>uci.config</i> file to ensure that the service name in the entry is a valid entry in the <i>unirpcservices</i> file on the server.
81016	The <i>unirpcd</i> daemon on the UNIX server, or the <i>unirpc</i> service on the Windows server, is not running. Start the daemon or service on the server.
930098	The server could not create the helper process for the connection.
930127	The directory pointed to by <i>szSchema</i> is not a database account.
930133	<i>szSchema</i> was not an absolute pathname and was not found to be either a valid account or a schema.
930137	Cannot attach to the directory pointed to by <i>szSchema</i> .

Connect Error Codes

SQLDataSources

SQLDataSources returns information about data sources.

Syntax

RETCODE SQLDataSources (*henv*, *fDirection*, *szDSN*, *cbDSNMax*, *pcbDSN*, *szDescription*, *cbDescriptionMax*, *pcbDescription*, *DBMSType*)

Input Variables

The following table describes the input variables.

Type	Argument	Description
HENV	<i>henv</i>	Environment handle.
UWORD	<i>fDirection</i>	Determines which data source to return information about. <i>fDirection</i> can be: SQL_FETCH_FIRST SQL_FETCH_NEXT
SWORD	<i>cbDSNMax</i>	Maximum length of the data source name buffer, in bytes.
SWORD	<i>cbDescriptionMax</i>	Maximum length of the configuration information buffer, in bytes.

SQLDataSources Input Variables

Output Variables

The following table describes the output variables.

Type	Argument	Description
UCHAR *	<i>szDSN</i>	Pointer to the data source name buffer.
SWORD *	<i>pcbDSN</i>	Number of bytes available to return in <i>szDSN</i> . If <i>pcbDSN</i> > <i>cbDSNMax</i> , the name is truncated and SQL_SUCCESS_WITH_INFO is returned.
UCHAR *	<i>szDescription</i>	Pointer to the configuration information buffer.
SWORD *	<i>pcbDescription</i>	Number of bytes available to return in <i>szDescription</i> . If <i>pcbDescription</i> > <i>cbDescriptionMax</i> , the configuration information is truncated and SQL_SUCCESS_WITH_INFO is returned.
SWORD *	<i>DBMSType</i>	Database type, which can be: 1 – UniVerse 2 – UniData 0 – Neither UniVerse nor UniData 999 – Not specified

SQLDataSources Output Variables

Description

An application can call SQLDataSources multiple times to retrieve all data source names. When there are no more data source names, UCI returns SQL_NO_DATA_FOUND. If SQLDataSources is called with SQL_FETCH_NEXT immediately after it returns SQL_NO_DATA_FOUND, it returns the first data source name.

Return Values

SQL_SUCCESS
SQL_SUCCESS_WITH_INFO
SQL_ERROR
SQL_NO_DATA_FOUND

SQLSTATE Values

When `SQLDataSources` returns `SQL_ERROR` or `SQL_SUCCESS_WITH_INFO`, you can call `SQLException` to get the associated `SQLSTATE` value. Common `SQLSTATE` values returned are:

SQLSTATE	Description
01004	Data truncated. Either the data source name buffer or the configuration information buffer is too small. Use the other arguments to determine which one is too small.
IA003	Bad argument. You must use either <code>SQL_FETCH_FIRST</code> or <code>SQL_FETCH_NEXT</code> .
IM998	UCI configuration file error. Either the configuration file does not exist, or an error was found in the file.
S1001	Memory allocation failure.

SQLDataSources SQLSTATE Values

SQLDescribeCol

SQLDescribeCol returns limited descriptive information (column name, data type, precision, scale, and nullability) about a specified column. This call can be used only after the statement has been prepared by an **SQLPrepare** or **SQLExecDirect** call. The **SQLColAttributes** function provides access to more information than **SQLDescribeCol**.

Syntax

RETCODE SQLDescribeCol (*hstmt*, *icol*, *szColName*, *cbColNameMax*, *pcbColName*, *pfSqlType*, *pcbColDef*, *pibScale*, *pfNullable*)

Input Variables

The following table describes the input variables.

Type	Argument	Description
HSTMT	<i>hstmt</i>	Statement handle.
UWORD	<i>icol</i>	Column number to describe, numbered sequentially from left to right starting at 1.
SWORD	<i>cbColNameMax</i>	Maximum length of the <i>szColName</i> buffer.

SQLDescribeCol Input Variables

Output Variables

The following table describes the output variables.

Type	Argument	Description
UCHAR *	<i>szColName</i>	Pointer to storage for the column name. If the column name is an expression, the expression is returned.
SWORD *	<i>pcbColName</i>	Total number of bytes available to return in <i>szColName</i> excluding the null byte. If this value is larger than <i>cbColNameMax</i> , the returned string is truncated to <i>cbColNameMax</i> – 1 bytes.
SWORD *	<i>pfSqlType</i>	The SQL data type of the column. See SQL Data Types Supported in Chapter 7, “Data Types,” for a list of SQL data types that can be returned.
UDWORD *	<i>pcbColDef</i>	The precision of the column. See Precision, Scale, and Display Size Definitions in Chapter 7, “Data Types,” for a discussion of the precision and scale of a column. Can be 0.
SWORD *	<i>pibScale</i>	The scale of the column if it is SQL data type DECIMAL or NUMERIC, otherwise it is 0.
SWORD	<i>pfNullable</i>	Indicates if the column can contain nulls, and contains either SQL_NO_NULLS or SQL_NULLABLE. If the column is an expression, SQL_NULLABLE_UNKNOWN is returned.

SQLDescribeCol Output Variables

Description

The information returned by SQLDescribeCol is a subset of the information returned by SQLColAttributes and is of limited use to database applications. In first-normal-form databases constrained to use fixed-length columns, the precision of a column is of fundamental importance to the database and can be viewed as an internal CHECK constraint on the data coming into the column. However, the database, being more flexible, has no need to enforce such constraints, and although a column may be defined as CHAR(30), the database does not prevent a user application from entering longer strings.

Return Values

SQL_SUCCESS
SQL_ERROR
SQL_INVALID_HANDLE
SQL_SUCCESS_WITH_INFO

SQLSTATE Values

The following table describes the SQLDescribeCol SQLSTATE values.

SQLSTATE	Description
S1000	General error for which no specific SQLSTATE code has been defined.
S1001	Memory allocation failure.
S1002	The value in <i>icol</i> is greater than the number of columns in the result set.
S1009	<i>szColName</i> , <i>pcbColName</i> , or <i>pfSqlType</i> is null.
S1010	Function sequence error. SQLDescribeCol was called before calling either SQLPrepare or SQLExecDirect for <i>hstmt</i> .
S1090	The value specified in <i>cbColNameMax</i> is less than or equal to 0.
01004	The <i>szColName</i> buffer was too short for the name to be returned and the result was truncated (SQL_SUCCESS_WITH_INFO).
24000	The SQL statement associated with <i>hstmt</i> did not return a result set.

SQLDescribeCol SQLSTATE Values

SQLDisconnect

SQLDisconnect closes the connection associated with a particular connection handle. You cannot use SQLDisconnect inside a transaction.

Syntax

RETCODE SQLDisconnect (*hdbc*)

Input Variable

The following table describes the input variable.

Type	Argument	Description
HDBC	<i>hdbc</i>	Connection handle to be closed.

SQLDisconnect Input Variable

Description

An application must explicitly issue a COMMIT or ROLLBACK statement for any active transactions before attempting to issue an SQLDisconnect call. If an application should terminate either normally or abnormally with transactions still active, an implicit ROLLBACK statement is executed at the server.

Return Values

SQL_SUCCESS
SQL_ERROR
SQL_INVALID_HANDLE
SQL_SUCCESS_WITH_INFO

SQLSTATE Values

The following table describes the SQLDisconnect SQLSTATE values.

SQLSTATE	Description
S1000	General error for which no specific SQLSTATE code has been defined.
S1001	Memory allocation failure.
S1010	Function sequence error. The connection handle has a statement handle that is currently being executed by the server.
01002	An error occurred during the disconnect, but the connection has been broken (SQL_SUCCESS_WITH_INFO).
08003	The connection specified by <i>hdbc</i> has not been established.
25000	An active transaction is present on the connection and remains active following this error return.

SQLDisconnect SQLSTATE Values

SQLError

SQLError returns error information and status from the server.

Syntax

RETCODE SQLError (*henv*, *hdbc*, *hstmt*, *szSqlState*, *pfNativeError*, *szErrorMsg*, *cbErrorMsgMax*, *pcbErrorMsg*)

Input Variables

The following table describes the input variables.

Type	Argument	Description
HENV	<i>henv</i>	Environment handle or SQL_NULL_HENV.
HDBC	<i>hdbc</i>	Connection handle or SQL_NULL_HDBC.
HSTMT	<i>hstmt</i>	Statement handle or SQL_NULL_HSTMT.
SWORD	<i>cbErrorMsgMax</i>	Maximum length of the <i>szErrorMsg</i> buffer.

SQLError Input Variables

Output Variables

The following table describes the output variables.

Type	Argument	Description
UCHAR *	<i>szSqlState</i>	Pointer to storage for a null-terminated string containing the SQLSTATE (storage for six characters is required).

SQLError Output Variables

Type	Argument	Description
SDWORD *	<i>pfNativeError</i>	Pointer to the database error code number.
UCHAR *	<i>szErrorMsg</i>	Pointer to storage for error text (storage for 256 characters is recommended).
SWORD *	<i>pcbErrorMsg</i>	Pointer to the total number of bytes (excluding the null terminator) available to return in <i>szErrorMsg</i> . If this exceeds <i>cbErrorMsgMax</i> , the message text returned is truncated to <i>cbErrorMsgMax</i> , and SQL_SUCCESS_WITH_INFO is returned.

SQLError Output Variables (Continued)

Description

Typically, an application calls SQLError whenever a previous call returns SQL_ERROR or SQL_SUCCESS_WITH_INFO, but it can be used after any call.

Error status information can be retrieved for an error associated with an environment, a connection, or a statement as follows:

To retrieve errors associated with...	Do this...
Environment	Pass the environment's <i>henv</i> , and pass SQL_NULL_HDBC in <i>hdbc</i> and SQL_NULL_HSTMT in <i>hstmt</i> . The error status of the ODBC function most recently called with <i>henv</i> is returned.
Connection	Pass the connection's <i>hdbc</i> , and pass SQL_NULL_HSTMT in <i>hstmt</i> (any <i>henv</i> argument is ignored). The error status of the ODBC function most recently called with <i>hdbc</i> is returned.
Statement	Pass the statement's <i>hstmt</i> (any <i>henv</i> and <i>hdbc</i> arguments are ignored). The error status of the ODBC function most recently called with <i>hstmt</i> is returned.

Error Status Information

Because more than one error or warning message can be posted for a single UCI call, an application should call **SQLError** until the function returns the value SQL_NO_DATA_FOUND. For each error, SQL_SUCCESS is returned and the error is removed from the error list.

The error text is returned in the form:

[IBM] [*component name*] *error message text*

For example:

[IBM] [UniVerse] . . .

[IBM] [RPC] . . .

All errors for a given handle are removed when `SQL_Error` is called repeatedly for that handle or when that handle is used in a subsequent function call. However, errors for a given handle are not removed by a call to a function using an associated handle of a different type.

SQLSTATE values are always five characters long, so *szSqlState* must point to storage for a maximum of six characters. Error messages vary widely in length, so the user application must allocate storage and inform UCI how much storage has been allocated via the *cbErrorMsgMax* parameter. It is recommended that at least 256 characters be reserved for error messages. For example:

```
#define          cbErrorMsgMax      256
SCHAR          szSqlState[6];
SCHAR          szErrorMsg[ cbErrorMsgMax ];
```

Return Values

SQL_SUCCESS

SQL_ERROR

SQL_INVALID_HANDLE

SQL_SUCCESS_WITH_INFO

SQL_NO_DATA_FOUND

SQLSTATE Values

The `SQL_Error` function does not post errors for itself. However, if an error message to be returned is larger than the buffer allocated to store it, the value `SQL_SUCCESS_WITH_INFO` is returned from the `SQL_Error` call, and the buffer contains truncated error text.

SQLExecDirect

SQLExecDirect executes a preparable SQL statement or procedure call using the current values of any parameter markers that are set up for the statement.

Syntax

RETCODE SQLExecDirect (*hstmt*, *szSqlStr*, *cbSqlStr*)

Input Variables

The following table describes the input variables.

Type	Argument	Description
HSTMT	<i>hstmt</i>	Statement handle.
UCHAR *	<i>szSqlStr</i>	Pointer to either an SQL statement or a call to an SQL procedure, to be executed at the data source.
SDWORD	<i>cbSqlStr</i>	Length of <i>szSqlStr</i> .

SQLExecDirect Input Variables

Description

This function both prepares and executes an SQL statement or procedure call. It differs from SQLExecute in that SQLExecDirect does not require a call to SQLPrepare. Use SQLExecDirect as the easiest way to execute an SQL statement or procedure when you do not need to execute it repeatedly.

The SQL statement or procedure call can contain parameter markers, which must be defined to UCI by an SQLBindParameter call before issuing SQLExecDirect. Before the SQL statement or procedure is executed, the current values of the markers are delivered to the server. Any data conversion problems caused by erroneous parameter marker values are detected when this call is given.

If the SQL statement is a SELECT statement or procedure call, there could be data that can be retrieved by SQLFetch as a result of executing SQLExecDirect. You can issue a call to SQLNumResultCols to determine if any result columns were produced by executing the SQL statement or procedure.

Calling SQL Procedures

To call an SQL procedure, use one of the following syntaxes:

```
call procedure [(parameter [, parameter] ...)]
call procedure [argument [argument] ...]
```

The following table describes each parameter of the syntax.

Parameter	Description
<i>procedure</i>	Name of the procedure. If this name contains characters other than letters or numbers, enclose the name in double quotation marks. To embed a single quotation mark in the procedure name, use two consecutive double quotation marks.
<i>parameter</i>	Either a literal value or a parameter marker that indicates where to insert values to send to or receive from the data source. Programmatic SQL uses a ? (question mark) as a parameter marker. You cannot use SQLBindMvParameter to bind parameter marks used in a call statement. Use parameters only if the procedure is a subroutine. The number and order of parameters must correspond to the number and order of the subroutine arguments.
<i>argument</i>	Any valid keyword, literal, or other token you can use in a database command line.

call Parameters

If SQLBindParameter defines a procedure's parameter type as SQL_PARAM_OUTPUT or SQL_PARAM_INPUT_OUTPUT, values are returned to the specified program variables.

When SQLExecDirect calls a procedure, it does not begin a transaction. If a transaction is active when a procedure is called, the current transaction nesting level is maintained.

Return Values

SQL_SUCCESS
SQL_ERROR
SQL_INVALID_HANDLE

SQLSTATE Values

The following table describes the SQLExecDirect SQLSTATE values.

SQLSTATE	Description
IA000	An SQL statement was not executed because it contains the EXPLAIN keyword. The EXPLAIN output is returned as error message text (see SQLError).
S0001	Table or view already exists. Several database error codes can produce this SQLSTATE. The specific reason is returned in the native error code argument of the SQLError call.
S0002	Table or view not found. Several database error codes can produce this SQLSTATE. The specific reason is returned in the native error code argument of the SQLError call.
S0021	Column already exists. Several database error codes can produce this SQLSTATE. The specific reason is returned in the native error code argument of the SQLError call.
S0022	Column not found. Several database error codes can produce this SQLSTATE. The specific reason is returned in the native error code argument of the SQLError call.
S1000	General error for which no specific SQLSTATE code has been defined.
S1001	Memory allocation failure.
S1008	Cancelled. Execution of the statement was stopped by an SQLCancel call.
S1010	Function sequence error. The <i>hstmt</i> specified is currently executing an SQL statement.
01004	A data value in a parameter marker was truncated, resulting in loss of precision.

SQLExecDirect SQLSTATE Values

SQLSTATE	Description
07001	Not all parameter markers in the SQL statement have been specified with SQLBindParameter.
21S01	Insert value list does not match the value list.
21S02	Number of columns in derived table does not match the column list.
22001	A parameter marker value was sent, but fractional truncation occurred.
22005	A value in a parameter marker is incompatible with the SQL data type of that marker.
23000	Integrity constraint violation.
24000	Invalid cursor state. Results are still pending from the previous SQL statement. Use SQLCancel to clear the results.
40001	An SQL statement with the NOWAIT keyword was not executed because it encountered a lock conflict. The application may choose to sleep and retry the operation a few times before giving up.
42000	Syntax error or access violation. This can happen for a variety of reasons. The native error code returned by the SQLError call indicates the specific database error that occurred.
SQLExecDirect SQLSTATE Values (Continued)	

SQLExecute

SQLExecute executes an SQL statement that has been prepared with **SQLPrepare**, using the current values of any parameter markers.

Syntax

RETCODE SQLExecute (*hstmt*)

Input Variable

The following table describes the input variable.

Type	Argument	Description
HSTMT	<i>hstmt</i>	Statement handle.

SQLExecute Input Variable

Description

This function is commonly used for such operations as inserting multiple rows into an SQL table.

You must call SQLPrepare to prepare the SQL statement before you can use SQLExecute. If the SQL statement specified in the SQLPrepare call contains parameter markers, you must also issue an SQLBindParameter or SQLSetParam call for each marker in the SQL statement before calling SQLExecute. After you load the parameter marker variables with data to send to the data source, you can issue a call to SQLExecute. By setting new values in the parameter marker variables and calling SQLExecute, new data values are sent to the data source and the SQL statement is executed using those values.

If the SQL statement uses parameter markers, SQLExecute performs any data conversions required by the SQLSetParam calls for the parameter markers.

Return Values

SQL_SUCCESS
SQL_ERROR
SQL_INVALID_HANDLE

SQLSTATE Values

The following table describes the SQLExecute SQLSTATE values.

SQLSTATE	Description
IA000	An SQL statement was not executed because it contains the EXPLAIN keyword. The EXPLAIN output is returned as error message text (see SQLError).
S1000	General error for which no specific SQLSTATE code has been defined.
S1001	Memory allocation failure.
S1008	Cancelled. Execution of the statement was stopped by an SQLCancel call.
S1010	Function sequence error. Either the SQL statement has not been prepared, or there is an SQL statement already executing on the statement handle.
01004	A data value in a parameter marker was truncated, resulting in loss of precision.
07001	Not all parameter markers in the SQL statement have been bound with SQLBindParameter.
22005	A value in a parameter marker is incompatible with the SQL data type of that marker.
23000	Integrity constraint violation.
24000	Invalid cursor state. Results are still pending from the previous SQL statement. Use SQLCancel to clear the results.
40001	An SQL statement with the NOWAIT keyword was not executed because it encountered a lock conflict. The application may choose to sleep and retry the operation a few times before giving up.

SQLExecute SQLSTATE Values

SQLFetch

SQLFetch returns the next row of data from a result set. Column data for all columns specified in a preceding SQLBindCol call is returned into the variables that were bound to the columns in the result set. If a column was bound with SQLBindMvCol, UCI allocates a correctly sized C_ARRAY structure and returns its address in the *pCArray* argument of the SQLBindMvCol call for each column.

Syntax

RETCODE SQLFetch (*hstmt*)

Input Variable

The following table describes the input variable.

Type	Argument	Description
HSTMT	<i>hstmt</i>	Statement handle.

SQLFetch Input Variable

Description

This function retrieves the next row’s column values from the result set and puts them into the variables specified with SQLBindCol or SQLBindMvCol. If the data was bound by a call to SQLBindCol and the data returned from the server is found to be multivalued, only the first value is returned in the bound parameter, along with a status of SQL_SUCCESS_WITH_INFO. Call SQLGetData to get subsequent values.

SQLFetch performs any required data conversions (see [C Data Types Supported](#) in Chapter 7, “Data Types,” for details). SQL_SUCCESS_WITH_INFO is returned if numeric data is truncated or rounded when converting SQL values to database values.

Each SQLFetch call logically advances the cursor to the next row in the result set (the database supports only forward scrolling cursors). When there is no more data to retrieve, SQL_NO_DATA_FOUND is returned.

Use SQLFetch only when a result set is pending at the data source.

Transactional Notes

Two rules govern the fetching of data in manual-commit mode:

- You must fetch data at the same transaction isolation level as that at which the original SELECT statement was executed:

```
SQLTransact ( SQL_HULL_HENV, hdbc,  
              SQL_BEGIN_TRANSACTION + SQL_TXN_READ_COMMITTED);  
SQLBindCol ( hstmt, 1, ...);  
SQLBindCol ( hstmt, 2, ...);  
SQLExecDirect ( hstmt, "SELECT COL1, COL2 FROM TABLE"  
);  
.  
.  
.  
SQLTransact ( SQL_HULL_HENV, hdbc,  
              SQL_BEGIN_TRANSACTION);  
  
SQLFetch ( hstmt );
```

The previous code sequence is legal. The following sequence fails because the SELECT statement is executed in a transaction started at isolation level READ_COMMITTED, whereas the SQLFetch is executed from a higher transaction isolation level (REPEATABLE_READ). Because the locking strategy for the SELECT statement is determined when SELECT is executed, trying to fetch data at a higher isolation level is not allowed because the data would be fetched using the lower level.

```
SQLTransact ( SQL_HULL_HENV, hdbc,  
              SQL_BEGIN_TRANSACTION + SQL_TXN_READ_COMMITTED);  
SQLBindCol ( hstmt, 1, ...);  
SQLBindCol ( hstmt, 2, ...);  
SQLExecDirect ( hstmt, "SELECT COL1, COL2 FROM TABLE"  
);  
.  
.  
.  
SQLTransact ( SQL_HULL_HENV, hdbc,  
              SQL_BEGIN_TRANSACTION + SQL_TXN_REPEATABLE_READ);  
  
SQLFetch ( hstmt );
```

- Once COMMIT or ROLLBACK is issued for a transaction in which a SELECT statement was executed, no further fetches from that cursor are permitted because the cursor has been closed by COMMIT or ROLLBACK. Attempting to fetch from the closed cursor returns an SQLSTATE of 24000.

Return Values

SQL_SUCCESS
SQL_ERROR
SQL_INVALID_HANDLE
SQL_SUCCESS_WITH_INFO
SQL_NO_DATA_FOUND

SQLSTATE Values

The following table describes the SQLFetch SQLSTATE values.

SQLSTATE	Description
IM981	One value was returned from multivalued data bound by SQLBindCol. The condition returns SQL_SUCCESS_WITH_INFO.
S1000	General error for which no specific SQLSTATE code has been defined.
S1001	Memory allocation failure.
S1002	Invalid column number. <i>icol</i> is 0 or is greater than the number of columns in the result set.
S1010	Function sequence error. Either <i>hstmt</i> is not in an executed state, or there is an SQL statement already executing on <i>hstmt</i> .
01004	One or more columns was truncated. If string data, data is truncated on the right. If numeric data, the fractional part is truncated. The condition causes a return of SUCCESS_WITH_INFO.
07006	Data could not be converted into the type specified by <i>fCType</i> in the SQLBindCol call.
24000	No results are pending on <i>hstmt</i> .
40001	The next row of results from an SQL SELECT with the NOWAIT keyword was not fetched because a lock conflict was encountered. The application may choose to sleep and retry the SQLFetch a few times before giving up.

SQLFetch SQLSTATE Values

SQLFreeConnect

SQLFreeConnect releases a connection handle and frees all resources associated with it.

Syntax

RETCODE SQLFreeConnect (*hdbc*)

Input Variable

The following table describes the input variable.

Type	Argument	Description
HDBC	<i>hdbc</i>	Connection handle to be freed.

SQLFreeConnect Input Variable

Description

You must use SQLDisconnect to disconnect the connection handle before you release the connection environment with SQLFreeConnect, otherwise an error is returned.

Return Values

SQL_SUCCESS
SQL_ERROR
SQL_INVALID_HANDLE

SQLSTATE Values

The following table describes the SQLFreeConnect SQLSTATE values.

SQLSTATE	Description
S1000	General error for which no specific SQLSTATE code has been defined.
S1010	Function sequence error. The connection on <i>hdbc</i> is still active.

SQLFreeConnect SQLSTATE Values

SQLFreeEnv

SQLFreeEnv frees the environment handle and releases memory associated with it.

Syntax

RETCODE SQLFreeEnv (*henv*)

Input Variable

The following table describes the input variable.

Type	Argument	Description
HENV	<i>henv</i>	Environment handle to be freed.

SQLFreeEnv Input Variable

Description

You must use **SQLFreeEnv** to release all environment handles attached to the ODBC environment before you release the environment with SQLFreeConnect; otherwise an error is returned.

Return Values

SQL_SUCCESS
SQL_ERROR
SQL_INVALID_HANDLE

SQLSTATE Values

The following table describes the SQLFreeEnv SQLSTATE values.

SQLSTATE	Description
S1000	General error for which no specific SQLSTATE code has been defined.
S1010	Function sequence error. The environment has at least one allocated <i>hdbc</i> .

SQLFreeEnv SQLSTATE Values

SQLFreeMem

SQLFreeMem releases memory allocated by UCI software, thus preventing problems caused by calling different memory managers from the same application.

Syntax

RETCODE SQLFreeMem (*memptr*)

Input Variable

The following table describes the input variable.

Type	Argument	Description
PTR	<i>memptr</i>	Address of the memory that SQLBindMvCol allocated when data was fetched from the server.

SQLFreeMem Input Variable

Description

The SQLBindMvCol function allocates memory as necessary to hold multivalued data, and returns a pointer to the allocated memory. The user is responsible for freeing the allocated memory, using the SQLFreeMem function.

SQLFreeStmt

SQLFreeStmt allows you to perform one of several operations on a statement handle, depending on the option chosen.

Syntax

RETCODE SQLFreeStmt (*hstmt*, *fOption*)

Input Variables

The following table describes the input variables.

Type	Argument	Description
HSTMT	<i>hstmt</i>	Statement handle.
UWORD	<i>fOption</i>	One of the following values: SQL_CLOSE, SQL_DROP, SQL_RESET_PARAMS, or SQL_UNBIND (see “Description”).

SQLFreeStmt Input Variables

Description

Use this function at the end of processing to free resources used by an SQL statement, to reset parameter marker bindings, or unbind column variables.

If your program uses the same SQL statement environment to execute different SQL statements, you can use SQLFreeStmt either with the SQL_CLOSE option, which should be sufficient in most cases, or with the SQL_DROP option. In the latter case, you need to call SQLAllocStmt to reallocate a new SQL statement environment.

It is good practice to issue a call to SQLFreeStmt with the SQL_CLOSE option when all results have been read from the data source, even if the SQL statement environment will not be reused immediately for another SQL statement.

fOption can be any one of the following:

<i>fOption</i>	Description
SQL_CLOSE	Closes any open cursor associated with the SQL statement environment and discards pending results at the data source. Using the SQL_CLOSE option cancels the current query. All parameter markers and columns remain bound to the variables specified in the SQLBindCol and SQLBindParameter (or SQLSetParam) calls. No more data can be fetched from this <i>hstmt</i> until the SQL statement associated with the <i>hstmt</i> is executed again with SQLExecute. Note that reexecuting an <i>hstmt</i> which has not been prepared is not permitted. This option is functionally equivalent to SQLCancel.
SQL_DROP	In addition to including all available options, SQL_DROP also deallocates the statement environment.
SQL_RESET_PARAMS	Releases all parameter marker variables set by SQLBindParameter.
SQL_UNBIND	Releases all bound column variables bound by SQLBindCol or SQLBindMvCol for this SQL statement environment.
SQLFreeStmt <i>fOptions</i>	

Return Values

SQL_SUCCESS
SQL_ERROR
SQL_INVALID_HANDLE

SQLSTATE Values

The following table describes the SQLFreeStmt SQLSTATE values.

SQLSTATE	Description
S1000	General error for which no specific SQLSTATE code has been defined.
S1092	<i>fOption</i> is not a valid value.
SQLFreeStmt SQLSTATE Values	

SQLGetData

SQLGetData retrieves data that exceeds the buffer space allocated for it. It also retrieves data from columns in the result set that were not bound.

Syntax

RETCODE SQLGetData (*hstmt*, *icol*, *fCType*, *rgbValue*, *cbValueMax*, *pcbValue*)

Input Variables

The following table describes the input variables.

Type	Argument	Description
HSTMT	<i>hstmt</i>	Statement handle.
UWORD	<i>icol</i>	Column number in the result set, numbered left to right starting at 1.
SWORD	<i>fCType</i>	C data type of the result for the column. See C Data Types Supported in Chapter 7, “Data Types,” for a complete list.
SDWORD	<i>cbValueMax</i>	If nonzero, this value specifies, for binary and character data, the maximum size allocated for the <i>rgbValue</i> buffer.

SQLGetData Input Variables

Output Variables

The following table describes the output variables.

Type	Argument	Description
PTR	<i>rgbValue</i>	Address of the buffer into which this call reads the data. A value of 0 causes an error return from the function.
SDWORD *	<i>pcbValue</i>	SQL_NULL_DATA if the cell contains the null value. SQL_BAD_DATA if the data could not be converted, or the total number of bytes available to return in <i>rgbValue</i> .

Description

This function is used in either of two circumstances:

- To get data from a column that returns a data truncation error because the application's buffer is too small to contain all of the column data.
- To get data from a column not bound by SQLBindCol. (You cannot use SQLGetData on columns that have been bound using SQLBindMvCol because fetching such columns always allocates enough memory automatically.)

Note: When retrieving data that does not fit into a buffer, SQLGetData retrieves the remaining data. It does not retrieve from the beginning of the text.

Return Values

SQL_SUCCESS
SQL_ERROR
SQL_INVALID_HANDLE
SQL_NO_DATA_FOUND



SQLSTATE Values

The following table describes the SQLGetData SQLSTATE values.

SQLSTATE	Description
IM979	The column was previously bound using SQLBindMvCol. Using the single-valued SQLGetData function is therefore illegal.
S1000	General error for which no specific SQLSTATE code has been defined.
S1001	Memory allocation failure.
S1002	Either <i>icol</i> is 0 or exceeds the number of columns in the result set.
S1009	The <i>rgbValue</i> or <i>pcbValue</i> argument is 0.
S1010	<i>hstmt</i> is not in an executed state. You must invoke SQLExecDirect or SQLExecute before fetching data. or No SQLFetch has been issued for <i>hstmt</i> to position the cursor on a data row prior to SQLGetData.
01004	Not all data for the column could be retrieved in this operation.
07006	Data could not be converted into the type specified by <i>fCType</i> in the SQLGetData call.
24000	No results are pending on <i>hstmt</i> .

SQLGetData SQLSTATE Values

Example

The following pseudo-code shows how an application might deal with a situation where the original buffer allocated for the data was too small:

```
SDWORD cbValueMax = 512, pcbValue = 0;
buffer = malloc (cbValueMax);
SQLBindCol (hstmt, 1, SQL_C_STRING, buffer, cbValueMax,
&pcbValue);
if (SQLFetch ( ) == 01004)
{
    buffer = realloc(buffer, pcbValue); /* Get full-size buffer *
    cbValueMax = pcbValue;
    /* Now transfer remaining data into correct sized buffer
       at the appropriate location in that buffer */
    SQLGetData (hstmt, 1, SQL_C_STRING, &buffer[cbValueMax],
        cbValueMax, &pcbValue);
}
```

SQLGetFunctions

SQLGetFunctions returns information regarding whether a driver supports certain functions.

Syntax

RETCODE SQLGetFunctions (*hdbc*, *fFunction*, *pfExists*)

Input Variables

The following table describes the input variables.

Type	Argument	Description
HDBC	<i>hdbc</i>	Connection handle.
UWORD	<i>fFunction</i>	A <i>#define</i> value that identifies an ODBC function. These are listed under “fFunction Values.”

SQLGetFunctions Input Variables

Output Variable

The following table describes the output variable.

Type	Argument	Description
UWORD *	<i>pfExists</i>	A single Boolean TRUE value if the function is supported.

SQLGetFunctions Output Variable

Description

This function is implemented wholly within UCI and is for tools vendors who want to use general code against UCI to verify that certain functions are available before deciding how to implement something.

fFunction Values

The following *fFunction* values are recognized and are defined in the *UCI.h* include file:

SQL_API_SQLALLOCCONNECT
SQL_API_SQLALLOCENV
SQL_API_SQLALLOCSTMT
SQL_API_SQLBINDCOL
SQL_API_SQLCANCEL
SQL_API_SQLCOLATTRIBUTES
SQL_API_SQLCONNECT
SQL_API_SQLDESCRIBECOL
SQL_API_SQLERROR
SQL_API_SQLEXECDIRECT
SQL_API_SQLEXECUTE
SQL_API_SQLFETCH
SQL_API_SQLFREECONNECT
SQL_API_SQLFREEENV
SQL_API_SQLFREESTMT
SQL_API_SQLGETCURSORNAME
SQL_API_SQLNUMRESULTCOLS
SQL_API_SQLPREPARE
SQL_API_SQLROWCOUNT
SQL_API_SQLSETCURSORNAME
SQL_API_SQLSETPARAM
SQL_API_SQLTRANSACT

Level 1 Functions

SQL_API_SQLCOLUMNS
SQL_API_SQLDRIVERCONNECT
SQL_API_SQLGETCONNECTIONOPTION
SQL_API_SQLGETDATA
SQL_API_SQLGETFUNCTIONS
SQL_API_SQLGETINFO
SQL_API_SQLGETSTMTOPTION
SQL_API_SQLGETTYPEINFO
SQL_API_SQLPARAMDATA
SQL_API_SQLSETCONNECTIONOPTION
SQL_API_SQLSETSTMTOPTION
SQL_API_SQLSPECIALCOLUMNS
SQL_API_SQLSTATISTICS
SQL_API_SQLTABLES

Level 2 Functions

SQL_API_SQLBROWSECONNECT
SQL_API_SQLCOLUMNPRIVILEGES
SQL_API_SQLDATASOURCES
SQL_API_SQLDESCRIBEPARAM
SQL_API_SQLEXTENDEDFETCH
SQL_API_SQLFOREIGNKEYS
SQL_API_SQLMORERESULTS
SQL_API_SQLNATIVESQL
SQL_API_SQLNUMPARAMS
SQL_API_SQLPARAMOPTIONS
SQL_API_SQLPRIMARYKEYS
SQL_API_SQLPROCEDURECOLUMNS
SQL_API_SQLPROCEDURES
SQL_API_SQLSETPOS
SQL_API_SQLSETSCROLLOPTIONS
SQL_API_SQLTABLEPRIVILEGES

ODBC 2.0 Additions

SQL_API_SQLBINDPARAMETER
SQL_API_SQLDRIVERS

Return Values

SQL_SUCCESS
SQL_ERROR
SQL_INVALID_HANDLE
SQL_SUCCESS_WITH_INFO

SQLSTATE Values

The following table describes the SQLGetFunctions SQLSTATE values.

SQLSTATE	Description
S1000	General error for which no specific SQLSTATE code has been defined.
S1010	Function sequence error. SQLGetFunctions was called before the connection was made.
S1095	An invalid <i>fFunction</i> was requested.

SQLGetFunctions SQLSTATE Values

SQLGetInfo

SQLGetInfo returns general information about the driver and the capabilities of the database release.

Syntax

RETCODE SQLGetInfo (*hdbc*, *fInfoType*, *rgbInfoValue*, *cbInfoValueMax*, *pcbInfoValue*)

Input Variables

The following table describes the input variables.

Type	Argument	Description
HDBC	<i>hdbc</i>	Connection handle.
UWORD	<i>fInfoType</i>	Type of information wanted (refer to fInfoType Values).
SWORD	<i>cbInfoValueMax</i>	Maximum length of the buffer, including any null terminator.

SQLGetInfo Input Variables

Output Variables

The following table describes the output variables.

Type	Argument	Description
PTR	<i>rgbInfoValue</i>	Pointer to storage for the returned information.
SWORD *	<i>pcbInfoValue</i>	Total number of bytes returned, excluding any null terminator.

SQLGetInfo Output Variables

Description

This function supports all of the possible requests for information defined in the ODBC 2.0 specification. The *#defines* for *fInfoType* are contained in the *UCI.h* include file.

For character-type data, *SQLGetInfo* checks *cbInfoValueMax* and truncates the returned value as needed, and sets *pcbInfoValue* to the total number of bytes (excluding any null terminator) actually returned, *not* the total number of bytes available to return. Thus, if the number of bytes available for return is greater than the buffer space allocated, there is no indication that bytes are missing.

For noncharacter-type data, *cbInfoValueMax* is ignored, and *pcbInfoValue* is not set.

fInfoType Values

The following table lists the valid values for *fInfoType* and documents the results returned by the database.

<i>fInfoType</i>	<i>rgbInfoValue</i>	Type
SQL_ACTIVE_CONNECTIONS	0	16-bit integer
SQL_ACTIVE_STATEMENTS	0	16-bit integer
SQL_DATA_SOURCE_NAME	(<i>szDSN</i> from <i>SQLConnect</i>)	char string
SQL_DRIVER_HDBC	0	32-bit value
SQL_DRIVER_HENV	0	32-bit value
SQL_DRIVER_HLIB	0	32-bit value
SQL_DRIVER_HSTMT	0	32-bit value
SQL_DRIVER_NAME	(empty string)	char string
SQL_DRIVER_ODBC_VER	“03.00”	char string
SQL_DRIVER_VER	(empty string)	char string
SQL_FETCH_DIRECTION	SQL_FD_FETCH_NEXT	32-bit bitmask
SQL_FILE_USAGE	SQL_FILE_NOT_SUPPORTED	16-bit integer

fInfoType Values

<i>fInfoType</i>	<i>rgbInfoValue</i>	Type
SQL_GETDATA_EXTENSIONS	SQL_GD_ANY_COLUMN SQL_GD_ANY_ORDER SQL_GD_BOUND	32-bit bitmask
SQL_LOCK_TYPES	0	32-bit bitmask
SQL_ODBC_API_CONFORMANCE	SQL_OAC_NONE	16-bit integer
SQL_ODBC_SAG_CLI_CONFORMANCE	SQL_OSSC_NOT_COMPLIANT	16-bit integer
SQL_ODBC_VER	(empty string)	char string
SQL_POS_OPERATIONS	0	32-bit bitmask
SQL_ROW_UPDATES	“N”	char string
SQL_SEARCH_PATTERN_ESCAPE	(empty string)	char string
SQL_SERVER_NAME	(empty string)	char string
DBMS Product Information		
SQL_DATABASE_NAME (not in <i>UCI.h</i>)	(deprecated)	
SQL_DBMS_NAME	“UNIVERSE” or “UNIDATA”	char string
SQL_DBMS_VER	(current release number, for example, 09.04.0001)	char string
Data Source Information		
SQL_ACCESSIBLE_PROCEDURES	“N”	char string
SQL_ACCESSIBLE_TABLES	“N”	char string
SQL_BOOKMARK_PERSISTENCE	0	32-bit bitmask
SQL_CONCAT_NULL_BEHAVIOR	0	16-bit integer
SQL_CURSOR_COMMIT_BEHAVIOR	SQL_CB_CLOSE	16-bit integer
SQL_DATA_SOURCE_READ_ONLY	“N”	char string
SQL_DEFAULT_TXN_ISOLATION	SQL_TXN_READ_COMMITTED	32-bit bitmask

***fInfoType* Values (Continued)**

<i>fInfoType</i>	<i>rgbInfoValue</i>	Type
SQL_MULT_RESULT_SETS	“N”	char string
SQL_MULTIPLE_ACTIVE_TXN	“Y”	char string
SQL_NEED_LONG_DATA_LEN	“N”	char string
SQL_NULL_COLLATION	SQL_NC_HIGH	16-bit integer
SQL_OWNER_TERM	“Owner”	char string
SQL_PROCEDURE_TERM	“procedure”	char string
SQL_QUALIFIER_TERM	“Schema”	char string
SQL_SCROLL_CONCURRENCY	SQL_SCCO_LOCK	32-bit bitmask
SQL_SCROLL_OPTIONS	SQL_SO_FORWARD_ONLY	32-bit bitmask
SQL_STATIC_SENSITIVITY	0	32-bit bitmask
SQL_TABLE_TERM	“Table”	char string
SQL_TXN_CAPABLE	SQL_TC_DML	16-bit integer
SQL_TXN_ISOLATION_OPTION	SQL_TXN_READ_UNCOMMITTED SQL_TXN_READ_COMMITTED SQL_TXN_REPEATABLE_READ SQL_TXN_SERIALIZABLE (but not SQL_TXN_VERSIONING)	32-bit bitmask
SQL_USER_NAME	(empty string)	char string
SQL_UVNLS_FIELD_MARK	(current field mark value)	char string
SQL_UVNLS_ITEM_MARK	(current item mark value)	char string
SQL_UVNLS_MAP	(current name of map table)	char string
SQL_UVNLS_LC_ALL	(current locale names)	char string
SQL_UVNLS_LC_COLLATE	(current locale name)	char string
SQL_UVNLS_LC_CTYPE	(current locale name)	char string
SQL_UVNLS_LC_MONETARY	(current locale name)	char string
SQL_UVNLS_LC_NUMERIC	(current locale name)	char string
SQL_UVNLS_LC_TIME	(current locale name)	char string

fInfoType Values (Continued)

<i>fInfoType</i>	<i>rgbInfoValue</i>	Type
SQL_UVNLS_SQL_NULL	(current value used to represent the null value)	char string
SQL_UVNLS_SUBVALUE_MARK	(current subvalue mark value)	char string
SQL_UVNLS_TEXT_MARK	(current text mark value)	char string
SQL_UVNLS_VALUE_MARK	(current value mark value)	char string
Supported SQL		
SQL_ALTER_TABLE	SQL_AT_ADD_COLUMN	32-bit bitmask
SQL_COLUMN_ALIAS	“Y”	char string
SQL_CORRELATION_NAME	SQL_CN_DIFFERENT	16-bit integer
SQL_EXPRESSIONS_IN_ORDER_BY	“N”	char string
SQL_GROUP_BY	SQL_GB_GROUP_BY_CONTAINS_SELECT	16-bit integer
SQL_IDENTIFIER_CASE	SQL_IC_SENSITIVE	16-bit integer
SQL_IDENTIFIER_QUOTE_CHAR	"	char string
SQL_KEYWORDS	(empty string)	char string
SQL_LIKE_ESCAPE_CLAUSE	“Y”	char string
SQL_NON_NULLABLE_COLUMNS	SQL_NNC_NON_NULL	16-bit integer
SQL_ODBC_SQL_CONFORMANCE	SQL_OSC_MINIMUM	16-bit integer
SQL_ODBC_SQL_OPT_IEF	“Y”	char string
SQL_ORDER_BY_COLUMNS_IN_SELECT	“N”	char string
SQL_OUTER_JOINS	“Y”	char string
SQL_OWNER_USAGE	0	32-bit bitmask
SQL_POSITIONED_STATEMENTS	0	32-bit bitmask
SQL_PROCEDURES	“Y”	char string
SQL_QUALIFIER_LOCATION	SQL_QL_START	16-bit integer
<i>fInfoType</i> Values (Continued)		

<i>fInfoType</i>	<i>rgbInfoValue</i>	Type
SQL_QUALIFIER_NAME_SEPARATOR	“.”	char string
SQL_QUALIFIER_USAGE	SQL_QU_DML_STATEMENTS	32-bit bitmask
SQL_QUOTED_IDENTIFIER_CASE	SQL_IC_SENSITIVE	32-bit bitmask
SQL_SPECIAL_CHARACTERS	(empty string)	char string
SQL_SUBQUERIES	SQL_SQL_CORRELATED_SUBQUERIES SQL_SQL_COMPARISON SQL_SQL_EXISTS SQL_SQL_IN SQL_SQL_QUANTIFIED	32-bit bitmask
SQL_UNION	0 SQL_U_UNION SQL_U_UNIONALL	32-bit bitmask
SQL Limits		
SQL_MAX_BINARY_LITERAL_LEN	0	32-bit integer
SQL_MAX_CHAR_LITERAL_LEN	0	32-bit integer
SQL_MAX_COLUMNS_IN_GROUP_BY	32	16-bit integer
SQL_MAX_COLUMNS_IN_INDEX	0	16-bit integer
SQL_MAX_COLUMNS_IN_ORDER_BY	32	16-bit integer
SQL_MAX_COLUMNS_IN_SELECT	0	16-bit integer
SQL_MAX_COLUMNS_IN_TABLE	1024	16-bit integer
SQL_MAX_COLUMN_NAME_LEN	18	16-bit integer
SQL_MAX_CURSOR_NAME_LEN	18	16-bit integer
SQL_MAX_INDEX_SIZE	254	32-bit integer
SQL_MAX_OWNER_NAME_LEN	18	16-bit integer
SQL_MAX_PROCEDURE_NAME_LEN	0	16-bit integer
SQL_MAX_QUALIFIER_NAME_LEN	18	16-bit integer
SQL_MAX_ROW_SIZE	0	32-bit integer
<i>fInfoType Values (Continued)</i>		

<i>fInfoType</i>	<i>rgbInfoValue</i>	Type
SQL_MAX_ROW_SIZE_INCLUDES_LONG	“N”	char string
SQL_MAX_STATEMENT_LEN	0	32-bit integer
SQL_MAX_TABLES_IN_SELECT	0	16-bit integer
SQL_MAX_TABLE_NAME_LEN	18	16-bit integer
SQL_MAX_USER_NAME_LEN	18	16-bit integer
Scalar Function Information		
SQL_CONVERT_FUNCTIONS	0	32-bit bitmask
SQL_NUMERIC_FUNCTIONS	0	32-bit bitmask
SQL_STRING_FUNCTIONS	0	32-bit bitmask
SQL_SYSTEM_FUNCTIONS	0	32-bit bitmask
SQL_TIMEDATE_ADD_INTERVALS	0	32-bit bitmask
SQL_TIMEDATE_DIFF_INTERVALS	0	32-bit bitmask
SQL_TIMEDATE_FUNCTIONS	0	32-bit bitmask
Conversion Information		
SQL_CONVERT_BIGINT	0	32-bit bitmask
SQL_CONVERT_BINARY	0	32-bit bitmask
SQL_CONVERT_BIT	0	32-bit bitmask
SQL_CONVERT_CHAR	0	32-bit bitmask
SQL_CONVERT_DATE	0	32-bit bitmask
SQL_CONVERT_DECIMAL	0	32-bit bitmask
SQL_CONVERT_DOUBLE	0	32-bit bitmask
SQL_CONVERT_FLOAT	0	32-bit bitmask
SQL_CONVERT_INTEGER	0	32-bit bitmask
<i>fInfoType</i> Values (Continued)		

<i>fInfoType</i>	<i>rgbInfoValue</i>	Type
SQL_CONVERT_LONGVARBINARY	0	32-bit bitmask
SQL_CONVERT_LONGVARCHAR	0	32-bit bitmask
SQL_CONVERT_NUMERIC	0	32-bit bitmask
SQL_CONVERT_REAL	0	32-bit bitmask
SQL_CONVERT_SMALLINT	0	32-bit bitmask
SQL_CONVERT_TIME	0	32-bit bitmask
SQL_CONVERT_TIMESTAMP	0	32-bit bitmask
SQL_CONVERT_TINYINT	0	32-bit bitmask
SQL_CONVERT_VARBINARY	0	32-bit bitmask
SQL_CONVERT_VARCHAR	0	32-bit bitmask

***fInfoType* Values (Continued)**

Return Values

SQL_SUCCESS
SQL_ERROR
SQL_INVALID_HANDLE
SQL_SUCCESS_WITH_INFO

SQLSTATE Value

The following table describes the SQLGetInfo SQLSTATE value.

SQLSTATE	Description
S1C00	Driver is not capable of handling any <i>fInfoType</i> not supported by this function.

SQLGetInfo SQLSTATE Value

SQLGetStmtTimeOut

SQLGetStmtTimeOut gets the wait time before terminating an attempt to execute a command and generating an error.

Syntax

SQLGetStmtTimeOut (*hstmt*, *stmt_timeout*)

Input Variables

The following table describes the input variable.

Type	Argument	Description
HSTMT	<i>hstmt</i>	Statement handle.

SQLSetParam Input Variables

Output Variables

The following table describes the output variable.

Type	Argument	Description
SWORD *	<i>stmt_timeout</i>	The time, in seconds, to wait for the command to execute.

SQLGetInfo Output Variables

SQLNumParams

SQLNumParams returns the number of parameters in an SQL statement.

Syntax

RETCODE SQLNumParams (*hstmt*, *pcpar*)

Input Variable

The following table describes the input variable.

Type	Argument	Description
HSTMT	<i>hstmt</i>	Statement handle.

SQLNumParams Input Variable

Output Variable

The following table describes the output variable.

Type	Argument	Description
SWORD *	<i>pcpar</i>	Number of parameters in the statement.

SQLNumParams Output Variable

Description

Use this function after preparing or executing an SQL statement or procedure call to find the number of parameters in an SQL statement. If the statement associated with *hstmt* contains no parameters, *pcpar* is set to 0.

A procedure call must be prepared before SQLNumParams can return a result.

Return Values

SQL_SUCCESS
SQL_INVALID_HANDLE
SQL_ERROR

SQLSTATE Values

The following table describes the SQLNumParams SQLSTATE values.

SQLSTATE	Description
S1000	General error for which no specific SQLSTATE code has been defined.
S1001	Memory allocation failure.
S1010	Function sequence error. SQLNumResultCols was called without a prior call to SQLPrepare or SQLExecDirect.

SQLNumParams SQLSTATE Values

SQLNumResultCols

SQLNumResultCols returns the number of columns in a result set.

Syntax

RETCODE SQLNumResultCols (*hstmt*, *pccol*)

Input Variable

The following table describes the input variable.

Type	Argument	Description
HSTMT	<i>hstmt</i>	Statement handle.

SQLNumResultCols Input Variable

Output Variable

The following table describes the output variable.

Type	Argument	Description
SWORD *	<i>pccol</i>	Pointer to the number of columns in the result set returned by <i>hstmt</i> (or 0 if <i>hstmt</i> did not return a result set).

SQLNumResultCols Output Variable

Description

Use this function after preparing or executing an SQL statement or procedure call to find the number of columns in the result set returned. An application can use this function to test whether a submitted SQL statement was a SELECT statement or a procedure call that produced a result set. If the prepared or executed statement was not a SELECT statement and therefore did not return a result set, *pccol* is set to 0. Because the process of preparing a DDL statement also executes it, it is not possible to prepare and test a DDL statement before it is executed.

A procedure call must be executed before SQLNumResultCols can return a result.

You can also use this function when the type of SQL statement is unknown or when the number of columns to be bound to application variables is unknown, for example, when an application is processing SQL statements entered ad hoc by users.

Return Values

SQL_SUCCESS
SQL_ERROR
SQL_INVALID_HANDLE

SQLSTATE Values

The following table describes the SQLNumResultCols SQLSTATE values.

SQLSTATE	Description
S1000	General error for which no specific SQLSTATE code has been defined.
S1001	Memory allocation failure.
S1010	Function sequence error. SQLNumResultCols was called without a prior call to SQLPrepare or SQLExecDirect. In the case of a procedure call, SQLNumResultCols was called without a prior call to SQLExecute or SQLExecDirect.

SQLNumResultCols SQLSTATE Values

SQLParamOptions

SQLParamOptions lets applications specify multiple values for each of the parameters assigned by SQLBindParameter.

Syntax

RETCODE SQLParamOptions (*hstmt*, *crow*, *pirow*)

Input Variables

The following table describes the input variables.

Type	Argument	Description
HSTMT	<i>hstmt</i>	Statement handle.
UDWORD	<i>crow</i>	Number of values for each parameter. If <i>crow</i> > 1, <i>rgbValue</i> in SQLBindParameter and SQLBindMvParameter points to an array of parameter values, and <i>pcbValue</i> points to an array of lengths.
UDWORD FAR *	<i>pirow</i>	Pointer to storage for the current row number. As each row of parameter values is processed, <i>pirow</i> is set to the number of that row. No row number is returned if <i>pirow</i> is empty.

SQLParamOptions Input Variables

Return Values

SQL_SUCCESS
SQL_SUCCESS_WITH_INFO
SQL_ERROR
SQL_INVALID_HANDLE

Description

The ability to specify multiple values for a set of parameters is useful for bulk inserts and other work requiring the data source to process the same SQL statement multiple times with various parameter values. An application can, for example, specify twenty sets of values for the set of parameters associated with an INSERT statement, then execute the INSERT statement once to perform the twenty insertions.

When **SQLExecute** or **SQLExecDirect** is issued after an **SQLParamOptions** call, the SQL statement is executed *crow* times, until another **SQLParamOptions** call is issued with a new *crow* value.

After an **SQLParamOptions** call, **SQLExecute** and **SQLExecDirect** can execute only the following statements:

- INSERT
- UPDATE
- DELETE

When the SQL statement is executed, all variables are checked, data is converted when necessary, and all values in the set are verified to be appropriate and within the bounds of the marker definition. Values are then copied to low-level structures associated with each parameter marker. If a failure occurs while the values are being checked, **SQLExecDirect** or **SQLExecute** returns **SQL_ERROR**, and *value* contains the number of the row where the failure occurred.

SQLParamOptions works only for input parameter types.

You can use **SQLParamOptions** before or after you issue an **SQLBindParameter** or **SQLBindMvParameter** call.

Example

This example shows how you might use **SQLParamOptions** to load a simple table. Table **TAB1** has two columns: an integer column and a **CHAR(30)** column.

```
SDWORD   crow;    /* number of rows SQLParamOption will do */
SDWORD   pirow;   /* storage for SQLParamOptions reply, rows done
*/

SCHAR    szSqlStr2[] = "INSERT INTO TAB1 VALUES (?,?)";
PTR      p1[20];
PTR      p2[20];
```



```

int  pkint[20];

status = SQLAllocEnv(&henv);
status = SQLAllocConnect(henv, &hdbc);
status = SQLSetConnectOption(hdbc, (UWORD)SQL_OS_UID, 0, OsUid);
status = SQLSetConnectOption(hdbc, (UWORD)SQL_OS_PWD, 0, OsPwd);
status = SQLConnect(hdbc, szDSN, strlen(szDSN),
szSchema, strlen(szSchema));
status = SQLAllocStmt(hdbc, &hstmt);

crow = 20;
status = SQLParamOptions(hstmt, crow, &pirow);
status = SQLBindParameter(hstmt, 1, SQL_PARAM_INPUT, SQL_C_SLONG,
SQL_INTEGER, 0, 0, p1, 0, 0);

status = SQLBindParameter(hstmt, 2, SQL_PARAM_INPUT, SQL_C_CHAR,
SQL_CHAR, 0, 0, p2, (SDWORD)23, 0);

status = SQLPrepare(hstmt, szSqlStr2, strlen(szSqlStr2));

for (index = 1; index <= crow; index++)
{
    p1[index - 1] = &(pkint[index - 1]);
    pkint[index - 1] = index;

    p2[index - 1] = itoa(index);
}

status = SQLExecute(hstmt);
printf("%d paramater marker sets were processed\n", pirow);

```

SQLPrepare

SQLPrepare passes an SQL statement or procedure call to the data source to prepare it for execution at the server.

Syntax

RETCODE SQLPrepare (*hstmt*, *szSqlStr*, *cbSqlStr*)

Input Variables

The following table describes the input variables.

Type	Argument	Description
HSTMT	<i>hstmt</i>	Statement handle.
UCHAR *	<i>szSqlStr</i>	Pointer to either an SQL statement or a call to an SQL procedure, to be prepared for execution at the data source.
SDWORD	<i>cbSqlStr</i>	Length of <i>szSqlStr</i> .

SQLPrepare Input Variables

Description

Use this function to deliver an SQL statement or procedure call to the data source where it can be prepared for execution. The application subsequently uses **SQLExecute** to execute the prepared SQL statement or procedure. However, for DDL statements (**CREATE TABLE**, **DROP TABLE**, **GRANT**, **REVOKE**, etc.), you need only prepare the statement. You do not need to explicitly issue **SQLExecute** because SQLPrepare handles the execution.



Use SQLPrepare with SQLExecute when you are issuing SQL statements or calling a procedure repeatedly. For example, if you are inserting or updating multiple rows in a table, you can supply the values for a row to a prepared INSERT or UPDATE statement and issue SQLExecute each time you change the values of the variables bound to parameter markers. SQLExecute sends the current values of the parameter markers to the data source and executes the prepared SQL statement or procedure with the current values.

Note: Before you issue an SQLExecute call, all parameter markers in the SQL statement or procedure call must be defined using the SQLBindParameter call; otherwise SQLExecute returns an error.

You cannot prepare a DDL statement while a transaction is active. You must first either commit or roll back the active transaction; otherwise SQLSTATE S1000 is returned.

Calling SQL Procedures

To call an SQL procedure, use one of the following syntaxes:

```
call procedure [(parameter [, parameter] ...)]  
call procedure [argument [argument] ...]
```

The following table describes each parameter of the syntax.

Parameter	Description
<i>procedure</i>	Name of the procedure. If this name contains characters other than letters or numbers, enclose the name in double quotation marks. To embed a single quotation mark in the procedure name, use two consecutive double quotation marks.
<i>parameter</i>	Either a literal value or a parameter marker that indicates where to insert values to send to or receive from the data source. Programmatic SQL uses a ? (question mark) as a parameter marker. You cannot us SQLBindMvParameter to bind parameter markers used in a call statement. Use parameters only if the procedure is a subroutine. The number and order of parameters must correspond to the number and order of the subroutine arguments.
<i>argument</i>	Any valid keyword, literal, or other token you can use in a database command line.

***call* Parameters**

If SQLBindParameter defines a procedure’s parameter type as SQL_PARAM_OUTPUT or SQL_PARAM_INPUT_OUTPUT, values are returned to the specified program variables.

Return Values

SQL_SUCCESS
SQL_ERROR
SQL_INVALID_HANDLE

SQLSTATE Values

The following table describes the SQLPrepare SQLSTATE values.

SQLSTATE	Description
IM975	You cannot use SQL_PARAM_OUTPUT parameter markers with an SQL statement that is not a called procedure.
S0001	Table or view already exists. Several database error codes can produce this SQLSTATE. The specific reason is returned in the native error code argument of the SQLError call.
S0002	Table or view not found. Several database error codes can produce this SQLSTATE. The specific reason is returned in the native error code argument of the SQLError call.
S0021	Column already exists. Several database error codes can produce this SQLSTATE. The specific reason is returned in the native error code argument of the SQLError call.
S0022	Column not found. Several database error codes can produce this SQLSTATE. The specific reason is returned in the native error code argument of the SQLError call.
S1000	General error for which no specific SQLSTATE code has been defined.
S1001	Memory allocation failure.
21S01	Insert value list does not match the value list.
21S02	Number of columns in derived table does not match the column list.
23000	Integrity constraint violation.
24000	Invalid cursor state. Results are still pending from the previous SQL statement. Use SQLCancel to clear the results.
42000	Syntax error or access violation. This can happen for a variety of reasons. The native error code returned by the SQLError call indicates the specific database error that occurred.

SQLPrepare SQLSTATE Values

SQLRowCount

SQLRowCount returns the number of rows affected by an INSERT, UPDATE, or DELETE statement.

Syntax

RETCODE SQLRowCount (*hstmt*, *pcrow*)

Input Variable

The following table describes the input variable.

Type	Argument	Description
HSTMT	<i>hstmt</i>	Statement handle.

SQLRowCount Input Variable

Output Variable

The following table describes the output variable.

Type	Argument	Description
SDWORD *	<i>pcrow</i>	Pointer to the location into which the row count is stored. If the count cannot be determined, this location is set to 0.

SQLRowCount Output Variable

Description

The value of *pcrow* returned after executing a stored procedure may not be accurate. It is accurate for a single INSERT, UPDATE, or DELETE statement. For a SELECT statement, a 0 row count is always returned, unless the SELECT statement includes the TO SLIST clause. In that case, SQLRowCount returns the number of rows in the select list.

Return Values

SQL_SUCCESS
SQL_ERROR
SQL_INVALID_HANDLE

SQLSTATE Values

The following table describes the SQLRowCount SQLSTATE values.

SQLSTATE	Description
S1000	General error for which no specific SQLSTATE code has been defined.
S1001	Memory allocation failure.
S1010	Function sequence error. SQLRowCount was called before calling SQLExecDirect or SQLExecute for <i>hstmt</i> .

SQLRowCount SQLSTATE Values

SQLSetConnectOption

SQLSetConnectOption lets an application control the way a particular connection operates.

Syntax

RETCODE SQLSetConnectOption (*hdbc*, *fOption*, *vParam*, *szParam*)

Input Variables

The following table describes the input variables.

Type	Argument	Description
HDBC	<i>hdbc</i>	Connection handle.
UWORD	<i>fOption</i>	Option to be set.
UDWORD	<i>vParam</i>	A 32-bit value associated with <i>fOption</i> when <i>fOption</i> is SQL_EMPTY_NULL, SQL_TXN_ISOLATION, or SQL_DATA_MODEL.
UCHAR *	<i>szParam</i>	Text value associated with <i>fOption</i> when <i>fOption</i> is SQL_OS_UID, SQL_OS_PWD, SQL_LIC_DEV_SUBKEY, or one of the SQL_UVNLS options.

SQLSetConnectOption Input Variables

The *vParam* values are as follows:

If <i>fOption</i> is...	<i>vParam</i> is...
SQL_DATA_MODEL	A 32-bit integer value that specifies whether the client is restricted to accessing first normal form (1NF) only: SQL_1NF_MODE_OFF = View base tables as containing all columns (NF² mode). This is the default. SQL_1NF_MODE_ON = View base tables as containing only single-valued columns (1NF mode). You cannot change the data model if a transaction is running.
SQL_EMPTY_NULL	A value that helps control whether UCI interprets empty strings in the database as equivalent to the null value. <i>vParam</i> is one of the following: SQL_EMPTY_NULL_OFF keeps empty strings and null values as distinct values. This is the default. SQL_EMPTY_NULL_ON forces empty strings to be treated as null values in those tables and files whose dictionary contains an @EMPTY.NULL X-descriptor.
<i>vParam</i> Values	

<i>If fOption is...</i>	<i>vParam is...</i>
SQL_TXN_ISOLATION	A 32-bit value that determines the default process isolation level for transactions, effectively setting the locking strategy to be used for executing the SQL statements. <i>vParam</i> must be one of the following: SQL_TXN_READ_UNCOMMITTED (isolation level 1) SQL_TXN_READ_COMMITTED (isolation level 2) SQL_TXN_REPEATABLE_READ (isolation level 3) SQL_TXN_SERIALIZABLE (isolation level 4) Used in two ways: In autocommit mode, <i>vParam</i> determines the isolation level to be used by the server when executing an SQL statement. When manual-commit mode is entered with SQLTransact and the SQL_BEGIN_TRANSACTION option, UCI treats <i>vParam</i> as if you had issued SQL_BEGIN_TRANSACTION plus any value established as the process default isolation level. If the SQL_TXN_ISOLATION option is used inside a transaction, it does not take effect until the current transaction has been committed or rolled back and a new transaction is started.

***vParam* Values (Continued)**

The *szParam* values are as follows:

<i>If fOption is...</i>	<i>szParam is...</i>
SQL_LIC_DEV_SUBKEY	A string of up to 24 characters, used to uniquely identify client devices for database licensing when an application connects to a database server via a multiple-tier connection.
SQL_OS_UID	The operating system user ID to be used when SQLConnect is called to make this <i>hdbc</i> active. It is passed in <i>szParam</i> .
SQL_OS_PWD	The operating system password to be used when SQLConnect is called to make this <i>hdbc</i> active. It is passed in <i>szParam</i> .

***szParam* Values**

<i>If fOption is...</i>	<i>szParam is...</i>
SQL_UVNLS_LC_ALL	A set of values separated by slashes that specifies all components of a locale.
SQL_UVNLS_LC_COLLATE	A value that specifies the name of a locale whose sort order to use.
SQL_UVNLS_LC_CTYPE	A value that specifies the name of a locale whose character type to use.
SQL_UVNLS_LC_MONETARY	A value that specifies the name of a locale whose monetary conventions to use.
SQL_UVNLS_LC_NUMERIC	A value that specifies the name of a locale whose numeric conventions to use.
SQL_UVNLS_LC_TIME	A value that specifies the name of a locale whose time conventions to use.
SQL_UVNLS_LOCALE	A value that specifies all components of a locale.
SQL_UVNLS_MAP	A value that defines the server NLS map for the connection. The server must be able to locate the map table, and the map table must be installed in the server's NLS shared memory segment. <i>szParam</i> is the name of the map table.

***szParam* Values (Continued)**

Description

Once `SQLSetConnectOption` sets an option for a connection, that option remains set until it is specifically reset or the connection is released using an `SQLFreeConnect` statement.

As of Release 9.4.1, if you are connecting to a server running with NLS enabled, you can use the `SQLSetConnectOption` call to specify the NLS map table (`SQL_UVNLS_MAP`) and NLS locale information (`SQL_UVNLS_LOCALE`). You can change these settings after opening the connection, provided a transaction is not active.



Note: Certain combinations of clients and servers may not transfer data predictably because of a mismatch in character mapping, locale settings, or both. See [Connecting to a UniVerse Server with NLS Enabled](#) in Chapter 4, “[Developing UCI Applications](#)” for more information.

Before issuing a call to `SQLConnect`, use `SQLSetConnectOption` calls to specify the user name (`SQL_OS_UID`) and password (`SQL_OS_PWD`) for logging in to a remote database server. On all systems but Windows NT 3.51, if the host specified for this DSN is either *localhost* or the TCP/IP loopback address (127.0.0.1), the user name and password are not required and are ignored if specified. On Windows NT 3.51 systems the user name and password are always required, so you must specify *localpc* as the DSN (for information about adding the *localpc* entry to the UCI configuration file, see [Editing the UCI Configuration File](#) in Chapter 3, “[Configuring UCI](#)”).

If the DSN is not the local host, the client passes the requested user name, password, and schema/account name through to the server. The server verifies the user name/password combination with the operating system and if that is valid, verifies that the requested schema is a valid schema or valid account on the server. Finally, the NLS map and locale settings, if set, are sent to the server. If any of these steps fails, an error is returned, indicating that the server rejected the connection request.

Return Values

`SQL_SUCCESS`
`SQL_ERROR`
`SQL_INVALID_HANDLE`

SQLSTATE Values

The following table describes the `SQLSetConnectOption` SQLSTATE values.

SQLSTATE	Description
S1000	General error for which no specific SQLSTATE code has been defined.

SQLSetConnectOption SQLSTATE Values

SQLSTATE	Description
S1001	Memory allocation failure.
S1009	The value of <i>fOption</i> is not valid for the database.
08002	<i>fOption</i> is SQL_OS_UID, SQL_OS_PWD, or SQL_DATA_MODEL, but <i>hdbc</i> was already connected to a data source.
SQLSetConnectOption SQLSTATE Values (Continued)	

SQLSetStmtTimeOut

SQLSetStmtTimeOut sets the wait time before terminating an attempt to execute a command and generating an error.

Syntax

SQLSetStmtTimeOut (*hstmt*, *stmt_timeout*)

Input Variables

The following table describes the input variables.

Type	Argument	Description
HSTMT	<i>hstmt</i>	Statement handle.
SDWORD	<i>stmt_timeout</i>	The time, in seconds, to wait for the command to execute.

SQLSetParam Input Variables

SQLSetParam

SQLSetParam is provided for compatibility with ODBC 1.0 and the UniVerse BASIC SQL Client Interface. It specifies where values for parameter markers can be found when an SQLExecute or SQLExecDirect call is issued. SQLSetParam is a front end to SQLBindParameter, which is the preferred interface to this functionality.

Syntax

RETCODE SQLSetParam (*hstmt*, *ipar*, *fCType*, *fSqlType*, *cbColDef*, *ibScale*, *rgbValue*, *pcbValue*)

Input Variables

The following table describes the input variables.

Type	Argument	Description
HSTMT	<i>hstmt</i>	Statement handle.
UWORD	<i>ipar</i>	Parameter number, counting sequentially from left to right starting at 1.
SWORD	<i>fCType</i>	The C data type of the parameter. Must be one of the supported C data types as described in Chapter 7, “ Data Types .”
UDWORD	<i>cbColDef</i>	Precision of the column or expression associated with the parameter marker. Not used currently. To ensure appropriate behavior in the future, you must set this variable to SQL_UV_DEFAULT_PARAMETER.
SWORD	<i>ibScale</i>	Scale of the column or expression associated with the parameter marker. Not used currently. To ensure appropriate behavior in the future, you must set this variable to SQL_UV_DEFAULT_PARAMETER.
PTR	<i>rgbValue</i>	Pointer to the buffer containing the parameter’s data.
SDWORD *	<i>pcbValue</i>	Pointer to the buffer containing the parameter’s length.

SQLSetParam Input Variables

Description

This call is mapped to the SQLBindParameter call as follows:

```
RETCODE SQLBindParameter (hstmt, ipar, SQL_PARAM_INPUT_OUTPUT,
fCType, fSqlType, cbColDef, ibScale, rgbValue,
SQL_SET_PARAM_VALUE_MAX, pcbValue)
```

See “SQLBindParameter” on page 27 for further information.

Return Values

```
SQL_SUCCESS
SQL_ERROR
SQL_INVALID_HANDLE
```

SQLSTATE Values

The following table describes the SQLSetParam SQLSTATE values.

SQLSTATE	Description
S1000	General error for which no specific SQLSTATE code has been specified.
S1001	Memory allocation failure.
S1003	The <i>fCType</i> argument is not a recognized data type.
S1093	<i>ipar</i> was less than 1 or greater than the number of parameters in the SQL statement.
07006	The <i>fCType</i> data type cannot be converted to the <i>fSqlType</i> data type.
SQLSetParam SQLSTATE Values	

SQLTables

SQLTables returns a result set listing the tables matching the search patterns.

Syntax

RETCODE SQLTables (*hstmt*, *szTableQualifier*, *cbTableQualifier*, *szTableOwner*, *cbTableOwner*, *szTableName*, *cbTableName*, *szTableType*, *cbTableType*)

Input Variables

The following table describes the input variables.

Type	Argument	Description
HSTMT	<i>hstmt</i>	Statement handle.
UCHAR *	<i>szTableQualifier</i>	Qualifier (schema) name search pattern.
SWORD	<i>cbTableQualifier</i>	Length of <i>szTableQualifier</i> .
UCHAR *	<i>szTableOwner</i>	Table owner number search pattern.
SWORD	<i>cbTableOwner</i>	Length of <i>szTableOwner</i> .
UCHAR *	<i>szTableName</i>	Table name search pattern.
SWORD	<i>cbTableName</i>	Length of <i>szTableName</i> .
UCHAR *	<i>szTableType</i>	Table type search pattern, which can be one of the following: BASE TABLE, VIEW, ASSOCIATION, or TABLE.
SWORD	<i>cbTableType</i>	Length of <i>szTableType</i> .

SQLTables Input Variables

Description

This function returns *hstmt* as a standard result set of five columns containing the qualifiers (schemas), owners, names, types, and remarks for all tables found by the search. The search criteria arguments can contain a literal or an SQL LIKE pattern, or be empty. If a literal or LIKE pattern is specified, only values matching the pattern are returned. If a criterion is empty, tables with any value for that attribute are returned. *szTableOwner* cannot specify a LIKE pattern. You can access the result set with SQLFetch. These five columns have the following descriptors:

	NF ² Mode	1NF Mode
TABLE_SCHEMA	CHAR(18)	CHAR(18)
OWNER	INTEGER	VARCHAR ^a
TABLE_NAME	CHAR(18)	CHAR(18)
TABLE_TYPE	VARCHAR(128)	VARCHAR(128)
REMARKS	CHAR(254)	CHAR(254)

hstmt Result Set

a. In 1NF mode, OWNER is always NULL.



Note: The table owner is the user ID of the person who created the table. SQLTables accepts the table owner search pattern as a character string, but that character string must equate to an integer value and must not contain wildcards.

Special Search Criteria

Three special search criteria combinations enable an application to enumerate the set of schemas, owners, and tables:

Table Qualifer	Table Owner	Table Name	Table Type	Return is...
%	empty string	empty string	<i>ignored</i>	Set of distinct schema names
empty string	%	empty string	<i>ignored</i>	Set of distinct table owners
empty string	empty string	empty string	%	Set of distinct table types

Special Search Criteria

Impact of 1NF Mode

The value returned in *szTableType* is impacted by the setting of the 1NF mode (specified by SQLSetConnectOption):

1NF Mode	Effect
SQL_1NF_MODE_OFF	The default. Virtual 1NF tables are returned from the catalog with their <i>TableType</i> defined as ASSOCIATION to distinguish them from the underlying physical NF ² tables. Refer to Handling Multivalued Columns in Chapter 4, “ Developing UCI Applications ” for an explanation of NF ² mode.
SQL_1NF_MODE_ON	Virtual 1NF tables are returned from the catalog with their <i>TableType</i> defined as TABLE. This enables 1NF users to treat them as discrete tables and to aid interfaces that look for the text TABLE to identify base table objects in the result set of SQLTables. Virtual 1NF tables are distinguishable from base tables because their <i>TableType</i> is TABLE, while the <i>TableType</i> for base tables is BASE TABLE. Refer to Handling Multivalued Columns in Chapter 4, “ Developing UCI Applications ” for an explanation of 1NF mode.

Impact of 1NF Mode

The SQL statement used in both modes when all four search patterns are empty is:

```
SELECT TABLE_SCHEMA, OWNER, TABLE_NAME, TABLE_TYPE, REMARKS
FROM UV_TABLES
ORDER BY 4, 1, 2, 3;
```

In 1NF mode, the TABLE_TYPE column is:

```
EVAL "IF TABLE_TYPE = 'ASSOCIATION' THEN 'TABLE' ELSE TABLE_TYPE"
```

If one or more search patterns are specified, the appropriate SQL WHERE clause is inserted.

The ability to obtain information about tables does not imply that you have any privileges on those tables.

Return Values

- SQL_SUCCESS
- SQL_ERROR
- SQL_INVALID_HANDLE
- SQL_SUCCESS_WITH_INFO

SQLSTATE Values

The following table describes the SQLTables SQLSTATE values.

SQLSTATE	Description
S1000	General error for which no specific SQLSTATE code has been defined.
S1001	Memory allocation failure.
S1008	Cancelled. Execution of the statement was stopped by an SQLCancel call.
S1010	Function sequence error. <i>hstmt</i> is currently executing an SQL statement.
S1C00	The table owner field was not numeric.
24000	Invalid cursor state. Results are still pending from the previous SQL statement. Use SQLCancel to clear the results.
42000	Syntax error or access violation. This can happen for a variety of reasons. The native error code returned by the SQLException call indicates the specific database error that occurred.

SQLTables SQLSTATE Values

SQLTransact

SQLTransact starts a manual-commit mode transaction, or requests a COMMIT or ROLLBACK for all SQL statements associated with a connection or all connections associated with an environment.

Syntax

RETCODE SQLTransact (*henv*, *hdbc*, *fType*)

Input Variables

The following tables describes the input variables.

Type	Argument	Description
HENV	<i>henv</i>	Environment handle.
HDBC	<i>hdbc</i>	Connection handle or SQL_NULL_HDBC.
UWORD	<i>fType</i>	One of the following: SQL_BEGIN_TRANSACTION [+ <i>level</i>], SQL_COMMIT, or SQL_ROLLBACK

SQLTransact Input Variables

Description

This function provides the UCI programmer with the same transaction functions as exist in BASIC with the BEGIN TRANSACTION, COMMIT, and ROLLBACK statements.

You can begin a transaction at a particular transaction isolation level by adding the isolation level to the *fType* parameter. This is equivalent to the BASIC syntax BEGIN TRANSACTION ISOLATION LEVEL *level*. The valid *fType* parameter values are as follows:

fType Value	Description
SQL_BEGIN_TRANSACTION	Puts the database in manual-commit mode. Otherwise, it is in autocommit mode by default, meaning that each SQL statement is executed as a separate transaction. The database supports nested transactions; for example, if SQLTransact is called when another transaction is already active, a nested transaction is begun. If this is the first transaction started, the isolation level used is the default level established with SQLSetConnectOption. If it is not the first transaction, the new transaction uses the same isolation level as the current one used.
SQL_BEGIN_TRANSACTION + SQL_TXN_READ_UNCOMMITTED	Starts a manual-commit mode transaction at isolation level 1.
SQL_BEGIN_TRANSACTION + SQL_TXN_READ_COMMITTED	Starts a manual-commit mode transaction at isolation level 2.
SQL_BEGIN_TRANSACTION + SQL_TXN_REPEATABLE_READ	Starts a manual-commit mode transaction at isolation level 3.

SQLTransact fType Values

fType Value	Description
SQL_BEGIN_TRANSACTION + SQL_TXN_SERIALIZABLE	Starts a manual-commit mode transaction at isolation level 4.
SQL_COMMIT	If the current transaction is not nested: Writes all modified data to the database, releases all locks acquired by the current transaction, and terminates the transaction. If the current transaction is nested: Internally commits any data written during the nested transaction and makes that data visible to the higher-level transaction.
SQL_ROLLBACK	If the current transaction is not nested: Discards any changes written during the transaction and terminates it. If the current transaction is nested: Discards only those changes made by the nested transaction.

SQLTransact fType Values

Setting *henv* to a valid environment handle and *hdbc* to SQL_NULL_HDBC requests the client to try to execute the requested action on all *hdbcs* that are in a connected state.

If any action fails, SQL_ERROR is returned, and the user can determine which connections failed by calling SQLError for each *hdbc* in turn.

If you call SQLTransact with an *fType* of SQL_COMMIT or SQL_ROLLBACK when no transaction is active, SQL_SUCCESS is returned.

Return Values

SQL_SUCCESS
SQL_ERROR
SQL_INVALID_HANDLE

SQLSTATE Values

The following table describes the SQLTransact SQLSTATE values.

SQLSTATE	Description
S1000	General error for which no specific SQLSTATE code has been defined.
S1001	Memory allocation failure.
S1012	<i>fType</i> did not contain SQL_COMMIT, SQL_ROLLBACK, or SQL_BEGIN_TRANSACTION.
08003	No connection is active on <i>hdbc</i> .
08007	The connection associated with the transaction failed during the execution of the function. It cannot be determined if the requested operation completed before the failure.

SQLTransact SQLSTATE Values

SQLUseCfgFile

SQLUseCfgFile lets an application specify which UCI configuration file to use.

Syntax

RETCODE SQLUseCfgFile (*henv*, *option*, *pathname*)

Input Variables

The following table describes the input variables.

Type	Argument	Description
HENV	<i>henv</i>	Environment handle.
UWORD	<i>option</i>	One of the following: SQL_USE_REGISTRY SQL_USE_FILE
UCHAR *	<i>pathname</i>	If <i>option</i> is SQL_USE_REGISTRY, <i>pathname</i> is ignored. If <i>option</i> is SQL_USE_FILE, <i>pathname</i> is the full pathname of the UCI configuration file or an empty string.

SQLUseCfgFile Input Variables

Description

SQLUseCfgFile specifies the full pathname of the UCI configuration file. You can use SQLUseCfgFile to change the default name of the UCI configuration file from *uci.config* to whatever you like. SQLUseCfgFile verifies the existence of the specified configuration file.

If *option* is SQL_USE_REGISTRY, the pathname specified by the following registry entry is used:

```
\HKEY_LOCAL_MACHINE\SOFTWARE\IBM\UCI\UciCfgFile
```

If *option* is SQL_USE_FILE and *pathname* is empty, the UCI configuration file specified by a previous call to SQLUseCfgFile is cleared.

If you do not use SQLUseCfgFile to specify a configuration file, UCI locates the *uci.config* file by searching the following directories in order:

- 1. The current working directory
- 2. The UV account directory
- 3. *On UNIX systems:* the /etc directory
On Windows systems: the directories specified by the PATH variable

Return Values

SQL_SUCCESS

SQL_ERROR

SQLSTATE Values

When SQLUseCfgFile returns SQL_ERROR, you can call SQLError to get the associated SQLSTATE value. Common SQLSTATE values returned are:

SQLSTATE	Description
S1009	<i>option</i> must be either SQL_USE_REGISTRY or SQL_USE_FILE. Only Windows clients can use SQL_USE_REGISTRY.
IA001	Cannot read the registry entry: \\HKEY_LOCAL_MACHINE\\SOFTWARE\\IBM\\UCI\\UciCfgFile
IA002	Cannot access the UCI configuration file specified by SQLUseCfgFile.
S1001	Memory allocation failure.

SQLUseCfgFile SQLSTATE Values

Error Codes

A

This appendix lists the SQLSTATE error codes and the SQL and ODBC error conditions they represent. General ODBC errors produce the default SQLSTATE error code of S1000.

SQLSTATE Error Codes

The following table lists the SQLSTATE values and the corresponding messages they generate.

SQLSTATE	Message
00000	Successful completion
01002	Disconnect failure
01004	Data has been truncated
07001	Not all parameters markers have been resolved
07006	Unsupported data type
08001	Connect failure
08002	Connection already established
08003	Connection is not established
08007	Transaction commit failure
08S01	Communications link failed during operation
21S01	Number of columns inserted doesn't match number expected
21S02	Number of columns selected doesn't match number defined in CREATE VIEW
22001	Character string truncation
22001	Fractional truncation
22003	Numeric value out of range
22005	Nonnumeric data was found where numeric is required
22005	Error in assignment – Data type mismatch (ODBC)
22008	Illegal date/time value
23000	Integrity constraint violation
24000	Invalid cursor state

SQLSTATE Error Codes

SQLSTATE	Message
25000	Connect/disconnect with an active transaction is illegal
34000	An invalid cursor name was specified
3C000	A duplicate cursor name was specified
40000	Transaction rolled back
40001	An SQL statement with NOWAIT encountered a conflicting lock.
42000	User lacks SQL privileges or operating system permissions
IA000	Output from the EXPLAIN keyword.
IM001	Unsupported function
IM002	The data source is not in the configuration file
IM003	An unknown DBMS type has been specified
IM975	Output parameter markers are valid only with procedure calls
IM976	UCI connections to databases other than UniVerse and UniData are not allowed
IM977	Multivalued parameter finding for CALL not allowed
IM978	SQLBindMvCol/SQLBindMvParam illegal on INP connection
IM979	SQLGetData on column bound as multivalued is illegal
IM980	Remote password is required
IM981	Multivalued data present, single result returned
IM982	Remote user ID is required
IM982	Only a single environment variable can be allocated
IM983	Nested transactions to databases other than UniVerse and UniData are not allowed
IM984	The SQL Client Extender is not installed
IM985	Error in RPC interface

SQLSTATE Error Codes (Continued)

SQLSTATE	Message
IM986	Improper SQLTYPE option
IM987	Improper MAPERROR option
IM988	Row exceeds maximum allowable width
IM989	Illegal expiration date format for the SQL Client Extender
IM990	The SQL Client Extender has not been authorized
IM991	License has expired for the SQL Client Extender
IM992	Exceeded licensed number of users for the SQL Client Extender
IM993	Failed opening SequeLink cursor
IM994	A SequeLink middleware error was detected
IM995	An illegal connect parameter was detected
IM996	Fetching into an ODBC environment variable not allowed
IM997	An illegal configuration option was found
IM998	There is no configuration file, or an error was found in the file
IM999	An illegal network type was specified
S0001	Table or view already exists
S0002	Table or view not found
S0021	Column already exists
S0022	Column not found
S1000	An error occurred at the data source
S1001	Memory allocation failure
S1002	An invalid column number specified
S1003	An illegal SQL data type was supplied
S1004	An unsupported SQL data type was encountered
S1009	A 0 or empty pointer was specified

SQLSTATE Error Codes (Continued)

SQLSTATE	Message
S1009	An illegal option value was specified
S1010	Function call is illegal at this point
S1012	Invalid transaction code
S1015	No cursor name was specified
S1090	Invalid parameter length
S1090	Invalid string or buffer length
S1091	An unsupported attribute was specified
S1092	An illegal option value was specified
S1093	An illegal parameter number was specified
S1095	Function type out of range
S1095	Redimensioning arrays containing SQL Client Extender variables, bound columns, or parameter markers
S1096	Information type out of range
S1C00	An invalid data type has been requested
S1C00	Driver does not support this function

SQLSTATE Error Codes (Continued)

UniVerse SQL Error Codes

The following list shows the UniVerse SQL error codes and error message text associated with certain SQLSTATE codes. Some texts are shown in abbreviated form.

Code	Message
S0001	Table or view already exists
950458	UniVerse/SQL: Table “ <i>tablename</i> ” already exists in VOC.
950459	UniVerse/SQL: Table “ <i>tablename</i> ” is being created twice.
950528	UniVerse/SQL: View “ <i>viewname</i> ” already exists in VOC.
950529	UniVerse/SQL: View “ <i>viewname</i> ” is being created twice.
S0002	Table or view not found
950311	UniVerse/SQL: “ <i>viewname</i> ” is a VIEW, not a BASE TABLE.
950313	UniVerse/SQL: “ <i>tablename</i> ” is a BASE TABLE, not a VIEW.
950390	UniVerse/SQL: Table “ <i>tablename</i> ” does not exist.
950455	UniVerse/SQL: View “ <i>viewname</i> ” does not exist.
950545	UniVerse/SQL: “ <i>name</i> ” is not a base table.
950596	UniVerse/SQL: “ <i>associationname</i> ” is an association; not valid for REFERENCES.
950597	UniVerse/SQL: “ <i>associationname</i> ” is an association, not a VIEW.
950598	UniVerse/SQL: “ <i>associationname</i> ” is an association, not a base table or view.
950599	UniVerse/SQL: “ <i>name</i> ” is not a base table; not valid for REFERENCES.

UniVerse SQL Error Codes

Code	Message
S0021	Column already exists
950416	UniVerse/SQL: Explicit column name “ <i>columnname</i> ” is not unique.
950570	UniVerse/SQL: Duplicate column name “ <i>columnname</i> ”.
S0022	Column not found
950418	UniVerse/SQL: Table constraint has an undefined column “ <i>columnname</i> ”.
950425	UniVerse/SQL: Column “ <i>columnname</i> ” not in table.
950428	UniVerse/SQL: Association key column not found.
950522	UniVerse/SQL: Invalid column “ <i>columnname</i> ” specified in constraint.
950523	UniVerse/SQL: Unknown column “ <i>columnname</i> ” specified in table constraint.
21S01	Number of columns INSERTed doesn’t match number expected
950059	UniVerse/SQL: Number of columns inserted doesn’t match number required.
21S02	Number of columns SELECTed doesn’t match number defined in CREATE VIEW
950415	UniVerse/SQL: More explicit column names than columns selected.
950417	UniVerse/SQL: More columns selected than explicit column names.
22005	Error in assignment – Data type mismatch (ODBC)
950043	UniVerse/SQL: <i>type1</i> and <i>type2</i> types are incompatible in this operation.
950121	UniVerse/SQL: Column “ <i>columnname</i> ” data type does not match insert value.

UniVerse SQL Error Codes (Continued)

Code	Message
950122	UniVerse/SQL: Column “ <i>columnname</i> ” data type does not match update value.
950169	UniVerse/SQL: Inconsistent data types in multivalued literal.
950617	UniVerse/SQL: Incorrect data type for literal DEFAULT.
23000	Integrity constraint violation
923012	Integrity Constraint Violation, Index not active
923013	Integrity Constraint Violation, Index not UNIQUE
950136	UniVerse/SQL: <i>constraintname</i> Constraint Violation <i>name</i> on column “ <i>columnname</i> ”.
950568	UniVerse/SQL: Can’t update existing rows with NULL default for NOT NULL column.
950645	UniVerse/SQL: Unable to alter table “ <i>tablename</i> ”, Integrity constraint violation.
40000	Transaction rolled back
040065	FATAL: The locks necessary for database operations at the current isolation level (<i>level</i>) are not held by this process.
909046	Transaction aborted. Roll back attempted.
950604	Fatal error: ISOLATION level cannot be changed during a transaction.
40001	An SQL statement with NOWAIT encountered a conflicting lock
930157	UniVerse/SQL: Locking system failure in CursorOpen
950251	UniVerse/SQL: NOWAIT, Can’t lock record, conflict with another user.
950259	UniVerse/SQL: NOWAIT, Can’t lock file, conflict with another user.
950260	UniVerse/SQL: NOWAIT, Can’t lock record, conflict with user “ <i>user</i> ”.

UniVerse SQL Error Codes (Continued)

Code	Message
950261	UniVerse/SQL: NOWAIT, Can't lock file, conflict with user " <i>user</i> ".
42000	User lacks SQL or operating system permissions
001397	User does not have write privileges to current directory.
001422	Insufficient SQL permissions to read <i>name</i> .
001423	Insufficient SQL permissions to write <i>name</i> .
001424	Insufficient SQL permissions to delete <i>name</i> .
020142	Unable to open " <i>filename</i> " file.
036010	Permission Denied.
950072	UniVerse/SQL: Permission needed to delete records in table " <i>tablename</i> ".
950076	UniVerse/SQL: Permission needed to insert records in table " <i>tablename</i> ".
950078	UniVerse/SQL: Permission needed to update records in table " <i>tablename</i> ".
950131	UniVerse/SQL: Permission needed to update column " <i>columnname</i> " in table " <i>tablename</i> ".
950303	UniVerse/SQL: No read/write permission for <i>username</i> , cannot create schema.
950304	UniVerse/SQL: No <i>rwX</i> permission for <i>name</i> , cannot create schema.
950305	UniVerse/SQL: <i>username</i> does not have <i>rwX</i> permission for <i>name</i> , cannot create schema.
950306	UniVerse/SQL: <i>username</i> does not have <i>rw</i> permission for <i>name</i> , cannot create schema.
950338	UniVerse/SQL: <i>username</i> is not an SQL user.
950343	UniVerse/SQL: <i>username</i> does not have permission to drop schema.
950350	UniVerse/SQL: <i>username</i> does not have permission to create schemas.
950352	UniVerse/SQL: You must be DBA to create a schema for another user.

UniVerse SQL Error Codes (Continued)

Code	Message
950361	UniVerse/SQL: <i>username</i> does not have DBA privilege.
950362	UniVerse/SQL: Command aborted, you may not revoke your own privileges.
950365	UniVerse/SQL: No read/write permission for <i>username</i> , cannot create table.
950391	UniVerse/SQL: You do not have sufficient privileges to REVOKE on this file.
950392	UniVerse/SQL: You do not have sufficient privileges to REVOKE SELECT on this file.
950393	UniVerse/SQL: You do not have sufficient privileges to REVOKE INSERT on this file.
950394	UniVerse/SQL: You do not have sufficient privileges to REVOKE DELETE on this file.
950395	UniVerse/SQL: You do not have sufficient privileges to REVOKE UPDATE on this file.
950398	UniVerse/SQL: Command aborted. <i>username</i> is not an SQL user.
950405	UniVerse/SQL: You do not have sufficient privileges to GRANT on this file.
950406	UniVerse/SQL: You do not have sufficient privileges to GRANT SELECT on this file.
950407	UniVerse/SQL: You do not have sufficient privileges to GRANT INSERT on this file.
950408	UniVerse/SQL: You do not have sufficient privileges to GRANT DELETE on this file.
950409	UniVerse/SQL: You do not have sufficient privileges to GRANT UPDATE on this file.
950534	UniVerse/SQL: Unable to alter table " <i>tablename</i> ".
950538	UniVerse/SQL: You do not have sufficient privileges to REVOKE ALTER on this file.
950539	UniVerse/SQL: You do not have sufficient privileges to REVOKE REFERENCES on this file.

UniVerse SQL Error Codes (Continued)

Code	Message
950540	UniVerse/SQL: You do not have sufficient privileges to GRANT ALTER on this file.
950541	UniVerse/SQL: You do not have sufficient privileges to GRANT REFERENCES on this file.
950546	UniVerse/SQL: Permission needed to alter table <i>tablename</i> .
950548	UniVerse/SQL: Write permission needed to create or delete index.
950563	UniVerse/SQL: You don't have enough privileges to DROP " <i>tablename</i> ".
950588	UniVerse/SQL: Cannot write to <i>tablename</i> .
950590	UniVerse/SQL: Unable to open <i>tablename</i> .
950607	UniVerse/SQL: Unable to create REFERENCES on table <i>tablename</i> .
950609	UniVerse/SQL: Permission needed to create REFERENCES to table <i>tablename</i> .

UniVerse SQL Error Codes (Continued)

UniRPC Error Codes

Remote procedure call (UniRPC) error codes appear if there is a problem in the communications between the client and the server, or if the server encounters one of several error conditions.

Error Code	Meaning
81001	Connection closed, reason unspecified.
81002	On an SQLConnect call, this indicates that the service name specified by the data source was not present on the server, the <i>unirpcservices</i> file was not found, or the service name was not found in the <i>unirpcservices</i> file.
81003	The UniRPC interface has not been initialized.
81004	Error occurred while trying to store an argument in the transmission packet.
81005	The client and server are running incompatible versions of the UniRPC protocol.
81006	A sequence number failure was detected on the connection.
81007	No more connections can be processed by the RPC interface.
81008	A bad UniRPC parameter was detected.
81009	An internal UniRPC error was detected.
81010	A mismatch in the number of arguments passed between the client and server was detected.
81011	Unknown host. The host name or IP address specified in the data source is not valid for the network.
81012	The UniRPC daemon (<i>unirpcd</i>) could not start the <i>uvserver</i> executable.
81015	The connection timed out.
81016	The connection was refused.
930098	The database server could not fork a helper process.

UniRPC Error Codes

The UCI Sample Program

This appendix lists the *ucisample.c* program:

```

/*****
*****
*
*   ucisample.c - the uniVerse UCI example program
*
*   Moduleucisample.cVersion3.1.1.3 Date09/16/96
*
*   (c) Copyright 1993 Ardent Software Inc. - All Rights
Reserved
*   This is unpublished proprietary source code of
Ardent Software Inc.
*   The copyright notice above does not evidence any
actual or intended
*   publication of such source code.
*
*****
*****
*
*   Maintenance log - insert most recent change
descriptions at top
*
*   Date.... GTAR# WHO
Description.....
*   08/23/96 19084 ENF Add destination, uid, pwd to
ucisample
*   07/10/96 18807 AGM Port to Windows
*   08/25/95 17233 ENF Move UCI.h after all other
includes
*   08/17/95 16977 RM   Release memoru from C_ARRAY
structure
*   07/24/95 16977 RM   Finish sample program for 8.3.3
*   06/24/95 15921 RM   New module
*
*****
*****/
#define   __MODULE__   "ucisample.c"
#define   __SCCSID__   "3.1.1.3"

#ifdef   __WIN32

```

```

#include <windows.h>
#include <direct.h>
#include <stdio.h>
#include <conio.h>
#else
#include <unistd.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <termio.h>
#endif

#include "UCI.h"

#ifdef _WIN32
/* begin external function declarations */
SCHAR      *itoa();
/* end external function declarations */
#endif

/* begin static variable declarations */
staticSCHAR szSqlStr1[] = "SELECT * FROM RIDES.F;";
staticSCHAR szSqlStr2[] = "UPDATE RIDES.F SET OPERATOR = ?
                          WHERE RIDE.ID = ?;";
staticSCHAR szSqlStr3[] = "SELECT * FROM RIDES.F WHERE RIDE.ID =
?;";
staticSCHAR szDSN[128];
staticSCHAR szBlank[] = "          ";
/* end static variable declarations */

/*
 * The ERRCHECK macro shows a way to simplify the checking of
errors
 * returned by UCI functions and obtaining error state and message
from
 * SQLError. This macro cannot be used to check SQLAllocEnv
 */

#define  ERRCHECK(fname){ if (ret == SQL_ERROR) {\
    ret = SQLError(henv, hdbc, hstmt, szSqlState, &fNativeError,
    szErrorMsg, sizeof(szErrorMsg)-1,
&cbErrorMsg);\
    if (ret == SQL_SUCCESS || ret == SQL_SUCCESS_WITH_INFO) {\
    printf("\n Died in %s with SQLSTATE %s\n", fname,
szSqlState); \
    printf("\n Native error: %d %s\n", fNativeError,
szErrorMsg); }\
    exit(EXIT_FAILURE); }}

/*
 * The print_carray function is provided to show how to use the
C_ARRAY
 * structure returned by SQLBindMvCol; it mimics uniVerse VERTICAL
listing

```



```

*/

void      print_carray( szLabel, pca )
    SCHAR *szLabel;
    C_ARRAY *pca;
{
    UWORD ui = pca->cDcount;
    UCI_DATUM*udptr = pca->Data;

    while (ui--)
    {
        printf("%-11s.", (udptr == pca->Data) ? szLabel : szBlank);
        if (udptr->fIndicator == SQL_NULL_DATA)
        {
            printf(" <null>\n");
        }
        else if (udptr->fIndicator == SQL_BAD_DATA)
        {
            printf(" <data could not be converted>\n");
        }
        else switch (pca->fCType)
        {
            case SQL_C_CHAR:
            case SQL_C_STRING:
                printf(" %s\n", udptr->uValue.string.text);
                break;
            case SQL_C_DOUBLE:
                printf(" %f\n", udptr->uValue.dbl);
                break;
            case SQL_C_FLOAT:
                printf(" %f\n", (double)udptr->uValue.flt);
                break;
            case SQL_C_TINYINT:
            case SQL_C_STINYINT:
                printf(" %d\n", (int)udptr->uValue.sbyte);
                break;
            case SQL_C_UTINYINT:
                printf(" %d\n", (int)udptr->uValue.ubyte);
                break;
            case SQL_C_SHORT:
            case SQL_C_SSHORT:
                printf(" %d\n", (int)udptr->uValue.sword);
                break;
            case SQL_C_USHORT:
                printf(" %d\n", (int)udptr->uValue.ushort);
                break;
            case SQL_C_LONG:
            case SQL_C_SLONG:
                printf(" %d\n", (int)udptr->uValue.sdword);
                break;
            case SQL_C_ULONG:
                printf(" %d\n", (int)udptr->uValue.udword);
                break;
            case SQL_C_DATE:

```

```

        printf(" %02d-%02d-%04d\n",
            (int)udptr->uValue.date.day,
            (int)udptr->uValue.date.month,
            (int)udptr->uValue.date.year);
        break;
    case SQL_C_TIME:
        printf(" %02d:%02d:%02d\n",
            (int)udptr->uValue.time.hour,
            (int)udptr->uValue.time.minute,
            (int)udptr->uValue.time.second);
        break;
    }
    udptr++;
}

return;
}

/*
 * This routine frees a C_ARRAY structure allocated by
 * SQLBindMvCol
 */

void free_carray( ppca )
    C_ARRAY **ppca;
{
    C_ARRAY *pca;
    UWORD ui;
    UCI_DATUM*udptr;

    if (!ppca || !(pca = *ppca)) return;

    if (pca->fCType == SQL_C_CHAR || pca->fCType == SQL_C_STRING)
    {
        ui = pca->cDcount;
        udptr = pca->Data;
        while (ui--)
        {
            if (udptr->uValue.string.text)
            {
                free(udptr->uValue.string.text);
            }
            udptr++;
        }
    }

    free(pca);
    *ppca = 0;
    return;
}

/*
 * This routine will get the password without echoing it
 */

```

```

char *getpasswd(passwd)
    char *passwd;
{
#ifdef _WIN32
    struct termio tio, tiosave;
    int status;
    char *ptr;

    status = ioctl(0, TCGETA, &tio);
    tiosave = tio;

    tio.c_lflag &= ~ECHO;
    tio.c_lflag &= ~ISIG;
    status = ioctl(0, TCSETA, &tio);

    fgets(passwd, 128, stdin);
    status = ioctl(0, TCSETA, &tiosave);
    ptr = passwd + strlen(passwd) - 1;
    while( *ptr == '\n' || *ptr == '\r') *ptr-- = 0;
    return (passwd);
#else
    char *ptr = passwd;
    int c = 0;

    for(;;)
    {
        c = getch();
        if ( c == '\r') break;
        *ptr++ = (char)c;
    }
    *ptr = 0;
    return( passwd );
#endif
}

main(argc, argv)
    int argc;
    char *argv[];
{
    HENV      henv;                /* the environment handle
*/
    HDBC      hdbc;                /* a connection handle
*/
    HSTMT     hstmt;               /* a statement handle
*/
    RETCODE   ret;                 /* the return code from UCI functions
*/
    SDWORD    i;                   /* local loop counter
*/
    SDWORD    fNativeError;        /* uniVerse error code from SQLError
*/
    SWORD     cbErrorMsg;          /* length of buffer for error text
*/

```

```

        SDWORD  crow;                /* storage for return from SQLRowCount
*/
        SWORD   cbDesc;              /* bytes returned by SQLColAttributes
*/
        SDWORD  fDesc;               /* numeric return from
SQLColAttributes */
        SCHAR   szLabel[5][30];     /* buffers for column headings
*/
        SCHAR   szErrorMsg[512];    /* buffer for uniVerse error message
*/
        SCHAR   szSqlState[9];       /* buffer for SQLSTATE from SQLError
*/
        SCHAR   szSchema[128];       /* path to local uniVerse acc
*/
        SCHAR   OsUid[64];           /* Server User ID
*/
        SCHAR   OsPwd[64];           /* Server password
*/
        C_ARRAY *pCarray[5];         /* holders for data returned by
SQLFetch*/
        C_ARRAY ca1;                 /* used as parameter in SQLBindMvParam
*/
        C_ARRAY ca2;                 /* used as parameter in SQLBindMvParam
*/

        printf("\n\n\tThis is the UCI sample program");
        printf("\n\t~~~~~");
        printf("\n\n\tThis program connects to a UniVerse schema or
account ");
        printf("\n\tusing a user specified data source, and lists the
RIDES.F
        file.");
        printf("\n\tRIDES.F is created by running MAKE.DEMO.FILES in
the
        server\'s");
        printf("\n\tdestination account.");
        printf("\n");

        /*-----
*/
        /* Connect to the uniVerse server
*/
        /*-----
*/

        henv = (HENV) SQL_NULL_HENV;
        hdbc = (HDBC) SQL_NULL_HDBC;
        hstmt = (HSTMT) SQL_NULL_HSTMT;

        /* Get a data source. On UNIX, localuv WILL work. On NT, it
will only
        work at NT 4.0 */

        printf("\n\nEnter the data source to use for the connection:

```

```

");
    szDSN[0] = 0;
    gets(szDSN);
    if ( szDSN[0] == 0)
    {
        printf("\nEmpty data source. Exiting\n");
        exit(EXIT_FAILURE);
    }

    /* Obtain path to the server account */
    printf("Enter destination schema name, account name or
        full path name: ");

    szSchema[0] = 0;
    gets(szSchema);
    if ( szSchema[0] == 0)
    {
        printf("\nEmpty destination. Exiting\n");
        exit(EXIT_FAILURE);
    }

    if (SQL_ERROR == SQLAllocEnv(&henv))
    {
        printf("\nDied in SQLAllocEnv\n");
        exit(EXIT_FAILURE);
    }

    ret = SQLAllocConnect(henv, &hdbc);
    ERRCHECK("SQLAllocConnect");

    /*
        Connect to the UniVerse server(szDSN)
        using the specified destination (szSchema)
    */
    printf("Enter valid server User Name: ");

    OsUid[0] = 0;
    gets(OsUid);
    if ( OsUid[0] == 0)
    {
        printf("\nEmpty user name. Exiting\n");
        exit(EXIT_FAILURE);
    }

    printf("Enter password for user %s: ", OsUid);
    OsPwd[0] = 0;
    getpasswd(OsPwd);
    if ( OsPwd[0] == 0)
    {
        printf("\nEmpty user password. Exiting\n");
        exit(EXIT_FAILURE);
    }

    ret = SQLSetConnectOption(hdbc, (UWORD)SQL_OS_UID, 0, OsUid);

```

```

ERRCHECK("SQLSetConnectOption")

ret = SQLSetConnectOption(hdbc, (UWORD)SQL_OS_PWD, 0, OsPwd);
ERRCHECK("SQLSetConnectOption")

ret = SQLConnect(hdbc, szDSN, strlen(szDSN), szSchema,
                strlen(szSchema));
ERRCHECK("SQLConnect");

ret = SQLAllocStmt(hdbc, &hstmt);
ERRCHECK("SQLAllocStmt");

/*-----
*/
/* First example
*/
/*-----
*/

/* Select the whole file */
ret = SQLExecDirect(hstmt, szSqlStr1, strlen(szSqlStr1));
ERRCHECK("SQLExecDirect");

/*
(1) Bind all columns using SQLBindMvCol even if they are
    single-valued because it's simpler
(2) Obtain the column headings for the report
*/
for (i = 0; i < 5; i++)
{
    ret = SQLBindMvCol(hstmt, i+1, (i == 1) ? SQL_C_STRING :
SQL_C_USHORT,
    &pCarray[i]);
    ERRCHECK("SQLBindMvCol");

    ret = SQLColAttributes(hstmt, i+1, SQL_COLUMN_LABEL,
szLabel[i], 30,
    &cbDesc, &fDesc);
    ERRCHECK("SQLColAttributes");
}

while (1)
{
    ret = SQLFetch(hstmt);
    ERRCHECK("SQLFetch");
    if (ret == SQL_NO_DATA_FOUND)
    {
        break;
    }
    else if (ret == SQL_SUCCESS || ret == SQL_SUCCESS_WITH_INFO)
    {
        printf("\n");
        for (i = 0; i < 5; i++)
        {

```

```

        print_carray( szLabel[i], pCarray[i] );
        free_carray( &pCarray[i] );
    }
}

/*-----
*/
/* Second example
*/
/*-----
*/

printf("\nThe next section will:");
printf("\n(1) begin a transaction");
printf("\n(2) update the operator code of the last record
listed
                        to 999");
printf("\n(3) re-read that record to show the update");
printf("\n(3) roll the transaction back");
printf("\n(4) re-read that record to show original value\n");

/* start a manual-mode transaction */
ret = SQLTransact(henv, hdbc, SQL_BEGIN_TRANSACTION);
ERRCHECK("SQLTransact");

/* bind the parameter for the set clause of the update */
(void) memset((char*)&ca1, 0, sizeof(C_ARRAY));
ca1.cDcount = 1;
ca1.cStorage = 1;
ca1.fCType = SQL_C_USHORT;
ca1.fSqlType = SQL_INTEGER;
ca1.Data[0].uValue.uword = 999;
ret = SQLBindMvParameter(hstmt, 1, &ca1);
ERRCHECK("SQLBindMvParam");

/* bind the primary key for the where clause */
(void) memset((char*)&ca2, 0, sizeof(C_ARRAY));
ca2.cDcount = 1;
ca2.cStorage = 1;
ca2.fCType = SQL_C_USHORT;
ca2.fSqlType = SQL_INTEGER;
ca2.Data[0].uValue.uword = 9;
ret = SQLBindMvParameter(hstmt, 2, &ca2);
ERRCHECK("SQLBindMvParam");

/* update the record */
ret = SQLExecDirect(hstmt, szSqlStr2, strlen(szSqlStr2));
ERRCHECK("SQLExecDirect");

/* verify that the row was updated */
ret = SQLRowCount(hstmt, &crow);
ERRCHECK("SQLRowCount");
if (crow != 1)

```

```

    {
        printf("\nUPDATE statement failed\n");
        exit(EXIT_FAILURE);
    }
    printf("\nUniVerse/SQL: %d record updated.", crow);

    /* the next statement only uses one parameter so clear out old
    ones */
    ret = SQLFreeStmt(hstmt, SQL_RESET_PARAMS);
    ERRCHECK("SQLFreeStmt");

    /* rebind the primary key as the first parameter */
    ret = SQLBindMvParameter(hstmt, 1, &ca2);
    ERRCHECK("SQLBindMvParam");

    /* read the updated record back in */
    ret = SQLExecDirect(hstmt, szSqlStr3, strlen(szSqlStr3));
    ERRCHECK("SQLExecDirect");
    ret = SQLFetch(hstmt);
    ERRCHECK("SQLFetch");
    printf("\nValue of operator code after update:\n");
    print_carray( szLabel[0], pCarray[0] );
    print_carray( szLabel[2], pCarray[2] );
    for (i = 0; i < 5; i++)
    {
        free_carray( &pCarray[i] );
    }

    /* roll the transaction back */
    ret = SQLTransact(henv, hdbc, SQL_ROLLBACK);
    ERRCHECK("SQLTransact");

    /* read the updated record back in */
    ret = SQLExecDirect(hstmt, szSqlStr3, strlen(szSqlStr3));
    ERRCHECK("SQLExecDirect");
    ret = SQLFetch(hstmt);
    ERRCHECK("SQLFetch");
    printf("\nValue of operator code after rolling back:\n");
    print_carray( szLabel[0], pCarray[0] );
    print_carray( szLabel[2], pCarray[2] );
    for (i = 0; i < 5; i++)
    {
        free_carray( &pCarray[i] );
    }

    /*-----
    */
    /* Clean up section
    */
    /*-----
    */

    ret = SQLDisconnect(hdbc);
    ERRCHECK("SQLDisconnect");

```



```
ret = SQLFreeConnect(hdbc);  
ERRCHECK("SQLFreeConnect");  
  
ret = SQLFreeEnv(henv);  
ERRCHECK("SQLFreeEnv");  
  
printf("\n\t*--- End of sample program ---*\n");  
return EXIT_SUCCESS;  
}
```

Glossary

1NF mode	A database mode in which all nonfirst-normal-form (NF ²) tables are treated as first-normal-form (1NF) tables. In 1NF mode, only singlevalued data is available to the application. Associations of multivalued columns are unnested into singlevalued tables.
API	Application programming interface. A set of function calls that provide services to application programs.
application program	A user program that issues function calls to submit SQL statements and retrieve results, and then processes those results.
association	A group of related multivalued columns in a table. The first value in any association column corresponds to the first value of every other column in the association, the second value corresponds to the second value, and so on. An association can be thought of as a nested table. A multivalued column that is not associated with other columns is treated as an association comprising one column.
autocommit mode	A mode of database operation in which each SQL statement is treated as a separate transaction.
binding	The process of associating an attribute with an SQL statement, such as associating parameters or columns with a statement.
CLI	Call level interface. See API .
coercion	The conversion of data returned by the server. Using UCI, an application program can coerce data from a UniVerse table or file into application program variables.
connection handle	A pointer to memory allocated and initialized with the data necessary to describe and maintain a connection between the SQL client application and the data source. Each connection can have multiple statements associated with it.
cursor	A virtual pointer to the set of results produced by a query. A cursor points to the “current row” of the result set, one row of data at a time, and advances one row at a time.

DDL	Data definition language. A subset of SQL statements used for creating, altering, and dropping schemas, tables, views, and indexes. These statements include ALTER TABLE, CREATE SCHEMA, CREATE TABLE, CREATE VIEW, CREATE INDEX, CREATE TRIGGER, DROP SCHEMA, DROP TABLE, DROP VIEW, DROP INDEX, DROP TRIGGER, GRANT, and REVOKE.
DLL	Dynamic link library. A collection of functions linked together into a unit that can be distributed to application developers. When the program runs, the application attaches itself to the DLL when the program calls one of the DLL functions.
DML	Data manipulation language. A subset of SQL statements used for retrieving, inserting, modifying, and deleting data. These statements include SELECT, INSERT, UPDATE, and DELETE.
DSN	Data source name. The name associated with a specific data source entry in the <i>uvodbc.config</i> file.
data source	A source of data, or database engine, represented by the specifications supplied in the data source entry in the <i>uvodbc.config</i> file. These specifications include the DBMS type, network, name of the service, and host platform.
dynamic normalization	On UniVerse systems, a mechanism for allowing first-normal-form data manipulation language (DML) statements to access an association as a virtual first-normal-form table.
embedded SQL	An interface mechanism that includes SQL statements in source code. The SQL statements are precompiled, converting the embedded SQL statements into the language of the host program.
environment handle	A pointer to a data area that contains information concerning the state of the application's data connections, including the valid connection handles.
handle	A pointer to an underlying data structure.
isolation level	A mechanism for separating a transaction from other transactions running concurrently, so that no transaction affects any of the others. There are five isolation levels, numbered 0 through 4.
manual-commit mode	A mode of database operation in which transactions are delimited by a BEGIN TRANSACTION statement and ended by a COMMIT or ROLLBACK statement.
multivalued column	A column that can contain more than one value for each row in a table.

NF ² mode	A database mode in which all nonfirst-normal-form (NF ²) tables are treated as such. This is the standard mode for UniVerse.
NLS	National Language Support.
nested transaction	A transaction that begins while another transaction is active.
null value	A special value representing an unknown value. This is not the same as 0 (zero), a blank, or an empty string.
null-terminated string	A string of characters terminated by a 0 byte.
ODBC	Open Database Connectivity. An interface that defines a library of function calls that permit a client application program to connect to a data source, execute SQL statements against that source, and retrieve results. It also provides a standard set of error codes, a way to connect to the data source, and a standard set of data types. The ODBC specifications from Microsoft for SQL-based database interoperability cover both the application programming interface and SQL grammar. UCI is modelled on this standard but is not ODBC compliant.
parameter marker	A single ? (question mark) in an SQL statement, representing a parameter or argument, where there would normally be a constant. For each iterative execution of the statement, a new value for the parameter marker is made available to the interface.
precision	The maximum number of digits defined for an SQL data type.
prepared SQL statement	An SQL statement that has been processed with the SQLPrepare function. Once prepared, an SQL statement can be executed repeatedly.
programmatic SQL language	A subset of the SQL language. Programmatic SQL differs from interactive SQL in that certain keywords and clauses used for report formatting in interactive mode are not supported in programmatic SQL.
registry	On Windows systems, a systemwide repository of information describing the hardware and software products installed on the system. Specific registry Win32 calls let applications operate on entries in the registry.
result set	A set of rows of data obtained via the SQLFetch call. A result set is returned when an SQL SELECT statement is executed. It is also returned by the SQLColumns and SQLTables calls.
scale	The maximum number of digits allowed to the right of the decimal point.

single-valued column	A column that can contain only one value for each row in a table.
statement handle	A pointer to memory allocated and initialized to hold the context of an SQL statement. A statement handle is always associated with a connection handle.
UCI	Uni Call Interface. A C application programming interface (API) that lets application programmers write client application programs that use SQL function calls to access data in UniVerse databases.
<i>uci.config</i> file	The client UCI configuration file, which defines data sources to which an application can connect in terms of DBMS, network, service, host, and optional extended parameters.
<i>udserver</i> process	A UniData server process that handles requests from the client. For each client connection to the server there is one <i>udserver</i> process.
UniDK	Uni Development Kit. The UniDK comprises UCI, UniObjects, UniObjects for Java, and InterCall.
UniRPC	Remote procedure call. UCI uses a library of calls developed by IBM to implement remote procedure calls. The UniRPC lets a server system execute a function (procedure) provided by a client application program. The client application program passes arguments to the server as well as an identifier specifying the procedure to be executed on the server. The server executes the procedure, using the arguments passed to it, and then returns the results to the client.
<i>unirpc</i> service	On Windows servers, the service that waits for a client's request to connect. When it receives a request, <i>unirpc</i> creates the connection to the server.
<i>unirpcd</i> daemon	On UNIX servers, the daemon that waits for a client's request to connect. When it receives a request, <i>unirpcd</i> creates the connection to the server.
<i>unirpcservices</i> file	The UniRPC services file on the server, used by the UniRPC daemon or service to locate the UniVerse server executable image (<i>uvserver</i>).
UniVerse BASIC SQL Client Interface	Also known as BCI (BASIC Client Interface). A UniVerse BASIC application programming interface that makes UniVerse a client in a client/server environment. Using BCI, UniVerse clients can access both UniVerse and non-UniVerse data sources.
<i>uvserver</i> process	A UniVerse server process that handles requests from the client. For each client connection to the server there is one <i>uvserver</i> process.

Win32 API

The primitive Windows operating system interface for Windows platforms. In this architecture the fundamental size of integers and pointers is 32 bits.

Index

Numerics

INF mode 4-23, 4-24
 definition GI-1
 and dynamic normalization 4-24
 and INSERT statements 4-24
 and SELECT statements 4-24
 tables 4-24

A

administering the UniRPC 2-10
 affected-row count 5-5, 5-6, 6-5, 6-6, 6-7
 allocating memory
 for connection handles 4-5, 8-11
 for environment handles 4-4, 8-13
 for statement handles 4-9
 ALTER statements 4-15
 analyzing result sets 4-16
 API (application programming interface) 1-3
 definition GI-1
 application programming interface, *see* API
 application programs 4-2 to 4-27
 definition GI-1
 initializing 4-4
 sample 4-3
 writing 4-3
 application variables, binding to columns 4-17
 arguments 8-6
 search patterns in 8-7
 associations 4-25
 definition GI-1

dynamically normalized 4-25
 keys 4-25
 autocommit mode 4-10
 definition GI-1
 AUTOINC configuration parameter 3-5

B

BASIC procedures 6-4 to 6-16
 compiling and cataloging 6-4
 restrictions 6-15
 SUBROUTINE statement 6-4
 BASIC programs
 as procedures 6-2
 calling as procedures 5-3
 BASIC subroutines 5-3
 as procedures 6-2
 binding
 application buffers to parameter markers 8-27
 application variable to columns 4-17
 columns 8-17
 definition GI-1
 multivalued columns to array 8-25
 parameter values 4-14
 bound columns, unbinding 4-18

C

C data types
 and coercion from SQL data types 7-10
 and internal/external formats 7-6
 representation of multivalued columns 7-8

- and SQL data types 7-9
- supported 7-3
- unsupported 7-5
- call arguments 8-6
- call interfaces
 - advantages 1-4
 - SQL 1-3
- call level interface (CLI), *see* API
- CALL statement 5-3, 5-5, 6-2
 - nested 6-15
- calling procedures 5-2 to 5-7
- cancelling
 - current query 8-79
 - processing of current SQL statement 8-32
- cascading changes to normalized NF² tables 4-27
- CASE configuration parameter 3-5
- cataloging BASIC procedures 6-4
- CHARACTER data type 7-9
- CHAR_MAX_LENGTH column attribute 8-42
- client interface, *see* UCI
- client system, configuring 3-5
- closing
 - connections 8-56
 - open cursors 8-79
 - @HSTMT variable 6-16
- coercion
 - data types 7-10
 - parameters 7-11
- columns
 - binding 8-17
 - fetching results from 8-69
 - getting descriptions 8-41, 8-52
 - getting detailed attributes of 8-34
 - getting number of columns in result set 8-100
 - unbound, retrieving data from 4-17, 8-82
 - values 8-69
- COLUMN_NAME column attribute 8-42
- commands
 - as procedures 6-2
 - calling as procedures 5-3
- COMMIT statement, requesting for all SQL statements 8-125
- COMO command 6-3

- compiling BASIC procedures 6-4
- configuration
 - client system 3-5
 - client system for NLS server 3-12 to 3-13
 - server system 3-3
 - server system running NLS 3-4
- configuration file, *see* UCI configuration file
- configuration parameters 3-5
 - not relevant to UCI 3-8
 - relevant to UCI 3-5
- configuring UCI 3-2 to 3-13
- connecting to a database 4-5, 8-45
- connection handles 4-5
 - allocating memory for 8-11
 - closing connection associated with 8-56
 - freeing resources associated with 8-73
 - releasing 8-73
- connections
 - closing 8-56
 - controlling 8-111
 - definition Gl-1
 - retrieving errors associated with 8-59
 - controlling how connections operate 8-111
- count, affected row 5-5, 5-6, 6-5, 6-6, 6-7
- CREATE INDEX statements 4-15
- CREATE SCHEMA statements 4-15
- CREATE TABLE statements 4-15
- CREATE VIEW statements 4-15
- cursors 4-16
 - closing open 8-79
 - definition Gl-2
- C_ARRAY data type 7-5
 - structure 7-8

D

- data definition language statements, *see* DDL statements
- data definition statements in procedures 6-6
- data manipulation language statements, *see* DML statements

- data model 4-5
 - see also* 1NF mode, NF² mode
 - setting 8-112
- data source
 - entry 4-5
 - name (DSN) 8-45
- data sources
 - connecting to a data source 8-45
 - definition Gl-2
- DATA statements 6-3, 6-15
- data truncation errors, retrieving truncated data 8-82
- data types 7-2 to 7-14
 - and C data types 7-9
 - and SQL data types 7-9
- databases, connecting to 4-5
- DATA_TYPE column attribute 8-42
- DATE data type 7-9
- date values returned as C structures 7-6
- DATEFETCH configuration parameter 3-8
- DATEFORM configuration parameter 3-8
- DATEPREC configuration parameter 3-8
- DBLPREC configuration parameter 3-8
- DBMSTYPE configuration parameter 3-5
- DDL statements
 - definition Gl-2
 - and dynamic normalization 4-27
 - and nested transactions 4-21
 - processing 4-15
- debugging procedures 6-16
- DECIMAL data type 7-9
- default isolation levels 4-5, 4-22
 - setting 8-113
- DELETE statements 4-15
 - and dynamic normalization 4-27
 - getting number of rows affected by 8-109
 - and nested transactions 4-21
 - in procedures 6-6
- DESCB4EXEC configuration parameter 3-5
- developing UCI applications 4-2 to 4-27
 - requirements 1-8

diagnostics area, *see* SQL diagnostics area
 disconnecting connections 8-56
 display size 7-12, 7-13, 7-14
 DML statements
 definition GI-2
 and dynamic normalization 4-26
 processing 4-15
 documentation conventions 1-ix
 DOUBLE PRECISION data type 7-9
 drivers
 getting general information about 8-89
 getting information about functions supported 8-85
 DROP INDEX statements 4-15
 DROP SCHEMA statements 4-15
 DROP TABLE statements 4-15
 DROP VIEW statements 4-15
 DSN (data source name) 4-6, 8-45
 definition GI-2
 DSPSIZE configuration parameter 3-5
 dynamic normalization 4-26
 and DDL statements 4-27
 definition GI-2
 and DELETE statements 4-27
 and DML statements 4-26
 and INF mode 4-24
 and primary keys 4-25
 and referential integrity 4-27

E

embedded SQL
 definition GI-2
 versus an SQL call interface 1-3
 empty strings 7-7
 as null values 7-7
 and SQLBindMvCol 8-23
 environment, retrieving errors
 associated with 8-59
 environment handles
 allocating memory for 4-4, 8-13
 definition GI-2
 freeing 8-75
 releasing memory associated with 8-75
 environment variables, PATH 2-10

EODCODE configuration
 parameter 3-8
 error
 information, getting 8-58
 return codes 4-18, 8-8, A-1 to ??
 UniRPC A-12
 error codes A-2 to ??
 UniVerse 5-7
 errors 6-7
 checking for 4-17
 messages 5-7
 SQL 6-12
 SQLSTATE 4-18
 executing procedures 5-2 to 5-7
 executing SQL statements 4-11
 directly 4-11
 preparable 8-62
 prepared 4-13, 8-66

F

fetching
 column results 8-18
 rows of data 4-17, 8-69
 specifying where to return results 8-17
 files
 configuration, *see* UCI configuration file
 installation 2-3
 Make.UCI 2-3
 UCI configuration 2-4, 4-5
 ucimsg.text 2-3, 2-4
 ucisample.c 2-3, 2-4
 UCI.a 2-3
 uci.config 3-2 to 3-13, GI-4
 UCI.h 2-3, 2-4, 4-4
 unirpcservices 2-8, 3-3
 @TMP 6-9
 first-normal-form mode, *see* INF mode
 first-normal-form tables 4-24
 FLOAT data type 7-9
 FLOATPREC configuration
 parameter 3-8
 function calls 8-4
 arguments used in 8-6
 functions
 disconnecting 8-5

exchanging data 8-4
 initializing 8-4
 memory management 8-5
 processing errors 8-5
 reference 8-4 to 8-130

G

GRANT statements 4-15

H

halting processing associated with
 statement handle 8-79
 handles 8-5
 connection 4-5
 definition GI-1
 definition GI-2
 environment 4-4
 definition GI-2
 statement 4-9
 definition GI-4
 HDBC argument 8-6
 HENV argument 8-6
 HOST configuration parameter 3-5
 HSTMT argument 8-6
 Hungarian naming conventions 1-xi, 8-8

I

implicit referential integrity 4-27
 initializing application program 4-4
 input in procedures 6-3, 6-15
 INPUT statements 6-15
 input variables 8-6
 INSERT statements 4-15
 getting number of rows affected
 by 8-109
 and nested transactions 4-21
 and INF mode 4-24
 in procedures 6-6
 installing UCI 2-3
 files 2-3
 INTEGER data type 7-9
 internal/external formats and C data
 types 7-6

INTPREC configuration parameter 3-8
isolation levels 4-22
 default 4-22
 definition GI-2
 setting 8-126

K

keys, association 4-25

L

LAN Manager 1-6, 1-8, 2-10, 4-5
language support 1-5
locales 3-6, 3-12, 4-5, 8-115

M

maintaining the UCI configuration file 2-10
Make.UCI file 2-3
manual-commit mode 4-10
 definition GI-3
 and nested transactions 4-20
 starting a manual-commit mode transaction 8-125
map names 3-12
map tables 4-5, 8-115
MAPERROR configuration parameter 3-6
maps 3-7
MARKERNAME configuration parameter 3-6
MAXCHAR configuration parameter 3-8
MAXFETCHBUFF configuration parameter 3-6
 changing 3-10
MAXFETCHCOLS configuration parameter 3-6
 changing 3-10
MAXVARCHAR configuration parameter 3-8
memory
 allocating for connection handles 4-5, 8-11

 allocating for environment handles 4-4, 8-13
 allocating for statement handles 4-9
 menus 6-3
 multicolumn result set 5-5, 5-6, 6-5, 6-6, 6-7, 6-9
 multivalued columns
 binding to array 8-25
 handling 4-23
 normalizing into C arrays 8-22
 representation of
 and C data types 7-8
MULTI_VALUE column attribute 8-42

N

National Language Support, *see* NLS
nested CALL statements in procedures 6-15
nested transactions 4-20
 and DDL statements 4-21
 and DELETE statements 4-21
 and INSERT statements 4-21
 and SELECT statements 4-21
 and UPDATE statements 4-21
nesting levels 4-20
NETWORK configuration parameter 3-6
NF² mode 4-23
 definition GI-3
NLS (National Language Support)
 configuring client 3-12 to 3-13
 configuring server 3-4
 locales 3-6, 3-12, 4-5, 8-115
 map names 3-12
 map tables 3-7, 4-5, 8-115
NLSDEFSRVLC parameter 4-6
NLSDEFSRVMAP parameter 4-6
NLSLCALL configuration parameter 3-6
NLSLCCOLLATE configuration parameter 3-6, 3-12
NLSLCCTYPE configuration parameter 3-6, 3-12
NLSLCLOCALE configuration parameter 3-7, 3-12
NLSLCMODE parameter 4-6

NLSLCMONETARY configuration parameter 3-6, 3-12
NLSLCNUMERIC configuration parameter 3-6, 3-12
NLSLCTIME configuration parameter 3-7, 3-12
NLSMAP configuration parameter 3-7, 3-12
NLSMODE parameter 4-6
nonfirst-normal-form mode, *see* NF² mode
nonfirst-normal-form tables 4-23
null values 7-7
NULLABLE
 column attribute 8-42
 configuration parameter 3-7
null-terminated string, definition GI-3
NUMERIC data type 7-9
NUMERIC_PRECISION column attribute 8-42
NUMERIC_PREC_RADIX column attribute 8-42
NUMERIC_SCALE column attribute 8-42

O

ODBC
 2.0 standard 1-7
 definition GI-3
 error conditions A-1 to ??
output parameters 5-6, 6-4
output variables 8-6
overview of UCI 1-2 to 1-8

P

paragraphs
 as procedures 6-2
 calling as procedures 5-3
parameter markers 4-14, 5-3, 6-4
 binding application buffer to 8-27
 definition GI-3
 releasing variables 8-79
 values 5-2, 8-28, 8-62, 8-66, 8-117
 binding 4-14, 4-15
parameters
 coercion 7-11

output 5-6, 6-4
 unbinding 4-18
 PATH environment variable 2-10
 precision 7-12, 7-13, 7-14
 definition G1-3
 PRECISION configuration
 parameter 3-8
 prefixes in function syntax 8-8
 preparable SQL statements,
 executing 8-62
 prepared SQL statements
 definition G1-3
 executing 8-66
 preparing SQL statements for
 execution 4-13, 8-105
 print result set 5-5, 5-7, 6-3, 6-5, 6-16
 PRINT statements 6-16
 procedures 5-2 to 5-7, 8-62, 8-105
 BASIC 6-4 to 6-16
 and data definition statements 6-6
 debugging 6-16
 DELETE statements 6-6
 INPUT statements 6-6
 nested CALL statements in 6-15
 processing results 5-5
 restrictions in BASIC 6-15
 and UniVerse commands 6-3, 6-15
 and UniVerse menus 6-3
 UPDATE statements 6-6
 and user input 6-3, 6-15
 writing 6-2 to 6-16
 processing SQL statements 4-11
 procs
 as procedures 6-2
 calling as procedures 5-3
 programmatic SQL language
 case-sensitivity 8-10
 definition G1-3
 programs
 as procedures 6-2
 calling as procedures 5-3
 ProVerb procs
 as procedures 6-2
 calling as procedures 5-3
 PTR argument 8-6

R

read committed isolation level 4-22
 read uncommitted isolation level 4-22
 REAL data type 7-9
 REALPREC configuration
 parameter 3-8
 reference page layout for functions 8-3
 referential integrity
 and dynamic normalization 4-27
 implicit 4-27
 and 1NF mode 4-27
 releasing statement handles 4-18
 REMARKS column attribute 8-42, 8-121
 remote procedure call, *see* UniRPC
 repeatable read isolation level 4-22
 restrictions in BASIC procedures 6-15
 result sets 5-5
 analyzing 4-16
 definition G1-4
 getting next row of data from 8-69
 getting number of columns in 8-100
 multicolumn 5-5, 5-6, 6-5, 6-6, 6-7, 6-9
 print 5-5, 5-7, 6-3, 6-5, 6-16
 results
 processing procedure 5-5
 RETCODE argument 8-6
 return codes, *see* error return codes
 return values 8-8
 see also error return codes
 REVOKE statements 4-15
 ROLLBACK statement, requesting for
 all SQL statements 8-125
 row count
 affected 5-5, 5-6, 6-5, 6-6, 6-7
 rows
 fetching 4-17
 fetching next row from result set 8-69
 getting number of rows affected by
 DELETE, INSERT, and
 UPDATE 8-109
 RPC, *see* UniRPC
 running UCI applications,
 requirements 1-8

S

sample application B-1 to B-11
 scale 7-12, 7-13, 7-14
 definition G1-4
 SCALE configuration parameter 3-8
 SDWORD argument 8-6
 SEARCH configuration parameter 3-7
 search pattern arguments in function
 calls 8-7
 SELECT statements
 and nested transactions 4-21
 and 1NF mode 4-24
 serializable isolation level 4-22
 server system
 configuring 3-3
 setup 2-8
 status, getting 8-58
 SERVICE configuration parameter 3-7
 SetDiagnostics function 6-12, 6-13, 6-14, 6-16
 SMINTPREC configuration
 parameter 3-8
 SPOOL command 6-3
 SQL
 embedded, definition G1-2
 SQL call interface
 advantages 1-4
 definition 1-3
 UCI as an 1-3
 versus embedded SQL 1-3
 SQL data types
 and C application data types 7-9
 and coercion to C data types 7-10
 supported 7-9
 display size 7-12, 7-13, 7-14
 precision 7-12, 7-13, 7-14
 scale 7-12, 7-13, 7-14
 and UniVerse SQL data types 7-9
 SQL diagnostics area 6-13
 SQL errors 6-12
 SQL result sets, *see* result sets
 SQL statements
 cancelling processing of current 8-32
 case-sensitivity 8-10
 executing 4-11
 preparable 8-62
 prepared 8-66

preparing for 8-105	SQLSTATE return codes 4-18, 5-7, 8-60	SQL_COLUMN_TYPE_NAME column attribute 8-38
processing 4-11	SQLSTATE values A-2	SQL_COLUMN_UNSIGNED column attribute 8-38
SQLAllocConnect function 4-4, 4-6, 8-11	SQLTables function 8-120	SQL_COLUMN_UPDATABLE column attribute 8-38
SQLAllocEnv function 4-4, 8-13	SQLTransact function 4-10, 4-20, 4-22, 8-125	SQL_C_CHAR data type 7-3
SQLAllocStmt function 4-4, 4-9	SQLTYPE configuration parameter 3-8	SQL_C_DATE data type 7-5, 7-9
SQLBindCol function 4-17, 5-5, 6-15, 8-17	SQLUseCfgFile function 8-129	SQL_C_DOUBLE data type 7-4, 7-9
SQLBindMvCol function 4-17, 4-24, 8-22	SQL_INF_MODE_OFF option 8-112 and SQLTables function 8-122	SQL_C_FLOAT data type 7-4, 7-9
SQLBindMvParameter function 4-14, 4-15, 4-24, 8-25	SQL_INF_MODE_ON option 8-112 and SQLTables function 8-122	SQL_C_LONG data type 7-4
SQLBindParameter function 4-14, 5-6, 8-27	SQL_CHAR data type 7-9	SQL_C_SHORT data type 7-3
SQLCancel function 8-32	SQL_CLOSE option 8-79	SQL_C_SLONG data type 7-4, 7-9
SQLClearDiagnostics function 6-13	SQL_COLUMN_AUTO_INCREMENT T column attribute 8-36	SQL_C_SSHORT data type 7-4
SQLColAttributes function 4-16, 5-5, 6-15, 8-34	SQL_COLUMN_CASE_SENSITIVE column attribute 8-36	SQL_C_STINYINT data type 7-3
SQLColumns function 8-41	SQL_COLUMN_CONVERSION column attribute 8-36	SQL_C_STRING data type 7-4, 7-9
SQLConnect function 4-4, 4-5, 8-45	SQL_COLUMN_COUNT column attribute 8-36	SQL_C_TIME data type 7-4, 7-9
SQLDataSources function 8-49	SQL_COLUMN_DISPLAY_SIZE column attribute 8-36	SQL_C_TINYINT data type 7-3
SQLDescribeCol function 4-16, 6-15, 8-52	SQL_COLUMN_FORMAT column attribute 8-36	SQL_C_ULONG data type 7-4
SQLDisconnect function 4-19, 8-56	SQL_COLUMN_LABEL column attribute 8-36	SQL_C_USHORT data type 7-4
SQLError function 5-7, 8-8, 8-58	SQL_COLUMN_LENGTH column attribute 8-37	SQL_C_UTINYINT data type 7-3
SQLExecDirect function 4-13, 4-15, 5-3, 5-7, 6-12, 8-62	SQL_COLUMN_MULTI_VALUED column attribute 8-37	SQL_DATA_MODEL option 8-112
SQLExecute function 4-13, 4-15, 5-3, 5-5, 5-7, 6-12, 8-66	SQL_COLUMN_NAME column attribute 8-37	SQL_DATE data type 7-9
SQLFetch function 4-17, 5-5, 6-15, 8-69	SQL_COLUMN_NULLABLE column attribute 8-37	SQL_DECIMAL data type 7-9
SQLFreeConnect function 8-73	SQL_COLUMN_PRECISION column attribute 8-37	SQL_DOUBLE data type 7-9
SQLFreeEnv function 8-75	SQL_COLUMN_PRINT_RESULT column attribute 8-37	SQL_DROP option 8-79
SQLFreeMem function 8-23, 8-77	SQL_COLUMN_SCALE column attribute 8-38	SQL_EMPTY_NULL option 7-7, 8-112
SQLFreeStmt function 4-9, 4-15, 4-18, 8-78	SQL_COLUMN_SEARCHABLE column attribute 8-38	SQL_ERROR return value 8-8
SQLGetData function 4-17	SQL_COLUMN_TABLE_NAME column attribute 8-38	SQL_FLOAT data type 7-9
SQLGetFunctions function 8-85	SQL_COLUMN_TYPE column attribute 8-38	SQL_INTEGER data type 7-9
SQLGetInfo function 8-89		SQL_INVALID_HANDLE return value 8-8
SQLNumResultCols function 4-16, 5-5, 5-6, 6-15, 8-100		SQL_LIC_DEV_SUBKEY option 8-114
SQLParamOptions function 8-102		SQL_NO_DATA_FOUND return value 8-8
SQLPrepare function 4-13, 5-5, 8-105		SQL_NUMERIC data type 7-9
SQLRowCount function 4-16, 5-5, 5-6, 8-109		SQL_OS_PWD option 8-114
SQLSetConnectOption function 4-5, 4-22, 4-23, 7-7, 8-111		SQL_OS_UID option 8-114
SQLSetParam function 4-15, 8-117		SQL_PARAM_INPUT option 8-25, 8-27
		SQL_PARAM_INPUT_OUTPUT option 8-27
		SQL_PARAM_OUTPUT option 8-27
		SQL_REAL data type 7-9
		SQL_RESET_PARAMS option 8-79
		SQL_SMALLINT data type 7-9
		SQL_SUCCESS return value 8-8

SQL_SUCCESS_WITH_INFO return value 8-8

SQL_TIME data type 7-9

SQL_TXN_ISOLATION option 8-113

SQL_UNBIND option 8-79

SQL_UVNLS_LC_ALL option 8-114

SQL_UVNLS_LC_COLLATE option 8-114

SQL_UVNLS_LC_CTYPE option 8-114

SQL_UVNLS_LC_MONETARY option 8-114

SQL_UVNLS_LC_NUMERIC option 8-114

SQL_UVNLS_LC_TIME option 8-114

SQL_UVNLS_LOCALE option 8-114

SQL_UVNLS_MAP option 8-114

SQL_VARCHAR data type 7-9

SSPPORTNUMBER configuration parameter 3-8

statement handles

- definition G1-4
- halting all processing associated with 8-79
- releasing 8-78
- releasing resources associated with 8-78

statements

- retrieving associated errors 8-59
- terminating 4-18

status variables 8-8

stored sentences

- as procedures 6-2
- calling as procedures 5-3

string math 7-6

SUBROUTINE statement 6-4

subroutines 5-3

- as procedures 6-2

subtransactions

- see also* nested transactions
- isolation levels 4-22

SWORD argument 8-6

syntax

- prefixes in 8-8
- tags in 8-8

system administration

- administering the UniRPC 2-10

maintaining the UCI configuration file 2-10

T

tables

- first normal form 4-24
- getting description of tables found by search pattern 8-120
- nonfirst-normal-form 4-23

TABLE_NAME table attribute 8-42, 8-121

TABLE_OWNER table attribute 8-42, 8-121

TABLE_SCHEMA table attribute 8-42, 8-121

TABLE_TYPE table attribute 8-121

tags in function syntax 8-8

TCP/IP network 1-6

terminating statements 4-18

TIME data type 7-9

time values returned as C structures 7-6

transaction isolation levels 4-22

- default 4-22

transaction modes 4-10

- autocommit 4-10
- manual-commit 4-10

transactions 4-20 to 4-22

- nested 4-20
- nesting levels 4-20

TXBEHAVIOR configuration parameter 3-7

TXCOMMIT configuration parameter 3-7

TXROLL configuration parameter 3-7

TXSTART configuration parameter 3-7

TYPENAME configuration parameter 3-7

TYPE_NAME column attribute 8-42

U

UCHAR argument 8-6

UCI

- compliance with ODBC 2.0 standard 1-7

configuration parameters not relevant to 3-8

configuration parameters relevant to 3-5

configuring 3-2 to 3-13

configuring client for NLS server 3-12 to 3-13

configuring NLS server 3-4

data types 7-3 to 7-14

developing applications 4-2 to 4-27

function call reference 8-4 to 8-130

language support 1-5

overview 1-2 to 1-8

requirements for developing applications 1-8

requirements for running applications 1-8

as an SQL call interface 1-3

SQL data types supported 7-9

UNIX installation 2-3

- files 2-3

UCI configuration file 2-4, 4-5

- editing 3-8
- maintaining on client 2-10

ucimsg.text file 2-3, 2-4

ucisample.c file 2-3, 2-4, 4-3

- source code B-1 to B-11

UCI.a file 2-3

uci.config file 3-2 to 3-13

- changing parameters in 3-10
- definition G1-4

UCI.h file 2-3, 2-4, 4-4

UCI_DATUM data type 7-5

UDWORD argument 8-6

unbinding

- bound columns 4-18
- parameters 4-18

unbound columns, retrieving data from 8-82

UniData data types, *see* data types

UniData internal/external formats, *see* internal/external formats

UniData release, getting information about 8-89

UniDK (Uni Development Kit) 2-4

- definition G1-4

UniRPC

- administering 2-10
- daemon 2-10, 3-3

definition G1-4
 error codes A-12
 library 1-6
 service 2-8, 3-3
 services file, *see* *unirpcservices* file
unirpc service 2-8, 3-3
unirpcd process
 see also UniRPC: daemon
 definition G1-4
unirpcservices file 2-8, 2-11, 3-3
 see also UniRPC: services file
 definition G1-5
 UniVerse commands
 as procedures 6-2
 calling as procedures 5-3
 in procedures 6-3, 6-15
 UniVerse data types, *see* data types
 UniVerse error codes 5-7
 UniVerse internal/external formats, *see*
 internal/external formats
 UniVerse menus 6-3
 UniVerse procedures, writing 6-2 to 6-
 16
 UniVerse release, getting information
 about 8-89
 UniVerse server 3-3
 UNSIGNED configuration
 parameter 3-7
 UPDATE
 configuration parameter 3-7
 statements 4-15
 getting number of rows affected
 by 8-109
 and nested transactions 4-21
 UPDATE statements
 in procedures 6-6
 user input in procedures 6-3, 6-15
 USETGITX configuration
 parameter 3-8
uvserver process 3-3
 see also UniVerse server
 definition G1-4, G1-5
uvsvrhelpd process 2-10
 UWORD argument 8-6

V

values

column 8-69
 empty string 7-7
 null 7-7
 parameter marker 5-2, 8-28, 8-62, 8-
 66, 8-117
 return 8-8
 SQLSTATE 8-60
 VARCHARACTER data type 7-9
 variables 8-5
 binding to columns 4-17
 in function calls 8-5
 input 8-6
 name prefixes 8-8
 name tags 8-8
 names 8-5
 output 8-6
 releasing variables bound to
 columns 8-79
 status 8-8
 @HSTMT 6-6, 6-7, 6-13, 6-15, 6-16

W

Windows NT 3.51 4-5, 8-46, 8-115
 writing UniVerse procedures 6-2 to 6-
 16

X

X-descriptors, @EMPTY.NULL 7-7

Symbols

% search pattern 8-7
 ? parameter marker 4-14
 see also parameter markers
 @ASSOC_ROW keyword 4-25
 @EMPTY.NULL X-descriptor 7-7, 8-
 112
 @HSTMT variable 6-6, 6-7, 6-13, 6-
 15
 closing 6-16
 @TMP file 6-9
 \ search pattern 8-7
 _ search pattern 8-7