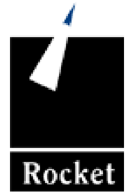


UniVerse

BASIC SQL Client Interface Guide

UNV-103-BCI-1

Notices

Edition

Publication date: February 2009

Book number: UNV-103-BCI-1

Product version: UniVerse 10.3

Copyright

© Rocket Software, Inc. 1985-2009. All Rights Reserved.

Trademarks

The following trademarks appear in this publication:

Trademark	Trademark Owner
Rocket Software™	Rocket Software, Inc.
Dynamic Connect®	Rocket Software, Inc.
RedBack®	Rocket Software, Inc.
SystemBuilder™	Rocket Software, Inc.
UniData®	Rocket Software, Inc.
UniVerse™	Rocket Software, Inc.
U2™	Rocket Software, Inc.
U2.NET™	Rocket Software, Inc.
U2 Web Development Environment™	Rocket Software, Inc.
wIntegrate®	Rocket Software, Inc.
Microsoft® .NET	Microsoft Corporation
Microsoft® Office Excel®, Outlook®, Word	Microsoft Corporation
Windows®	Microsoft Corporation
Windows® 7	Microsoft Corporation
Windows Vista®	Microsoft Corporation
Java™ and all Java-based trademarks and logos	Sun Microsystems, Inc.
UNIX®	X/Open Company Limited

The above trademarks are property of the specified companies in the United States, other countries, or both. All other products or services mentioned in this document may be covered by the trademarks, service marks, or product names as designated by the companies who own or market them.

License agreement

This software and the associated documentation are proprietary and confidential to Rocket Software, Inc., are furnished under license, and may be used and copied only in accordance with the terms of such license and with the inclusion of the copyright notice. This software and any copies thereof may not be provided or otherwise made available to any other person. No title to or ownership of the software and associated documentation is hereby transferred. Any unauthorized use or reproduction of this software or documentation may be subject to civil or criminal liability. The information in the software and documentation is subject to change and should not be construed as a commitment by Rocket Software, Inc.

Restricted rights notice for license to the U.S. Government: Use, reproduction, or disclosure is subject to restrictions as stated in the “Rights in Technical Data-General” clause (alternate III), in FAR section 52.222-14. All title and ownership in this computer software remain with Rocket Software, Inc.

Note

This product may contain encryption technology. Many countries prohibit or restrict the use, import, or export of encryption technologies, and current use, import, and export regulations should be followed when exporting this product.

Please be aware: Any images or indications reflecting ownership or branding of the product(s) documented herein may or may not reflect the current legal ownership of the intellectual property rights associated with such product(s). All right and title to the product(s) documented herein belong solely to Rocket Software, Inc. and its subsidiaries, notwithstanding any notices (including screen captures) or any other indications to the contrary.

Contact information

Rocket Software
275 Grove Street Suite 3-410
Newton, MA 02466-2272
USA
Tel: (617) 614-4321 Fax: (617) 630-7100
Web Site: www.rocketsoftware.com

Table of Contents

Preface

Organization of This Guide ii
Documentation Conventions. iii
UniVerse Documentation. v
Related Documentation vii
API Documentation viii

Chapter 1 Introduction

UniVerse Data Sources 1-2
ODBC Data Sources 1-2
Additional BASIC Functions 1-4
The CONNECT Command 1-5
System Requirements 1-6
Administering the RPC on UniVerse Servers 1-7
ODBC Dynamic Link Libraries. 1-8

Chapter 2 Getting Started

Configuring the BASIC SQL Client Interface 2-4
Changing the Size of the Server’s Result-Set Buffer. 2-5
Location of the Configuration File 2-5
Format of the Configuration File. 2-6
Client Configuration for NLS-Enabled UniVerse Servers 2-6
Creating and Modifying Data Source Definitions 2-7
Using the UniVerse System Administration Menus 2-11
Maintaining the Configuration File 2-12
Using the SQL Client Interface 2-19
Running the Demonstration Program 2-20
Create a Schema 2-20
Run the Program. 2-21

Chapter 3 Using the CONNECT Command

Command Syntax 3-3
Command Options 3-3
Logging On to the Data Source 3-9
Logging On to a Local UniVerse Server 3-10
Logging On to a Remote UniVerse Server 3-10
Logging On to an ODBC Data Source 3-11
Executing SQL Statements on the Data Source 3-13
Using Block Mode 3-13
Using Local Commands 3-15
Displaying and Storing Output 3-17
Examples 3-18
Using Verbose Mode 3-19
Changing the Display Width of Columns 3-20
Exiting CONNECT 3-21
Using UniVerse Output Mode 3-21
Using Block Mode 3-23

Chapter 4 Using the SQL Client Interface

Establishing a Connection to a Data Source 4-3
Connecting to NLS-Enabled Data Sources 4-3
Allocating the Environment 4-5
Allocating the Connection Environment 4-5
Connecting to a Data Source 4-5
Processing SQL Statements 4-8
Allocating the SQL Statement Environment 4-8
Executing SQL Statements. 4-8
Processing Output from SQL Statements 4-11
Freeing the SQL Statement Environment 4-13
Terminating the Connection 4-14
Transaction Management 4-18
Distributed Transactions 4-18
Nested Transactions 4-19
Detecting Errors. 4-21
UniVerse Error and System Messages 4-22
Displaying Environment Variables in RAID 4-24

Chapter 5 Calling and Executing Procedures

What Can You Call as a UniVerse Procedure? 5-3
Processing UniVerse Procedure Results 5-5

Print Result Set 5-5
Affected-Row Count 5-6
Processing Errors from UniVerse Procedures 5-7
Calling and Executing ODBC Procedures 5-8

Chapter 6 How to Write a UniVerse Procedure

Using UniVerse Paragraphs, Commands, and Procs as Procedures . . . 6-3
Writing UniVerse BASIC Procedures 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 6-14
Fetching Rows and Closing @HSTMT Within a Procedure 6-15
Hints for Debugging a Procedure 6-15

Chapter 7 SQL Client Interface Functions

Variable Names 7-5
Return Values 7-6
Error Codes 7-7
ClearDiagnostics 7-8
GetDiagnostics 7-9
SetDiagnostics. 7-10
SQLAllocConnect 7-12
SQLAllocEnv **7-14**
SQLAllocStmt. 7-16
SQLBindCol **7-18**
SQLBindParameter **7-21**
SQLCancel **7-25**
SQLColAttributes. **7-27**
SQLColumns **7-33**
SQLConnect **7-36**
SQLDescribeCol **7-39**
SQLDisconnect **7-41**
SQLError **7-43**
SQLExecDirect **7-47**
SQLExecute 7-51
SQLFetch **7-54**
SQLFreeConnect 7-56
SQLFreeEnv **7-57**

SQLFreeStmt 7-58
SQLGetInfo **7-60**
SQLGetTypeInfo 7-66
SQLNumParams 7-70
SQLNumResultCols **7-72**
SQLParamOptions **7-74**
SQLPrepare **7-78**
SQLRowCount **7-81**
SQLSetConnectOption **7-83**
SQLSetParam **7-89**
SQLSpecialColumns 7-90
SQLStatistics **7-95**
SQLTables **7-100**
SQLTransact **7-103**

Appendix A Data Conversion

Converting BASIC Data to SQL Data A-6
BASIC to SQL Character Types A-7
BASIC to SQL Binary Types A-8
BASIC to SQL.DECIMAL and SQL.NUMERIC. A-9
BASIC to SQL Integer Types A-9
BASIC to SQL.REAL, SQL.FLOAT, and SQL.DOUBLE A-10
BASIC to SQL.DATE A-10
BASIC to SQL.TIME A-11
BASIC to SQL.TIMESTAMP. A-12
Converting SQL Data to BASIC Data A-13
Converting SQL Character Types to BASIC Data Types A-14
Converting SQL Binary Types to BASIC Data Types A-15
Converting SQL Numeric Types to BASIC Data Types. A-16
Converting SQL Date, Time, and Timestamp Types to BASIC Types . A-17

Appendix B SQL Client Interface Demonstration Program

Main Program B-2
Creating Local UniVerse Files B-5
Inserting Data into Local UniVerse Tables B-7
Creating Tables on the Data Source B-8
Inserting Data into the Data Source Table. B-9
Selecting Data from the Data Source B-11
Checking for Errors B-12

Appendix C **Error Codes**

Appendix D **UniVerse Extended Parameters**

Appendix E

Appendix F **The ODBC.H File**

Preface

This guide shows how to use the UniVerse BASIC SQL Client Interface (BCI). It is for application developers who are familiar with UniVerse BASIC and want to connect to an SQL server DBMS. The server database can be a local or remote UniVerse system, or it can be an ODBC database such as DB2, INFORMIX, or SYBASE. The SQL Client Interface lets you access, create, delete, and modify server databases on your local system or on one or more remote systems.

Much of the information in this book originally appeared in *uV/SQL Client Option Guide*.

Organization of This Guide

This guide contains the following:

Chapter 1, "Introduction," introduces the UniVerse BASIC SQL Client Interface.

Chapter 2, "Getting Started," describes how to configure your system to use the SQL Client Interface.

Chapter 3, "Using the CONNECT Command," describes how to use the CONNECT command.

Chapter 4, "Using the SQL Client Interface," tells how to use the SQL Client Interface to communicate with servers.

Chapter 5, "Calling and Executing Procedures," describes how to call and execute procedures stored on a UniVerse or ODBC data source.

Chapter 6, "How to Write a UniVerse Procedure" describes how to write a UniVerse procedure.

Chapter 7, "SQL Client Interface Functions," describes the SQL Client Interface functions.

Appendix A, "Data Conversion," describes how the SQL Client Interface converts data between the client system and various server DBMSs.

Appendix B, "SQL Client Interface Demonstration Program," contains a demonstration program that uses the SQL Client Interface.

Appendix C, "Error Codes," describes error codes and messages issued by the SQL Client Interface.

Appendix D, "UniVerse Extended Parameters," lists data source and DBMS-type extended parameters.

Appendix E, "The ODBC.H File," lists the contents of the ODBC.H file, which defines the values of column attributes.

The "Glossary" defines terms used in this guide.

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 UniVerse commands, keywords, and options; BASIC statements and functions; and SQL statements and keywords. In text, uppercase also indicates UniVerse identifiers such as filenames, account names, schema names, and Windows filenames and paths.
Italic	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 that 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.
?	A right arrow between menu options indicates you should choose each option in sequence. For example, "Choose **File** ?**Exit**" means you should choose **File** from the menu bar, then choose **Exit** from the File pull-down menu.

Documentation Conventions

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.

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 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 UniVerse 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, INFORMIX, SYBASE, or DB2. 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 UniAdmin 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 manage servers and databases, and monitor UniVerse performance and locks.

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 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 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 install, configure, and use UniVerse ODBC drivers on client systems. This book is for experienced UniVerse developers who are familiar with SQL and ODBC.

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.

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 for UniData and UniVerse, 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 UniVerse and UniData 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.

Chapter

Introduction

1

UniVerse Data Sources 1-2
ODBC Data Sources 1-2
Additional BASIC Functions 1-4
The CONNECT Command 1-5
System Requirements 1-6
Administering the RPC on UniVerse Servers 1-7
ODBC Dynamic Link Libraries 1-8

The UniVerse BASIC SQL Client Interface is an application programming interface (API) that makes UniVerse a client in a client/server environment. The server data source can be either:

- A local or remote UniVerse database
- A relational DBMS such as INFORMIX, SYBASE, or DB2

You use the SQL Client Interface to connect to one or more data sources.

UniVerse Data Sources

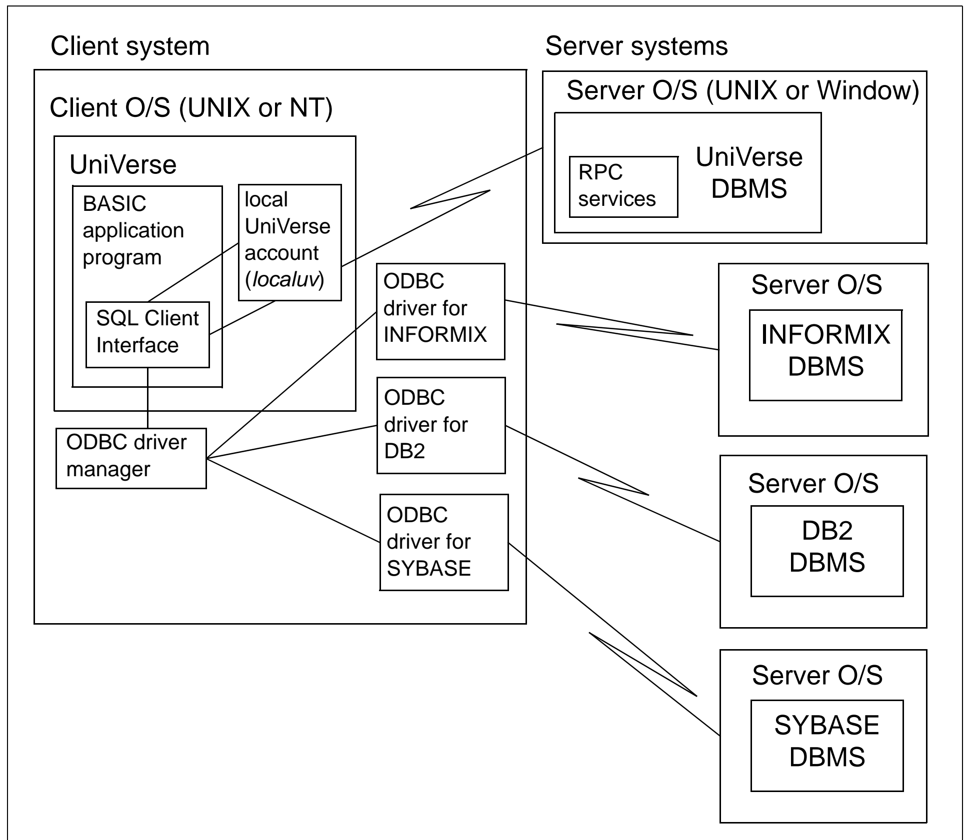
The SQL Client Interface connects directly to local UniVerse servers. Remote UniVerse servers use the remote procedure call services (RPC) to communicate with the SQL Client Interface. To connect to a UniVerse data source, the RPC daemon (service) must be running on the UniVerse server.

As of Release 9.4.1, UniVerse servers can run with NLS (National Language Support) enabled. NLS provides extensive support for many singlebyte and multibyte character sets and country-specific locale settings. NLS lets client application programs store and retrieve data in a UniVerse data source using the character sets and locale settings most appropriate for the client application and operating system. NLS is fully documented in the *UniVerse NLS Guide*.

ODBC Data Sources

To connect to an ODBC data source, an ODBC driver manager and suitable ODBC drivers for the data sources you want to connect to must be installed on the client system.

Once connected to any data source, the SQL Client Interface lets you read data from and write data to the data source. Your application program can access the capabilities of the server DBMS, as shown in the following example.



UniVerse SQL Client / Server Relationships

The SQL Client Interface also includes the CONNECT command, which lets users access data sources interactively. For detailed information about the CONNECT command, see Chapter 3, "Using the CONNECT Command."

The SQL Client Interface is based on the core-level definition of the Microsoft Open Database Connectivity (ODBC) interface. The ODBC interface lets you write programs that can operate across a wide range of data sources. With the SQL Client Interface, application developers have access to the full range of capabilities offered by the server DBMS. For complete information about the ODBC interface, see *Microsoft ODBC 2.0 Programmer's Reference and SDK Guide*.

Additional BASIC Functions

UniVerse BASIC includes a set of functions that make up the SQL Client Interface. A client application program uses these functions to do the following:

- Allocate resources for connections
- Connect to one or more local or remote data sources
- Send SQL statements to the data source for execution
- Call procedures stored on the data source for execution
- Receive results row by row from SELECT statements
- Insert, update, and delete rows using SQL data manipulation statements
- Create and drop tables and views using SQL data definition statements
- Receive status and error information from the data source
- Disconnect from the data source

The I/O operations in the SQL Client Interface differ from normal BASIC I/O. The SQL Client Interface does not use the BASIC file operations OPEN, READ, WRITE, and CLOSE.

The CONNECT Command

The SQL Client Interface provides a utility, invoked with the CONNECT command, that lets you connect to a server DBMS and interactively manipulate and display data from that system on the client system.

System Requirements

To use the SQL Client Interface to access the local UniVerse database, you need Release 9.6 or later of UniVerse.

To use the SQL Client Interface to access a remote UniVerse database, you need the following:

- TCP/IP hardware and software installed on both client and server systems
- Release 9.6 or later of UniVerse on the client system
- Release 9.6 or later of UniVerse on the server system
- The UniRPC daemon (*unirpcd*) running on a UNIX server
- The UniRPC service (*unirpc*) running on a Windows server

To use the SQL Client Interface to access an ODBC database, you need the following:

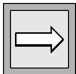
- TCP/IP hardware and software installed on both client and server systems
- At least one DBMS installed on a server system
- ODBC driver manager and ODBC driver for the data source, installed on the client system
- Release 9.6 or later of UniVerse installed on the client system

Administering the RPC on UniVerse Servers

On UniVerse servers, the RPC handles requests from client systems. On UNIX systems the RPC daemon *unirpcd* receives **SQLConnect** requests and starts up the appropriate server processes to support each application using the SQL Client Interface. Each application has two supporting processes, *uvserver* and *uvsrvhelpd*, on the server while it is connected. The server daemon (*uvsrvd*) is the process to which a client connects.

On Windows platforms, the RPC service *unirpc* receives connection requests and starts the server processes. A helper thread runs as part of the *uvserver* process on Windows platforms.

Before any client can connect to the UniVerse server, the system administrator of the server must ensure that the RPC daemon or service is running and that the *unirpcservices* file, which defines the RPC services available on the server, contains an entry (*uvserver*) for the *uvsrvd* daemon. Once the RPC daemon or service is started, it automatically restarts whenever you reboot UniVerse.



***Note**: The RPC must be configured and running even if the client and server systems are on the same machine, unless you make a direct connection to the local UniVerse server using the host name localhost or the loopback IP address 127.0.0.1.*

See *Administering UniVerse* for information about executing the following tasks:

- Add and remove nodes from the network
- Change the number of the RPC port
- Start and stop the RPC daemon or service

Administering UniVerse also describes the structure and function of the *unirpcservices* file in the UV account directory. The *unirpcservices* file contains an entry for *uvserver*, which is required to run applications using the SQL Client Interface.

ODBC Dynamic Link Libraries

On UNIX systems, the ODBC dynamic link library (DLL) is installed in the *uvdlls* directory in the UV account directory (*$uvhome/uvdlls*). This library has the name *libodbc*.xx, where *xx* is supplied by the system you are running on. The installation program creates a symbolic link from */.uvlibs* to the *uvdlls* directory. Both the *uvsh* and *uvsrvd* modules look for their shared libraries in this directory, so it is necessary that this symbolic link not be broken.

If for any reason the symbolic link is broken, or if the system administrator moves the shared libraries to another directory, the *relink.uvlibs* shell script must be used to relink them. The *relink.uvlibs* script resides in *$uvhome/bin*. To relink the shared libraries, use the following syntax:

relink.uvlibs *pathname*

pathname is the full path of the directory containing the shared libraries. For example:

% **relink.uvlibs $uvhome/uvdlls**

Before using the SQL Client Interface to connect to an ODBC data source, the administrator must reestablish the link to *uvdlls* in order to use the ODBC driver manager. To do this, complete the following steps:

1. Install the ODBC driver manager according to the vendor's instructions.
2. Determine where the ODBC DDL *libodbc*.xx resides. For example, the library for the Intersolv driver resides in *$odbchome/dlls*, and the library for the Visigenics driver resides in *$odbchome/libs*.
3. Shut down UniVerse.
4. Execute the *relink.uvlibs* script to relink to the ODBC DDL. For example, to relink to the Intersolv driver library, enter the following:

   % **relink.uv.libs $odbchome/dlls**
5. Restart UniVerse.

The library directory containing the ODBC driver's DDL is stored in an environment variable, which may not be the same name on all systems. For example, the environment variable is called LD_LIBRARY_PATH on Solaris systems, SHLIB_PATH on HP systems, and so on. If this environment variable is not properly set, running SQL Client Interface programs may produce errors such as the following:

ld.so.1: uvsh: fatal: lib*xxxx*: can't open file: errno=2

xxxx may be some unrecognizable combination of letters and numbers. To correct this, set up your environment according to the vendor's instructions.

Chapter

Getting Started

2

Configuring the BASIC SQL Client Interface 2-3
Changing the Size of the Server's Result-Set Buffer 2-4
Location of the Configuration File 2-4
Format of the Configuration File 2-5
Client Configuration for NLS-Enabled UniVerse Servers 2-5
Creating and Modifying Data Source Definitions 2-6
Using the UniVerse System Administration Menus. 2-10
Maintaining the Configuration File. 2-11
Using the SQL Client Interface 2-18
Running the Demonstration Program 2-19
Create a Schema 2-19
Run the Program 2-20

This chapter describes how to do the following:

- Define data sources in the configuration file
- Maintain the configuration file
- Run the SQL Client Interface demonstration program

Configuring the BASIC SQL Client Interface

The SQL Client Interface needs information about data sources to which it can connect. A data source is a combination of hardware and software to which a client application connects and with which it exchanges data.

The SQL Client Interface defines a data source by means of a *data source specification*. A data source specification contains all information needed to let an application connect to and interact with a data source. Data source specifications are stored in the configuration file *uvodbc.config*.

When you install UniVerse, the configuration file contains one specification (*localuv*) for the local UniVerse server. The *uvodbc.config* file should contain at least one data source specification for each UniVerse data source to which you want to connect.

ODBC data sources need not be specified in *uvodbc.config*. However, if you want to use the CONNECT command to connect to an ODBC data source, you must specify the data source in *uvodbc.config*.

You can specify several different data source specifications (with different names) for the same data source. Each specification must include the following:

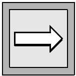
- Data source name, which the SQL Client Interface uses to reference the data source. You use this name with the CONNECT command or the BASIC **SQLConnect** call to specify the data source you are connecting to.
- DBMS type, which defines the type of data source (UNIVERSE or ODBC). The DBMS type determines what internal parameters the SQL Client Interface needs in order to exchange data with the data source.

UniVerse data source specifications also require the following three elements, which are not required for ODBC data sources:

- Network connection type. On UNIX systems the only supported network type is TCP/IP. On Windows platforms, the network type can be TCP/IP or LAN.
- Service name as defined on the server. For local and remote UniVerse services, the service name is *uvserver.*

- Network host name or IP address of the machine running the UniVerse data source to which you want to connect. For a direct connection to the local UniVerse server, the host name is either *localhost* or the loopback IP address 127.0.0.1.

In addition to the required items, a data source specification can include extended parameters that control data precision, transaction management, NLS locale information, and so on. Extended parameters tailor the operation of a particular data source or of all data sources.



***Note**: The ODBC data source name in the uvodbc.config file must match the data source name specified in the initialization file (odbc.ini) or registry of the ODBC data source.*

Changing the Size of the Server's Result-Set Buffer

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 improves 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.

Location of the Configuration File

The configuration file is called *uvodbc.config*. It is normally in the UV account directory but can also be in your current working directory. On UNIX systems, the configuration file can also be in */etc*; on Windows platforms, it can be in one of the directories specified in the PATH environment variable. The SQL Client Interface searches for the configuration file in the following order:

1. Current working directory
2. UV account directory

3. **On UNIX systems:** the */etc* directory

   On Windows platforms: each directory specified in the PATH environment variable

Format of the Configuration File

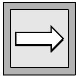
The following example illustrates the format of the configuration file:

```
<localuv>
DBMSTYPE = UNIVERSE
NETWORK = TCP/IP
SERVICE = uvserver
HOST = localhost

<uv8dg>
DBMSTYPE = UNIVERSE
NETWORK = TCP/IP
SERVICE = uvserver
HOST = dg8500

<ora>
DBMSTYPE = ODBC

<syb>
DBMSTYPE = ODBC
```



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

Appendix D, "UniVerse Extended Parameters," lists the extended parameters and their default values. It also tells which parameters you can modify.

Client Configuration for NLS-Enabled UniVerse Servers

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

If you need to specify particular server locale information, change the *uvodbc.config* file to contain this information either for each data source or for all UniVerse server connections. NLS users should note that the *uvodbc.config* file entries are in ASCII format. When you specify NLS and locale parameters in the *uvodbc.config* file, you do not need to make changes to your programs to let client programs work with an NLS-enabled server.

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 locales. These locales specify the five locale categories in the order listed above.

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.ALL table with ID = NL-DUTCH, except for the LCMONETARY entry, which is loaded from the NLS.LC.MONETARY table for the DEFAULT entry.

If more than one entry is found 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 one of the most important locale parameter because it affects the order in which rows are returned to the application.

Creating and Modifying Data Source Definitions

Use UniAdmin or an editor such as Sysedit, *vi*, or *emacs*, to edit the configuration file. Using UniVerse Admin you can create, modify, and delete UniVerse and ODBC data sources.

On Windows platforms, to add or change extended parameters for a data source or for all data sources of a particular type, you must edit the file manually.

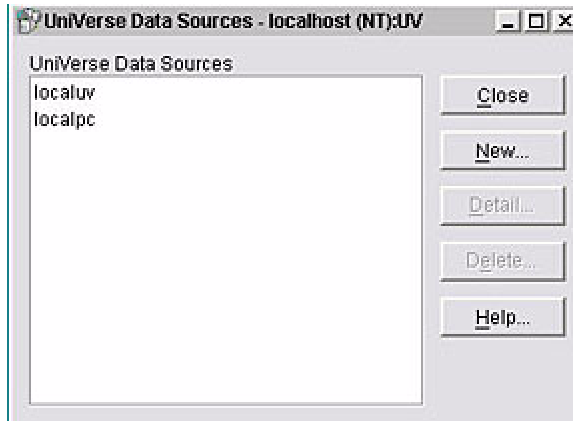
On UNIX systems you can use the UniVerse System Administration menus to edit the configuration file. You must be a UniVerse Administrator to use the System Administration menus.

Using UniAdmin

Creating the Configuration File

Data source administration lets you view and change the current set of defined BASIC SQL Client Interface (BCI) data sources.

To administer UniVerse data sources, choose **Data Sources** from the UniAdmin main window. The **UniVerse Data Sources** window appears, as shown in the following example:



This window lists data sources defined in the *uvodbc.config* file.

You can perform the following data source administration operations:

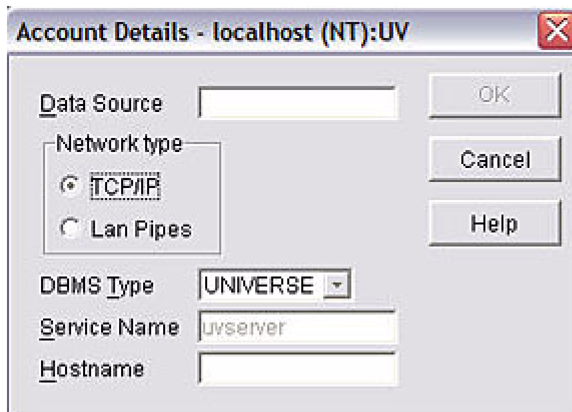
If you want to...	Do this...	To...
Create a new data source	Click **New…**	Display the **Account Details** dialog box.
Delete an existing data source	Select the data source, then click **Delete…**	Prompt you for confirmation, and delete the data source if you click **Yes**.
View information about a data source	Select the data source, then click **Detail…**	Display the **Account Details** dialog box.
Dismiss the Data Source Admin window	**Close**	Return to the UniAdmin main window.
Get information about data source administration	**Help**	Display the online help.

Data Source Administration Operations

Creating a New Data Source

Complete the following steps to create a new data source:

1. Click **New**. The **Account Details** dialog box appears, as shown in the following example:



2. Enter the following data source information:

In this field...	Enter...
Data Source	The name you want to assign to the new data source.
Network Type	Click TCP/IP or LAN Pipes to specify the communication transport to use to communicate with the data source. You can use LAN pipes only with Windows servers.
DBMS Type	Name of the database management system on the server. Currently, the only choices are UNIVERSE and ODBC.
Service name	UniVerse server name. Currently you cannot change this field.
Hostname	Name or IP address of the server on which the data source resides.

3. Click **OK** to add the new data source or click **Cancel** to return to the **UniVerse Data Sources** window without making any changes.

Deleting a Data Source

To delete an existing data source:

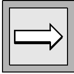
1. Choose the data source you want to delete.
2. Click **Delete**. The system prompts you to confirm the deletion.
3. Click **Yes** to delete the data source, or click **Cancel** to return to the **UniVerse Data Sources** window without making any changes.

Viewing or Modifying a Data Source

To view or modify an existing data source:

1. Choose the data source you want to view or change.
2. Click **Detail**. The **Account Details** dialog box appears.
3. View or change the data source information.
4. Click **OK** to implement any changes to the data source, or click **Cancel** to return to the **Account Details** dialog box without making any changes.

Using the UniVerse System Administration Menus



***Note**: The UniVerse System Administration menus are available only on UNIX systems.*

To display the **Sql Client Administration** menu, choose **Package** from the **System Administration** main menu, then choose **Sql client administration**. The submenu shown in the following example appears.

```
                         UniVerse System Administration
<Package  > Accounts   Recovery   Spooler   sh Mem    Import    Devices
+------------------------------------+
| Rpc administration             =>  |
| License administration             |
| Install package                    |
| De-install package                 |
|<Sql client administration      =>> |+----------------------------------+
| dEadlock daemon administration =>  || data Source specifications       |
| Gci administration             =>  || data source eXtended parameters  |
+------------------------------------+| Dbms type extended parameters    |
                                      +----------------------------------+
```

The first time you choose an option from the **Sql client administration** menu, UniVerse prompts you to enter the full path of the configuration file. The default is *uvodbc.config* in the current directory.

Maintaining the Configuration File

The **Sql client administration** menu has three options:

Option	Description
data Source specifications	Defines new data sources and change information about existing data sources.
data source eXtended parameters	Defines additional parameters for a particular UniVerse data source.
Dbms type extended parameters	Defines additional parameters for all data sources of a particular type (UNIVERSE or ODBC).

SQL client administration Options

When you choose an option, a data entry screen appears. The menu bar at the top of the screen provides three pull-down menus. Press **F10** or **Ctrl-T** to move the cursor to the menu bar.

The File and Help menus have the same options on all **Sql client administration** screens.

The **File** menu has three options:

Option	Description
Save	Saves the currently displayed values to the configuration file.
Refresh	Discards the currently displayed values and retrieves the most recently saved data.
Exit	Returns to the previous pull-down menu. If the data source name field is not empty, the system asks whether you want to save your changes to the configuration file.

File Menu Options

The **Help** menu provides help about the Application, the Keys, and the current Version number. Action menu options are explained in the following sections.

Defining and Changing Data Source Specifications

When you choose the **data Source specifications** option, the following screen appears.

```
+------------------ Maintain Data Source Specifications ------------------+
|            File                  Action                Help             |
+-------------------------------------------------------------------------+
|                                                                         |
|                                                                         |
|   Data Source Name :                                                    |
|   DBMS Type        :                                                    |
|   Network Type     :                                                    |
|   Service Name     :                                                    |
|   Host Name/Address:                                                    |
|                                                                         |
+------------------------------ Help Region ------------------------------+
|    Enter full path of Configuration file (default is current directory) |
|    +-------------------------------------------------------------+      |
|    |?                                                            |      |
|    +-------------------------------------------------------------+      |
|                                                                         |
+-------------------------------------------------------------------------+
```

At the **Data Source Name** prompt, enter the name of a data source. When you enter the name of a new data source, you can define the data source by entering values in each field. When you enter the name of an existing data source, the details of that source are displayed and you can change the values in any field. Press **F4** or enter ***** (asterisk) to list all currently defined data sources. You can then enter the name of a data source by choosing it.

In the "DBMS Type" field, press **F4** or enter ***** (asterisk) to list the valid DBMS types. The SQL Client Interface supports these DBMS types:

- UNIVERSE
- ODBC

The **Network Type** field is relevant only for UniVerse data sources. It can be TCP/IP or LAN. Use LAN only for two or more Windows platforms connected to each other.

The **Service Name** field is relevant only for UniVerse data sources. The service name of the UniVerse server is *uvserver.*

The **Host Name/Address** field is relevant only for UniVerse data sources. If you specify a network host name for a UniVerse data source, the host name must be in the hosts file (or equivalent file if you use another network administration mechanism such as Yellow Pages) on the client system. You can specify the host name or an IP address for the local UniVerse server as well as for remote data sources. For direct connections to the current account on the local UniVerse server, use the host name *localhost* or the TCP/IP loopback address 127.0.0.1.

If you specify an IP address, use the following format:

```
198.232.158.155
```

When you press **ENTER** at the end of the **Host Name/Address** field, UniVerse prompts to change data in any field.

If you choose **Yes**, use the cursor keys to move to the fields you want to change, enter the new values, then move to the **Host Name/Address** field and press **ENTER**.

When you choose **No**, UniVerse prompts to save your data. Choose **Yes** to save your data in the configuration file.

The Action menu has three options:

Option	Description
Delete	Deletes the displayed data source specification.
Rename	Prompts you to enter a data source name and changes the current data source name to the name you enter.
Copy	Prompts you to enter a data source name and copies the currently displayed data to the new data source specification. Extended data source parameters are not copied.

Action Menu Options

Adding or Changing a Data Source's Extended Parameters

When you choose the **data source eXtended parameters** option, the following screen appears.

```
Maintain Data Sources Extended Parameters
File    Action    Help

Mode         : Add

Data Source ?
Parameter    :

Help Region
Press <F10> to activate the Menu Bar, <ESC> to Abort, Up/Down Arrow to go
up/down a field, and <F1> for longer help about each prompt.
Using  <./uvodbc.config>  as the SQL Client configuration file.
Enter a data source name or <F4> for a list of all currently configured data
sources.
```

At the **Data Source** prompt, enter the name of a data source. Press **F4** or enter ***** (asterisk) to list data sources currently defined in the configuration file.

At the **Parameter** prompt, enter **?** (question mark) to list possible parameters. Press **F4** or enter ***** (asterisk) to list all extended parameters currently defined for this data source. You can add, change, or delete an entry using options on the **Action** menu.

Parameters affecting data type precision or error mapping are the only ones likely to be of interest.

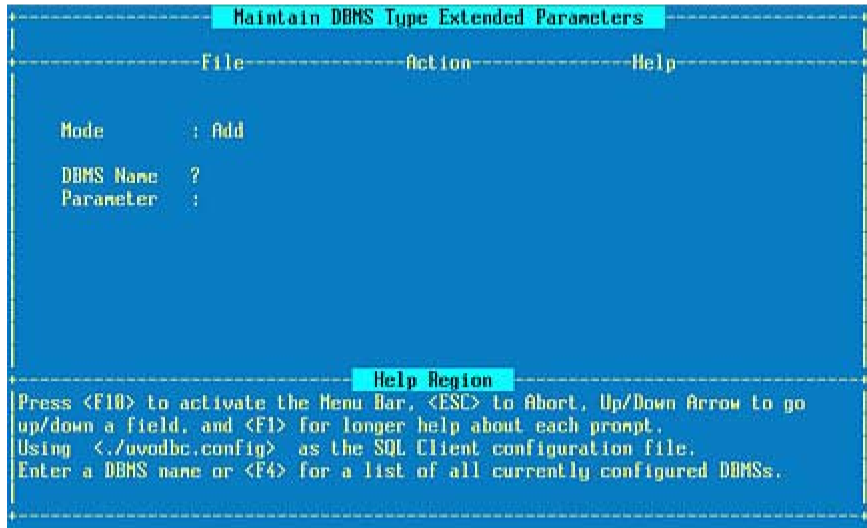
The **Mode** field indicates the current mode of operation of the menu. To change mode, use the **Action** menu on the menu bar. The **Action** menu has four options. After you enter a data source name and a parameter, you can choose one of the following actions:

Option	Description
Add	This is the default action. If the extended parameter is one value (for example, `MAXFETCHCOLS = 400`), the new value overwrites the old value. If the extended parameter has two values (for example, `MAPERROR = S0001 = 955`), it is parsed to see if it should overwrite or be added to the extended parameters of this data source.
List	Lists all the extended parameters currently defined for this data source. This output is the same as that produced by pressing **F4** in the **Parameter** field except that you cannot choose an entry.
Delete	If the **Parameter** field is empty, the extended parameters of the data source are listed, and you can choose an entry to delete. You cannot delete generic parameters that were not set by a user.
Modify	If the **Parameter** field is empty, the extended parameters of the data source are listed, and you can select choose an entry to change. If the **Parameter** field is blank, and the Mode is left as Modify, an error message appears stating that the field cannot be blank. If you change a parameter name, the original entry is unchanged. You can use Delete to remove the original entry. If the extended parameter has two values (for instance, `MAPERROR = S0001 = 955`), it is parsed to see if it should overwrite or be added to the extended parameters of this data source.

Action Menu Options

Adding or Changing Parameters for All Data Sources

When you choose the **Dbms type extended parameters** option, the following screen appears.



At the **DBMS Name** prompt, enter **`UNIVERSE`** or **`ODBC`**.

At the **Parameter** prompt, enter **`?`** (question mark) to list possible parameters. Press **F4** or enter **`*`** (asterisk) to see a list of all extended parameters currently defined for UniVerse. The chosen entries can then be added, modified, or deleted by using the options of the **Action** menu.

The **Mode** field indicates the current mode of operation of the menu. To change mode, use the **Action** menu on the menu bar. The **Action** menu has four options. After you enter a DBMS name and a parameter, you can choose one of the following actions:

Action	Description
Add	This is the default action. If the extended parameter is one value (for example, `MAXFETCHCOLS = 400`), the new value overwrites the old value. If the extended parameter has two values (for instance, `MAPERROR = S0001 = 955`), it is parsed to see if it should overwrite or be added to the extended parameters of this DBMS type.
List	Lists all the extended parameters currently defined for this DBMS type. This output is the same as that produced by pressing **F4** in the **Parameter** field except that you cannot choose an entry.
Delete	If the **Parameter** field is empty, the extended parameters of the DBMS type are listed, and you can choose an entry to delete. You cannot delete generic parameters that were not set by a user.
Modify	If the **Parameter** field is empty, the extended parameters of the DBMS type are listed, and you can choose an entry to change. If the **Parameter** field is blanked now, and the Mode is left as Modify, an error message appears stating that the field cannot be blank. If you change the parameter name, the original entry is unchanged. You can use Delete to remove the original entry. If the extended parameter has two values (for instance, `MAPERROR = S0001 = 955`), it is parsed to see if it should overwrite or be added to the extended parameters of this DBMS type.

Using the SQL Client Interface

After you configure the SQL Client Interface, you can do any of the following:

- Run the SQL Client Interface demonstration program SQLBCIDEMO
- Use the CONNECT command to connect to a data source
- Use UniVerse BASIC to write an SQL Client Interface application program

See Chapter 3, "Using the CONNECT Command," for details about the CONNECT command. See Chapter 4 "Using the SQL Client Interface," for details about the SQL Client Interface.

Running the Demonstration Program

When you install UniVerse, the installation process copies the SQLBCIDEMO demonstration program to the BP file of the UV account. Appendix B, "SQL Client Interface Demonstration Program," contains the source code for SQLBCIDEMO and explains what it does.

Create a Schema

You must run the demonstration program as an SQL user with appropriate privileges, and you must be in a UniVerse schema. To create a new schema on either a UNIX or a Windows system, complete the following steps:

1. Make a new directory:

   % **mkdir *newdir***

   C:*path*> **mkdir *newdir***

   newdir is the name of the new directory.

2. Have the DBA do the following:

   - Invoke UniVerse in the SQL catalog directory:

   % **cd /*uvpath*/sql/catalog**
   % **/*uvpath*/bin/uv**

   C:*path*> **cd \uvpath\sql\catalog**
   C:*path*> **uvsh**

   uvpath is the directory where UniVerse is installed.

   - Register you as an SQL user, if necessary:

   >**GRANT CONNECT TO *user*;**

   user is your login name.

   - Create a schema in your new directory:

   >**CREATE SCHEMA *name***
   SQL+**AUTHORIZATION *user***
   SQL+**HOME /*newdirpath*;**

   name is the name of the schema. On Windows platforms, *name* must be fully qualified. *newdirpath* is the full path of your new directory. On Windows platforms, the full path includes the drive letter.

3. Invoke UniVerse in *newdir*:

   % **cd /*newdirpath***
   % ***/uvpath*/bin/uv**

   C:*path*> **cd *newdirpath***
   C:*path*> **uvsh**

Run the Program

To run the demonstration program, complete the following steps:

1. To run the program, enter the following at the UniVerse prompt:

   >**SQLBCIDEMO**

   The program prompts you to enter the name of the data source to which you want to connect:

   Please enter the target data source ?

2. Enter the name of a data source defined in the configuration file.

   If you are connecting to a UniVerse data source, the prompt sequence is as follows:

```
Testing for data source connectivity....
Please enter the username for the server operating system
   login ?terry
Please enter the operating system password for user terry ?
Enter name or path of remote schema/account (hit return if
   local)?D:/users/terry/uv
```

   If you are connecting to an ODBC data source, the prompt sequence is as follows:

```
Testing for data source connectivity....
Enter the first DBMS connection parameter: ?smythe
Enter the second DBMS connection parameter:<Return>
```

   After the data source accepts your login parameters, the program displays output similar to the following:

```
Connecting to data source: localuv
Deleting local SQLCOSTAFF file
DELETEd file "SQLCOSTAFF", Type 2, Modulo 1.
DELETEd file "D_SQLCOSTAFF", Type 3, Modulo 1.
DELETEd file definition record "SQLCOSTAFF" in the VOC file.

Creating file "SQLCOSTAFF" as Type 2, Modulo 1, Separation 1.
Creating file "D_SQLCOSTAFF" as Type 3, Modulo 1, Separation 2.
Added "@ID", the default record for RetrieVe, to "D_SQLCOSTAFF".
```

```
File "SQLCOSTAFF" has been cleared.
Dropping SQLCOSTAFF table at localuv

Creating SQLCOSTAFF table at localuv

Setting values for the parameter markers

Prepare the SQL statement to load SQLCOSTAFF table
Loading row 1 of SQLCOSTAFF
Loading row 2 of SQLCOSTAFF
Loading row 3 of SQLCOSTAFF
Loading row 4 of SQLCOSTAFF
Loading row 5 of SQLCOSTAFF
Execute a SELECT statement against the SQLCOSTAFF table


Bind columns to program variables
EMPNUM    EMPNAME   GRADE    CITY
-------------------------------------
E1        Alice     12       Deale
E2        Betty     10       Vienna
E3        Carmen    13       Vienna
E4        Don       12       Deale
E5        Ed        13       Akron
Exiting SQLBCIDEMO

>
```

Chapter

3

Using the CONNECT Command

Command Syntax 3-3
    Command Options 3-3
Logging On to the Data Source 3-9
    Logging On to a Local UniVerse Server 3-10
    Logging On to a Remote UniVerse Server 3-10
    Logging On to an ODBC Data Source. 3-11
Executing SQL Statements on the Data Source 3-13
    Using Block Mode 3-13
Using Local Commands 3-15
Displaying and Storing Output 3-17
Examples 3-18
    Using Verbose Mode 3-19
    Changing the Display Width of Columns 3-20
    Exiting CONNECT 3-21
    Using UniVerse Output Mode 3-21
    Using Block Mode 3-23

This chapter describes how to use the CONNECT command to connect to a data source from a UniVerse client. You enter the CONNECT command at the UniVerse prompt. The CONNECT command lets you submit SQL statements to the data source and receive results at your terminal.

While you are connected to a data source, you can enter any SQL statement understood by the data source's DBMS engine, including SELECT, INSERT, UPDATE, DELETE, GRANT, and CREATE TABLE. You cannot, however, successfully submit commands understood only by a front-end tool at the server. ODBC data sources can use SQL language that is consistent with the ODBC grammar specification as documented in Appendix C of *Microsoft ODBC 2.0 Programmer's Reference and SDK Guide*.

The CONNECT command runs in autocommit mode: that is, all changes made to the data source's DBMS are committed immediately. Do not use transaction control statements such as BEGIN TRANSACTION, COMMIT, and ROLLBACK when you are using CONNECT. For information about transactions, see Transaction Management in Chapter 4, Using the SQL Client Interface.

Some database systems, such as SYBASE, treat SQL identifiers and keywords case-sensitively, whereas others such as ORACLE and INFORMIX do not. UniVerse treats SQL identifiers case-sensitively, but it treats SQL keywords as case-insensitive. For example, in UniVerse and SYBASE a table or column called BUDGET is different from one called Budget. ORACLE and INFORMIX treat these as duplicates. Also, SYBASE requires you to specify data types (char, int, float, and so forth.) in lowercase. In UniVerse, ORACLE, and INFORMIX you can use either upper- or lowercase letters for these keywords.

ODBC Data Sources

If you execute a stored procedure or enter a command batch with multiple SELECT statements, the results of only the first SELECT statement are returned.

Command Syntax

The syntax of the CONNECT command is as follows:

CONNECT *data.source* [*option setting* [*option setting …*]]

data.source is the name of the data source to which you want to connect. The data source must be defined in the *uvodbc.config* file. If you do not enter the name of a data source, CONNECT lists all data sources in the *uvodbc.config* file.

option is any of the following:

BLOCK	PREFIX
INVERT	UVOUT
MVDISPLAY	VERBOSE
NULL	WIDTH

Command Options

You can specify any option by typing the word or its first letter. Each option must be followed by a *setting*. The following section describes the options and their possible settings in detail.

BLOCK Option

The BLOCK option defines how input statements will be terminated. *setting* is one of the following:

Setting	Description
ON	Enables block mode. In this mode you can enter a series of SQL statements, ending each with a ; (semicolon). To terminate the block of SQL statements, press **ENTER** immediately after an `SQL+` prompt.
OFF	Disables block mode. In this mode if you type a semicolon at the end of a line of input, the SQL Client Interface terminates your input and sends it to the data source. This is the default setting.
string	Enables block mode (see ON, above). *string* must be from 1 to 4 characters. To terminate the block of SQL statements, enter *string* immediately after an `SQL+` prompt.

BLOCK Settings

See "Executing SQL Statements on the Data Source" on page 13 for more details.

INVERT Option

The INVERT option lets you control case inversion for alphabetic characters you type while CONNECT is running. *setting* is one of the following:

Setting	Description
ON	Inverts the case of all alphabetic characters you type—that is, lowercase letters change to uppercase, and uppercase letters change to lowercase. This is equivalent to setting PTERM CASE parameters to INVERT and LC-IN.
OFF	Disables case inversion. This is equivalent to setting PTERM CASE parameters to NOINVERT and LC-IN. This is the default setting for ODBC data sources.
INIT	Sets case-inversion parameters to the values they had when you invoked CONNECT. This is the default setting for UniVerse data sources.

INVERT Settings

When you exit CONNECT, case inversion for input is restored to the state it was in when you invoked CONNECT.

MVDISPLAY Option

The MVDISPLAY option defines how to display value marks in multivalued data selected from a UniVerse data source. For each row, multiple values in the same field are displayed on the same line, separated by value marks. *setting* is one of the following:

Setting	Description
SPACE	Displays a value mark as a blank space.
NOCONV	Displays a value mark as CHAR(253).
char	Displays a value mark as *char* (one character).

MVDISPLAY Settings

By default, value marks are displayed as * (asterisk).

NULL Option

The NULL option defines how to display the SQL null value. *setting* is one of the following:

Setting	Description
SPACE	Displays SQL null as a blank space.
NOCONV	Displays SQL null as CHAR(128).
string	Displays SQL null as *string*. The string can be from 1 to 4 characters. By default, UniVerse displays null as the four-character string NULL.

NULL Settings

PREFIX Option

The PREFIX option defines the prefix character for local commands. *setting* is any valid prefix character. The default prefix character is a period (**.**). You can use only the following characters as the prefix character:

!	exclamation point	?	question mark
@	at sign	(	left parenthesis
#	hash sign	)	right parenthesis
$	dollar sign	{	left brace
%	percent	}	right brace
&	ampersand	[	left bracket
*	asterisk	]	right bracket
/	slash	`	left quotation mark
\	backslash	‘	right quotation mark
:	colon	.	period
=	equal sign	|	vertical bar
+	plus sign	"	double quotation mark
–	minus sign	,	comma

PREFIX Settings

For more details see “Using Local Commands” on page 15.

UVOUT Option

The UVOUT option specifies how to handle output from SELECT statements executed on the data source. *setting* is either:

Setting	Description
filename	Stores output in *filename* on the client, then displays the output from *filename*. If the file does not exist, the CONNECT command creates it.
OFF	Displays output from the data source directly on the client's screen. This is the default setting.

UVOUT Settings

See "Displaying and Storing Output" on page 17 for more details.

VERBOSE Option

The VERBOSE option displays extended column information and system messages. *setting* is either:

Setting	Description
ON	Enables verbose mode. In this mode the name, SQL data type, precision, scale, and display size are displayed for each column's definition when selecting data from the data source. Error messages are displayed in extended format that includes the type of call issued, status, SQLSTATE, error code generated by the data source, and the complete error text.
OFF	Disables verbose mode. This is the default setting.

VERBOSE Settings

WIDTH Option

The WIDTH option defines the width of display columns. *setting* is one of the following:

Option	Description
col#,width	Sets the width of column *col#* to *width*. Do not type a space after the comma. Specify *col#* as * (asterisk) to set the width of all columns. *width* can be from 4 to the maximum line length allowed by your terminal. The default width for all columns is 10.
T	Truncates data that is wider than the specified *width*. This is the default setting.
F	Folds data that is wider than the specified *width* onto multiple lines.
?	Displays the current column width settings, and tells whether data will be truncated or folded.

WIDTH Options

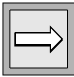
Logging On to the Data Source

After you enter the CONNECT command and the initial validity checks succeed, the SQL Client Interface does the following:

- If you are connecting to a UniVerse data source, you are prompted to enter your login name and password to connect to the server operating system. Next you are prompted to enter the name of the schema or account you want to connect to. You can enter a path in place of the schema or account name.
- If you are connecting to an ODBC data source, you are prompted to enter your user name and password to connect to the data source.

Be sure to enter these parameters in the proper upper- and lowercase letters. The password is not echoed to the screen.

The user name and password must be valid on the server machine. The default user name is your login name on the client system. You have three chances to enter your user name and password correctly.



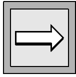
***Note**: You can connect to the local UniVerse server through a local connection or through the RPC facility (remote procedure call). To connect locally, the data source specification in the configuration file must specify localhost or the IP loopback address 127.0.0.1. If you are using a local connection to connect to the local UniVerse server, you are not prompted for your user name and password.*

After you log on successfully to the server operating system, and if the DBMS is currently running, the SQL Client Interface prompts you to enter the login parameters you use to access the DBMS on the server. The following table lists login parameters for each type of data source:

Data Source	Login Parameters
UNIVERSE	Either a UniVerse account or schema name, or the full path where the account or schema is located.
ORACLE	A user name and a password (not echoed to the screen).
SYBASE	A user name and a password (not echoed to the screen).

Login Parameters for Data Sources

You have three chances to enter your data source parameters correctly.



***Note**: If you are using a local connection to connect to the local UniVerse server, you are not prompted to enter a schema name or account name. You connect directly to the schema or UniVerse account in which you are currently working.*

After accepting the DBMS login parameters, the data source prompt appears:

```
data.source.name>
```

The next sections show examples of the login sequence on different systems. Brackets enclose default entries, which you can accept by pressing **ENTER**.

Logging On to a Local UniVerse Server

The following example shows what happens when you use a local connection to connect to the local UniVerse server:

```
>CONNECT localuv
localuv>
```

To connect to the local UniVerse server through the RPC, the data source specification in the configuration file must specify the host name of the local UniVerse server. The following example shows what happens when you connect to the local UniVerse server through the RPC:

```
>CONNECT uv
Enter your login name for server operating system [george]: fred
Enter password for fred (use SHIFT for lower case):
Enter name or path of remote schema/account [FRED]:<Return>
'FRED' is a non-existent or invalid schema/account on 'uv'
Enter name or path of remote schema/account [FRED]: SALES
uv>
```

Logging On to a Remote UniVerse Server

The following example shows what happens when you use CONNECT to log on to a remote UniVerse server:

```
>CONNECT remuv
Enter your login name for server operating system [fred]: george
Enter password for george (use SHIFT for lower case):
Enter name or path of remote schema/account [GEORGE]:<Return>
'GEORGE' is a non-existent or invalid schema/account on 'remuv'
Enter name or path of remote schema/account [GEORGE]: SALES
remuv>
```

Logging On to an ODBC Data Source

The following example shows what happens when you use CONNECT to log on to an ODBC data source:

```
>CONNECT odbc-ora
Enter username for connecting to 'odbc-ora' DBMS [VEGA\george]:
fred
Enter password for fred:
odbc-ora>
```

Errors When Logging On to a Data Source

The SQL Client Interface does several validity checks when you enter **CONNECT** ***data.source***. It runs these checks before prompting you for your user name and password. The most common errors that can occur at this point are the following:

- *uvserver* is not defined in the *uvrpcservices* file of the UniVerse server. You will see a message similar to the following example:

```
>CONNECT uv
SQLConnect error: Status = -1 SQLState = 08001 Natcode = 81016
[RPC] error code = 81016
Connection attempt to 'uv' failed; - RPC daemon probably not
running
```

- The data source is not defined in the configuration file (for example, you may have misspelled it). You will see a message similar to the following example:

```
>CONNECT badname
Data source 'badname' not found in uvodbc.config
```

- The server does not respond. This can be due to problems on the network or problems with the server software. You will see a message similar to the following example:

```
>CONNECT fred-vega-sybsrv
Enter username for connecting to 'fred-vega-sybsrv' DBMS
   [VEGA\george]: fred
Enter password for fred:
SQLConnect error: Status = -1  SQLState = 01000 Natcode = 4
[ODBC:4][INTERSOLV][ODBC SQL Server driver][SQL
Server]ct_connect():
   network packet lay
```

- The configuration file contains incorrect information for the requested data source or for its DBMS type. You will see a message similar to the following example:

```
>CONNECT syb
SQLConnect error: Status = -1  SQLState = IM997  Natcode = 0
[SQL Client] An illegal configuration option was found
Invalid parameter(s) found in uvodbc.config file
```

- If the DBMS is not currently running on the server, you will a message similar to the following example:

```
SQLConnect error:   Status = -1   SQLState = S1000   Natcode =
9352

[ODBC] [INTERSOLV][ODBC Oracle driver][Oracle]ORA-09352: Windows
32-
   bit

Two-Task driver unable to spawn new ORACLE task
```

The failure of a connection to an ODBC data source generates error messages particular to that ODBC driver and database server.

Executing SQL Statements on the Data Source

At the data source prompt you can enter any valid SQL statement, or you can enter a local command beginning with the prefix character. After processing the statement or command, the data source prompt reappears.

How you terminate your SQL input depends on whether block mode is enabled or disabled. If block mode is disabled (the default), you must end SQL statements with a : (semicolon) or a ? (question mark). Statements ending with a semicolon are executed on the data source. Statements ending with a question mark are not sent to the data source but are stored on the client. The most recently entered statement is stored on the client so you can recall it and edit it.

You can enter an SQL statement on several lines. If a statement line does not end with a semicolon or a question mark and you press **ENTER**, the SQL continuation prompt appears:

SQL+

You can terminate your input by pressing **ENTER** immediately after an `SQL+` prompt. SQL statements can be up to 38 lines long.

Using Block Mode

Block mode lets you send a series of SQL statements, each terminated with a semicolon, to the data source as a single block. For instance, you would use block mode to send PL/SQL blocks or stored procedures to an ODBC data source running ORACLE.

If block mode is enabled, a semicolon does not terminate your input. SQL statements are sent to the data source for execution only when you press **ENTER** or enter a termination string immediately after an `SQL+` prompt. SQL statements ending with a question mark are not sent to the data source; they are stored on the client.

Use the BLOCK option of the CONNECT command or the local command .BLOCK to enable and disable block mode.

You can enable block mode in two ways. With block mode set to ON, you terminate input by pressing **ENTER** immediately after an `SQL+` prompt.

With block mode set to a character string, you enter the character string immediately after an `SQL+` prompt to terminate input. For example, you might want to terminate your input with a line such as GO. The string can be up to four characters long. The string is not case-sensitive, so if you specify GO with the BLOCK option or the .B command, for example, you can terminate input with **`GO`** or **`go`**.

Using Local Commands

Commands starting with the designated prefix character are treated as local commands—that is, the client machine processes them. The default prefix character is a . (period). You can change the prefix character by using the PREFIX option of the CONNECT command.

You can enter local commands as a word or as the first letter of the word. Local commands cannot end with a semicolon or a question mark. The following are valid local commands:

Command	Description
.A *string*	Appends *string* to the most recent SQL statement.
.B [LOCK] *setting*	Enables or disables block mode. *setting* can be ON, OFF, or a character string. See "Executing SQL Statements on the Data Source" on page 13 for details.
.C/*old*/*new* [/[G]]	Changes the first instance of *old* to *new* in the most recent SQL statement. If you use the G (global) option, .C changes all instances of *old* to *new*. You can replace the slash with any valid delimiter character. Valid delimiters are the same as valid prefix characters. For a list of valid delimiters, see the "PREFIX Option" on page 6.
.EXECUTE	Executes the most recent SQL statement.
.I [NVERT] *setting*	Enables or disables case inversion for alphabetic characters you type. *setting* can be ON, OFF, or INIT. For more details see the "INVERT Option" on page 4.
.M [VDISPLAY] *setting*	Defines how to display value marks in multivalued data. *setting* can be SPACE, NOCONV, or a character.
.N [ULL] *setting*	Defines how to display the SQL null value. *setting* can be SPACE, NOCONV, or a character string.
.P [RINT]	If the most recent SQL statement is SELECT, executes the statement and sends output to logical print channel 0. If the most recent SQL statement is not SELECT, executes the statement.

Valid Local Commands

Command	Description
.Q[UIT]	Exits from CONNECT and returns to the UniVerse prompt.
.R[ECALL] [*name*]	Displays, but does not execute, the SQL statement stored as *name* in the VOC. If you do not specify *name*, .RECALL displays the most recent SQL statement.
.S[AVE] *name*	Saves the most recent SQL statement as the sentence *name* in the VOC file.
.T[OP]	Clears the screen.
.U[VOUT] *setting*	Specifies how to handle output from SELECT statements. *setting* can be OFF or the name of a file on the client system. See "Displaying and Storing Output" on page 17 for details.
.V[ERBOSE] *setting*	Enables or disables verbose mode. *setting* can be ON or OFF. For more details see the "VERBOSE Option" on page 7.
.W[IDTH] *setting*	Defines the width of display columns. For information about how to set and display column widths, see the WIDTH option of the CONNECT command.
.X	Executes the most recent SQL statement.

Valid Local Commands (Continued)

Displaying and Storing Output

Use the UVOUT option or the .UVOUT local command to turn UniVerse output mode on and off. By default UniVerse output mode is off, and CONNECT displays output from SQL statements on the screen as follows. If a row of output is wider than the line length defined for your screen, CONNECT displays each row in UniVerse's vertical mode, with each column on a separate line. Otherwise, blank columns two characters wide separate display columns. CONNECT folds column headings into several lines if they do not fit on one line, and truncates or folds data that is wider than a column, depending on the WIDTH setting (T or F). An * (asterisk) appears next to truncated data, a - (hyphen) appears next to folded data.

In UniVerse output mode, CONNECT writes each row of data to a UniVerse file on the client (the file is first cleared). It then generates a UniVerse SQL SELECT statement that displays the data in the file.

In both output modes, data is left-justified if its type is CHAR or DATE, text-justified if its type is VARCHAR, and right-justified if it is any other data type.

You can use UniVerse output mode to transfer data from the data source to your UniVerse database. This is because output from a SELECT statement is stored in a UniVerse file. However, each SELECT clears the UniVerse output file before writing output to it. If you want to save the results of several SELECTs in your UniVerse database, you must use several UniVerse output files.

Examples

The following example shows a normal login to a UniVerse data source:

```
>CONNECT uv
Enter your login name for server operating system [josh]:<Return>
Enter password for josh (use SHIFT for lower case):
Enter name or path or remote schema/account [JOSH]: SALES
uv> LIST VOC
SQL+
[UNIVERSE:930145] UniVerse/SQL: LIST not a SQL DDL or DML verb.
uv> SELECT * FROM VOC LPTR;
[UNIVERSE:930142] UniVerse/SQL: Most Report Qualifiers are not
supported for Clients, scanned command was FROM VOC SELECT * LPTR;
uv> .C/LPTR/SAMPLE 2
SELECT * FROM VOC SAMPLE 2
uv> .X
NAME           TYPE   DESC
-------------- ----  -----------------------------------
GLOBAL.CATDIR   F     File - Used to access system c*
STAT            V     Verb - Produce the STAT of a n*

2 rows selected
uv> .Q
Disconnecting from 'uv'
>
```

The LIST command fails because the server accepts only DDL and DML statements from the client. The first SELECT statement fails because LPTR is not valid in programmatic SQL. The second SELECT statement succeeds, and the user disconnects from the data source.

The next example shows a normal login to an ODBC data source running SYBASE. Some data is selected from a table.

```
>CONNECT syb
Enter username for connecting to 'syb' DBMS [hatch]:<Return>
Enter password for hatch:
syb> select * from tedtab1;
        pk  colchar8      colint     colreal
----------  --------  ----------  ----------
         1  New York        9876  3.40000009*
         2  Chicago          543  23.3999996*

2 rows selected
```

Using Verbose Mode

The next example turns on verbose mode and executes the previous SELECT statement:

```
syb> .v on
syb> .x
There are 4 columns
Column 1 name is:  pk
Column 1 type is:  4  (SQL.INTEGER)
Column 1 prec is:  10
Column 1 scale is: 0
Column 1 dispsize is: 11
Column 2 name is:  colchar8
Column 2 type is:  1  (SQL.CHAR)
Column 2 prec is:  8
Column 2 scale is: 0
Column 2 dispsize is: 8
Column 3 name is:  colint
Column 3 type is:  4  (SQL.INTEGER)
Column 3 prec is:  10
Column 3 scale is: 0
Column 3 dispsize is: 11
Column 4 name is:  colfloat
Column 4 type is:  8  (SQL.DOUBLE)
Column 4 prec is:  15
Column 4 scale is: 0
Column 4 dispsize is: 22
        pk  colchar8      colint    colfloat
----------  --------  ----------  ----------
         1  New York        9876  3.40000009*
         2  Chicago          543  23.3999996*

2 rows selected
syb> .v off
syb> .w f
syb> .x
        pk  colchar8      colint    colfloat
----------  --------  ----------  ----------
         1  New York        9876  3.40000009-
                                  5
         2  Chicago          543  23.3999996-
                                  19

2 rows selected
syb> .w 4,15
syb> .x
        pk  colchar8      colint      colfloat
----------  --------  ----------    ----------
         1  New York        9876   3.400000095
         2  Chicago          543  23.399999619
```

```
2 rows selected
syb> .w
Truncate/Fold mode is:            F
Column width settings are:
                     Column 4:   15
             All other columns:  10
```

Changing the Display Width of Columns

The following example shows what happens when you change the display width to fold data that does not fit (as shown in the previous example, in the `colreal` column):

```
syb> .w f
syb> .x
        pk  colchar8      colint     colreal
----------  --------  ----------  ----------
         1  New York        9876  3.40000009-
                                  5
         2  Chicago          543  23.3999996-
                                  19

2 rows selected
```

By changing the display width of column 4 to 15 characters, you get the following display:

```
syb> .w 4,15
syb> .x
        pk  colchar8      colint          colreal
----------  --------  ----------  ---------------
         1  New York        9876      3.400000095
         2  Chicago          543     23.399999619

2 rows selected
```

To display the current display width settings, use a question mark after the .W command, as follows:

```
syb> .w ?
Truncate/Fold mode is:          F
Column width settings are:
                    Column 4:  15
            All other columns: 10
```

Exiting CONNECT

The next example inserts values into a table, selects and displays them, then quits from CONNECT:

```
syb> insert into tedtab3 values (3,9,1,8);
1 row affected
syb> select * from tedtab3;
        pk  coltinyint      colbit         colsmint
----------  ----------  ----------  ---------------
         1         255           0           -32768
         2           0           0            32768
         3           9           1                8

3 rows selected
syb> .q
Disconnecting from 'syb' database
>
```

Using UniVerse Output Mode

The following example shows how to use two UniVerse output files to save the results of two SELECT statements.

First, enter UniVerse output mode while you are connected to the data source *ora*:

```
ora> .U UVFILE1
Opening file UVFILE1
```

Next, select data from a table in *ora*:

```
ora> SELECT * FROM TEDTAB2;
COLC......   COLD......

detroit      lions
pittsburgh   steelers
new york     giants

3 records listed.
```

Now switch to a different UniVerse output file and enter another SELECT statement:

```
ora> .U UVFILE2
Closing file UVFILE1
Creating file "UVFILE2" as Type 30.
Creating file "D_UVFILE2" as Type 3, Modulo 1, Separation 2.
Added "@ID", the default record for RetrieVe, to "D_UVFILE2".
Opening file UVFILE2
ora> SELECT DISTINCT COLM, COLM+3, COLM*7 FROM TEDTAB7;
COLM......   COLM+3....   COLM*7....

         0            3            0
         1            4            7
         4            7           28
         6            9           42

4 records listed.
```

Next, exit CONNECT and enter two SELECT statements, one on UVFILE1 and one on UVFILE2:

```
ora> .Q
Disconnecting from 'ora' database
>SELECT * FROM UVFILE1;
COLC......   COLD......

detroit      lions
pittsburgh   steelers
new york     giants

3 records listed.
>SELECT * FROM UVFILE2;
COLM......   COLM+3....   COLM*7....

         6            9           42
         0            3            0
         1            4            7
         4            7           28

4 records listed.
>
```

Using Block Mode

In the following example, the .B command enables block mode. The user enters a multiline SELECT statement, terminating it with the string GO instead of a semicolon.

```
syb> .b go
syb> select * from emps
SQL+
SQL+where deptno < 300
SQL+go
    empno  lname         fname          deptno
---------  ----------   ----------   ---------
    17543  Smith         George            301
    23119  Brown         George            307

2 rows selected
```

The next example enables block mode and creates a stored procedure on an ORACLE database:

```
ora> .B ON
ora> CREATE PROCEDURE sal_raise (emp_id IN NUMBER,
SQL+sal_incr IN NUMBER) AS
SQL+BEGIN
SQL+UPDATE emp
SQL+SET sal = sal + sal_incr
SQL+WHERE empno = emp_id;
SQL+IF SQL%NOTFOUND THEN
SQL+raise_application_error (-20011, 'Invalid Employee Number:'||
SQL+TO_CHAR (emp_id));
SQL+END IF;
SQL+END sal_raise;
SQL+<Return>
ora>
```

Chapter

Using the SQL Client Interface

4

Establishing a Connection to a Data Source 4-3
Connecting to NLS-Enabled Data Sources 4-3
Allocating the Environment 4-5
Allocating the Connection Environment 4-5
Connecting to a Data Source. 4-5
Processing SQL Statements 4-8
Allocating the SQL Statement Environment 4-8
Executing SQL Statements 4-8
Processing Output from SQL Statements 4-11
Freeing the SQL Statement Environment 4-13
Terminating the Connection 4-14
Transaction Management 4-18
Distributed Transactions 4-18
Nested Transactions 4-19
Detecting Errors 4-21
UniVerse Error and System Messages 4-22
Displaying Environment Variables in RAID 4-24

UniVerse BASIC programs use the SQL Client Interface to exchange data between a UniVerse client and a server data source. A data source is a network node and a DBMS on that node. For example, you might have an order entry system on the machine running UniVerse and want to post this information to a central database. You use the SQL Client Interface to connect your applications to the data source and exchange data.

This chapter describes how to do the following:

- Establish a connection to a data source
- Execute SQL statements at the data source
- Execute procedures stored on the data source
- Terminate the connection

Establishing a Connection to a Data Source

Before connecting to a data source, the application does two things:

- It creates an *SQL Client Interface environment*, in which all connections to both UniVerse and ODBC data sources are established.
- It creates one or more *connection environments*. Each connection environment supports a connection to a single data source.

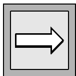
If you are connecting to a UniVerse data source, it must be defined in the *uvodbc.config* file.

As of Release 9, each UniVerse process opens a local connection to itself. This is true for both user and server processes. BASIC programs running on a UniVerse server can use the @variables @HENV, @HDBC, and @HSTMT to refer to this local connection. Using these @variables in your programs lets you execute programmatic SQL statements without having to allocate SQL Client Interface, connection, and statement environments.

Connecting to NLS-Enabled Data Sources

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

Certain combinations of UniVerse client and server may not reliably transfer data because of a mismatch in the locale settings at the client end. The following sections show which combinations of server locale specifications are allowed depending on the type of client and the NLS states of both client and server.



***Note**: The SQL Client Interface can connect to an ODBC data source only if the client system is running with NLS mode turned off.*

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

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

Server NLS Locale State	Client NLSLocaleState	Action
ON	ON, or requested via **SQLSetConnectOption**	Client NLSLOCALE or requested NLSLOCALE used if ID is valid. If ID is not valid, the connection is rejected.
	OFF, and not requested via **SQLSetConnectOption**	Server default locale used if valid.
OFF	ON	Connection rejected unless requested locale is WIN:0409, US-ENGLISH, or OFF.
	OFF	No server locale used.

Locale Specifications Combinations

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

UniVerse releases before 9.4 do not support NLS. Therefore any Release 9.3 (or earlier) client can connect to a Release 9.4 (or later) server only if the default server map as set by the NLSDEFSRVMAP is ISO8859-1+MARKS or ASCII+MARKS. If server locale support is enabled, NLSDEFSRVLC is used, if valid.

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

Because UniVerse releases before 9.4 do not support NLS, Release 9.4 (or later) clients can connect to Release 9.3 (or earlier) servers only if the requested locale is WIN:0409, US-ENGLISH, or OFF.

Allocating the Environment

The SQL Client Interface environment is a data structure that provides the context for all future connections to both UniVerse and ODBC data sources. You allocate the environment with the SQLAllocEnv function. This function allocates memory for an environment and returns its address in a variable. This variable is the environment *handle*. You can allocate more than one environment at a time.

UniVerse BASIC programs running locally on a UniVerse server can use the @variable @HENV to refer directly to the SQL Client Interface environment. They do not need to allocate one.

Allocating the Connection Environment

The connection environment is a data structure that supports a connection to a single data source. One SQL Client Interface environment can support many connection environments. You allocate the connection environment with the SQLAllocConnect function. This function allocates memory for a connection environment and returns its address, or handle, in a variable. Use this variable when you refer to a specific connection. You can establish more than one connection environment at a time.

UniVerse BASIC programs running locally on a UniVerse server can use the @variable @HDBC to refer directly to the connection environment. They do not need to allocate one.

Connecting to a Data Source

Before or after issuing an **SQLConnect** call to connect to a UniVerse data source, an application can establish certain conditions for the connection by issuing one or more SQLSetConnectOption calls. Use the **SQLSetConnectOption** call to specify the default transaction isolation level, specify the NLS locale information to use, or supply the user ID and password for the connection, if required.

You do not use **SQLSetConnectOption** for login ID and password if you are connecting to the *localuv* server.

Use SQLConnect to establish a session between your application and the DBMS on the server. A BASIC program running locally on a UniVerse server does not need to execute **SQLConnect**. It is automatically connected to the *localuv* server, at the schema or account where the program is running. When connecting to a remote UniVerse server, the **SQLConnect** function contains an account or schema name, or a path specifying where the UniVerse data source is located. When connecting to an ODBC server, **SQLConnect** contains the name of the data source and information needed to log on to the data source. After the connection is established, you can allocate an SQL statement environment to prepare to process SQL statements.

Connecting to a UniVerse Server with NLS Enabled

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

The server honors the following configurable parameters in the *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. If the value is 1, users can switch of NLS mode by setting their $UVNLSOFF environment variable to 1.
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 a UniVerse server, which is used by all client programs accessing the server.

NLS-Enabled Configuration Parameters

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

1. It gets the current locale settings.
2. It reads the *uvodbc.config* file, if found, and replaces the current locale settings with values set by *uvodbc.config*.

Client programs can use the SQLSetConnectOption to override any of the default locale values.

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

Processing SQL Statements

First you allocate an SQL statement environment, then you execute SQL statements. If your application is running locally on a UniVerse server, it can execute SQL statements without allocating a statement environment by using the @variable @HSTMT.

Allocating the SQL Statement Environment

The SQL statement environment is a data structure that provides a context for delivering SQL statements to the data source, receiving data from the data source row by row, and sending data to the data source. An SQL statement environment belongs to one connection environment. You allocate the SQL statement environment with the SQLAllocStmt function. This function allocates memory for an SQL statement environment and returns its address, or handle, in a variable.

You can establish more than one SQL statement environment for the same connection environment.

BASIC programs running locally on a UniVerse server can use the @variable @HSTMT to refer directly to the SQL statement environment. They do not need to allocate one.

Executing SQL Statements

You can execute SQL statements in two ways:

- Direct execution
- Prepared execution

Your SQL statements must conform to the SQL language and conventions supported by the server's DBMS engine. For example, you cannot execute RetrieVe commands on a UniVerse data source, or SQL*Plus commands such as DESCRIBE on an ORACLE data source. ODBC data sources can use SQL language that is consistent with the ODBC grammar specification as documented in Appendix C of the *Microsoft ODBC 2.0 Programmer's Reference and SDK Guide*.

If you are executing SQL statements on an ODBC data source running SYBASE, remember that it may treat identifiers and keywords case-sensitively. Also, if you are connected to an ODBC data source and you execute a stored procedure or enter a command batch with multiple SELECT statements, the results of only the first SELECT statement are returned.

Executing SQL Statements Directly

Direct execution is the simplest way to execute an SQL statement. Use the SQLExecDirect function to execute an SQL statement once, or when your program does not need information about the column results before executing the statement.

Preparing and Executing SQL Statements

Use prepared execution when you want to execute the same SQL statement more than once or when you need information about SQL statement results before the statement is executed. Use the SQLPrepare and SQLExecute functions for prepared execution. **SQLPrepare** prepares the SQL statement once, then **SQLExecute** is called each time the SQL statement is to be executed.

For example, if you are inserting many data rows in a table, use the **SQLPrepare** function once, then use one **SQLExecute** for each row you insert. Before each **SQLExecute** call, set new values for the data to insert. To set new data values, you use parameter markers in your SQL statement (see "Using Parameter Markers in SQL Statements" on page 9). Using the **SQLPrepare** and **SQLExecute** functions in this way is more efficient than using separate **SQLExecDirect** calls, one for each row.

Using Parameter Markers in SQL Statements

You can use parameter markers in SQL statements to mark the place where you will insert values to send to the data source. If your SQL statements contain parameter markers, you must call SQLBindParameter once for each marker, to specify where to find the current value for each marker. For example, assume you create a table on the data source with the following command:

```
CREATE TABLE STAFF
   ( EMPNUM CHAR(3) NOT NULL,
   EMPNAME CHAR(20),
   GRADE INT,
   CITY CHAR(15))
```

To insert data into this table, you might use SQLExecDirect to execute a series of INSERT statements. For example:

```
STATUS = SQLExecDirect(STMTENV,"INSERT INTO STAFF VALUES ('E9',
'Edward',
   10, 'Arlington')")
STATUS = SQLExecDirect(STMTENV,"INSERT INTO STAFF VALUES ('E10',
'John',
   12, 'Belmont')")
STATUS = SQLExecDirect(STMTENV,"INSERT INTO STAFF VALUES ('E11',
'Susan',
   13, 'Lexington')")
STATUS = SQLExecDirect(STMTENV,"INSERT INTO STAFF VALUES ('E12',
'Janet',
   13, 'Waltham')")
```

The **SQLExecDirect** function takes two input variables. The first, STMTENV, is the name of the SQL statement environment. The second is the INSERT statement to be sent to the data source for execution.

Using several **SQLExecDirect** calls to insert data in this way is relatively slow. A better way to do this is to prepare the following INSERT statement for execution:

SQL = "INSERT INTO STAFF VALUES (?, ?, ?, ?)"
STATUS = SQLPrepare(STMTENV,SQL)

Each question mark in the statement is a parameter marker representing a value to be obtained from the application program when you execute the statement. You use the SQLBindParameter function to tell the SQL Client Interface where to find the variables that will resolve each question mark in the statement. When the **SQLExecute** call is issued, the SQL Client Interface picks up the variables you provided for these parameter markers, executes any required data conversions, and sends them to the data source, which executes the SQL statement with the new values.

Before you execute this statement, use **SQLBindParameter** calls to inform the SQL Client Interface where to find the parameter values to use in the statement. For example:

```
STATUS = SQLBindParameter(STMTENV, 1, SQL.B.BASIC, SQL.CHAR, 3, 0,
EMPNUM)
STATUS = SQLBindParameter(STMTENV, 2, SQL.B.BASIC, SQL.CHAR, 20,
0, EMPNAME)
STATUS = SQLBindParameter(STMTENV, 3, SQL.B.BASIC, SQL.INTEGER, 0,
0, GRADE)
STATUS = SQLBindParameter(STMTENV, 4, SQL.B.BASIC, SQL.CHAR, 15,
0, CITY)
STATUS = SQLPrepare(STMTENV, "INSERT INTO STAFF VALUES ( ?, ?, ?,
```

```
? )")
    .
    .
    .
EMPNUM = 'E9'
EMPNAME = 'Edward'
GRADE = 10
CITY = 'Arlington'
STATUS = SQLExecute(STMTENV)
EMPNUM = E10'
EMPNAME = 'John'
GRADE = 12
CITY = 'Belmont'
STATUS = SQLExecute(STMTENV)
    .
    .
    .
```

The **SQLBindParameter** function takes seven input variables. The first, STMTENV, is the name of the SQL statement environment. The second is the number of the parameter marker in the INSERT statement to be sent. The third (SQL.B.BASIC) and fourth (SQL.CHAR, SQL.INTEGER) specify data types, used to convert the data from BASIC to SQL data types. The fifth and sixth specify the parameter's precision and scale. The seventh is the name of the variable that will contain the value to use in the INSERT statement.

You can also use parameter markers with SELECT statements when you want to specify variable conditions for queries. For example, you might use the following statements to select rows from STAFF when CITY is a variable loaded from your application:

```
STATUS = SQLBindParameter(STMTENV, 1, SQL.B.BASIC, SQL.CHAR, 15,
0, CITY)
STATUS = SQLPrepare(STMTENV, "SELECT * FROM STAFF WHERE CITY = ?")
PRINT "ENTER CITY FOR QUERY":
INPUT CITY
STATUS = SQLExecute(STMTENV)
```

Processing Output from SQL Statements

Once you execute an SQL statement at the data source, you can issue calls that provide more detail about the results and that let you bring results from the data source back to your application.

You use the SQLNumResultCols function to find out how many columns the SQL statement produced in the result set. If it finds columns, you can use SQLDescribeCol or SQLColAttributes to get information about a column, such as its name, the SQL data type it contains, and (on UniVerse data sources) whether it is defined as multivalued.

You use the SQLRowCount function to find out if the SQL statement changed any rows in the table. For example, if an SQL UPDATE statement changes 48 rows, **SQLRowCount** returns 48.

If an SQL statement produces a set of results at the data source, we say that a *cursor* is opened on the result set. You can think of this cursor as a pointer into the set of results, just as a cursor on a screen points to a particular line of text. An open cursor implies that there is a set of results pending at the data source.

To bring the results of the SQL statement from the data source to your application, use the SQLBindCol and SQLFetch functions. You use **SQLBindCol** to inform the SQL Client Interface where to put data from a specific column and what application data type to store it as. For example, to print rows from the STAFF table, you might write the following:

```
SQLBindCol(STMTENV, 1, SQL.B.DEFAULT, EMPNUM)
SQLBindCol(STMTENV, 2, SQL.B.DEFAULT, EMPNAME)
SQLBindCol(STMTENV, 3, SQL.B.DEFAULT, EMPGRADE)
SQLBindCol(STMTENV, 4, SQL.B.DEFAULT, EMPCITY)

LOOP
WHILE STATUS <> SQL.NO.DATA.FOUND DO
   STATUS = SQLFetch(STMTENV)
   IF STATUS <> SQL.NO.DATA.FOUND
   THEN
      PRINT EMPNUM form:EMPNAME form:EMPGRADE form:EMPCITY
   END
REPEAT
```

The **SQLBindCol** function takes four input variables. The first, STMTENV, is the name of the SQL statement environment. The second is the number of the column. The third specifies the data type to which to convert the incoming data. The fourth is the name of the variable where the column value is stored.

For each column, a call to **SQLBindCol** tells the SQL Client Interface where to put each data value when **SQLFetch** is issued. Each **SQLFetch** stores the next row of data in the specified variables. Normally you fetch data rows until the end-of-data status flag is returned to the **SQLFetch** call.

The SQL.B.DEFAULT argument to **SQLBindCol** lets the result column's SQL data type determine the BASIC data type to which to convert data from the data source. For information about converting data, see Appendix A, "Data Conversion."

For other examples showing how to execute SQL statements, see Appendix B, "SQL Client Interface Demonstration Program."

Freeing the SQL Statement Environment

Once all processing of an SQL statement is complete, use **SQLFreeStmt** to free resources in an SQL statement environment. Use one of the following options in the **SQLFreeStmt** call:

- SQL.CLOSE closes any open cursor associated with an SQL statement environment and discards any pending results. The SQL statement environment can then be reused by executing another SQL statement with the same or different parameters and bound column variables. SQL.CLOSE releases all locks held by the data source.
- SQL.UNBIND releases all bound column variables set by Description for the SQL statement environment.
- SQL.RESET.PARAMS releases all parameter marker variables set by SQLBindParameter for the SQL statement environment.
- SQL.DROP releases the SQL statement environment, frees all resources, closes any cursor, and cancels all pending results. All column variables are unbound and all parameter marker variables are reset. This option terminates access to the SQL statement environment.

For example, the following statement frees the SQL statement environment in the demonstration program (see Appendix B, "SQL Client Interface Demonstration Program"):

```
STATUS = SQLFreeStmt(STMTENV, SQL.DROP)
```

Terminating the Connection

When your program is ready to terminate the connection to the data source, it should do the following:

- Disconnect from the data source
- Release the connection environment
- Release the SQL Client Interface environment

UniVerse BASIC programs running locally on a UniVerse server (*localuv*) using @HSTMT need not disconnect, release the connection or SQL Client Interface environments.

Use SQLDisconnect to close the connection associated with a specific connection environment. All active transactions must be committed or rolled back before disconnecting (see "Transaction Management" on page 18). If there are no transactions pending, **SQLDisconnect** frees all allocated SQL statement environment*s*.

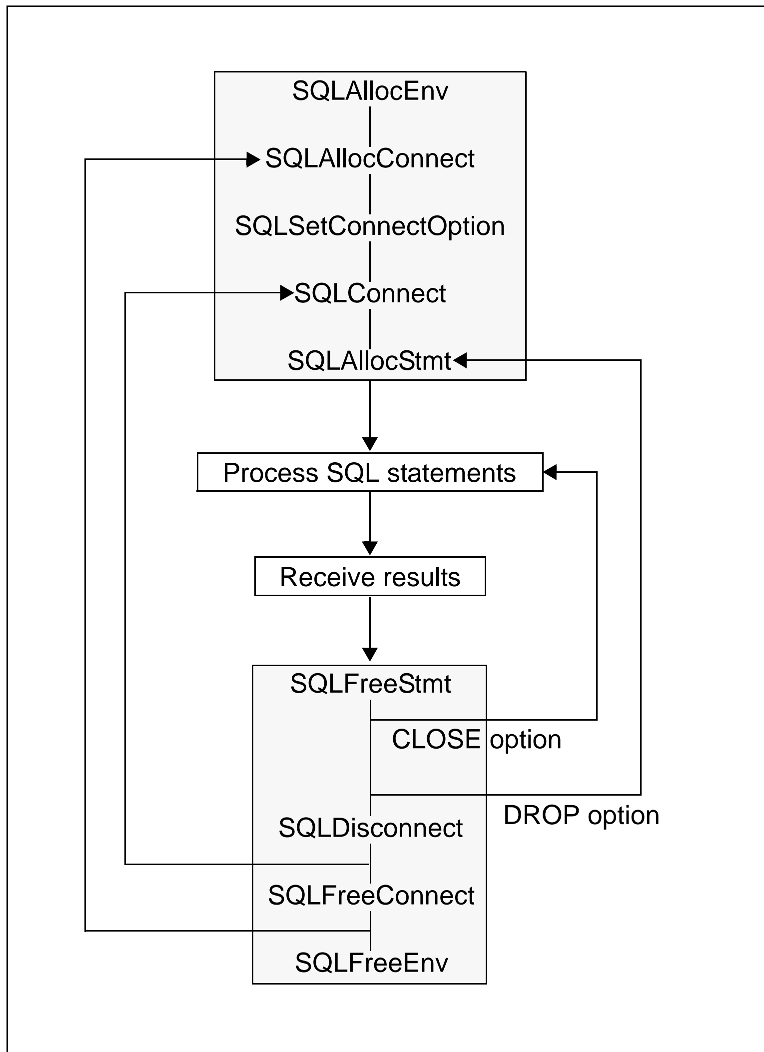
Use SQLFreeConnect to release a connection environment and its associated resources. The connection environment must be disconnected from the data source before you use this call or an error occurs.

Use SQLFreeEnv to release the SQL Client Interface environment and all resources associated with it. Disconnect all sessions with the **SQLDisconnect** and **SQLFreeConnect** functions before you use **SQLFreeEnv**.

In the demonstration program (see Appendix B, "SQL Client Interface Demonstration Program"), the following statements close the connection to the data source and free the connection and SQL Client Interface environments:

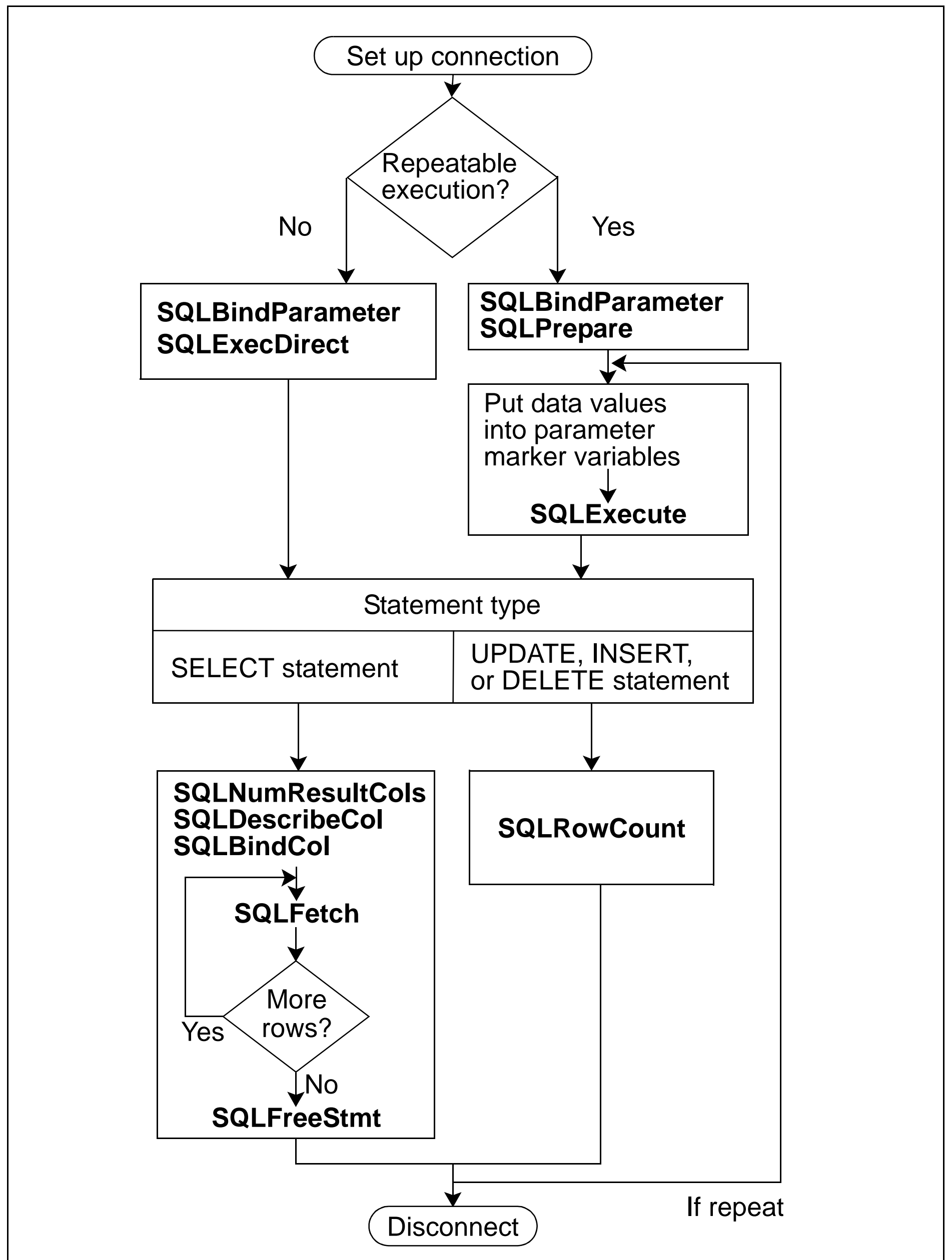
```
STATUS = SQLDisconnect(CONENV)
STATUS = SQLFreeConnect(CONENV)
STATUS = SQLFreeEnv(DBCENV)
```

The following example shows some function calls used in a BASIC application.



Function Calls Used in a Simple BASIC Application

The next example shows the order of function calls you use to execute a simple SQL statement.

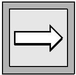


Order of Function Calls

Transaction Management

Use the UniVerse BASIC statements BEGIN TRANSACTION, COMMIT, ROLLBACK, and END TRANSACTION to provide transaction management control in your application.

- Outside a transaction, changes are committed immediately (autocommit mode).
- Use BEGIN TRANSACTION to put all current connections into manual commit mode. Connections established within a transaction are also in manual commit mode.
- Use COMMIT or ROLLBACK to terminate the transaction.
- Use END TRANSACTION to indicate where to continue processing after the preceding COMMIT or ROLLBACK statement is executed.



***Note**: Your application programs should not try to issue transaction control statements directly to the data source (for instance, by issuing a COMMIT statement with SQLExecDirect or SQLPrepare). Programs should use only BASIC transaction control statements. The SQL Client Interface issues the correct combination of transaction control statements and middleware transaction control function calls that are appropriate for the DBMS you are using. Trying to use* **SQLExecDirect** *and* **SQLExecute** *to execute explicit transaction control statements on ODBC data sources can cause unexpected results and errors.*

Distributed Transactions

A distributed transaction is a transaction that updates more than one data source or that updates a local UniVerse database and one or more data sources. Be careful when you use distributed transactions. The UniVerse transaction manager does not support the two-phase commit protocol that ensures that all operations are either committed or rolled back properly. If a COMMIT fails, the systems involved may be out of sync, since the local part of the COMMIT can succeed even though the remote part fails.

If your program uses distributed transactions, you must ensure that it can recover from a COMMIT failure, or that enough information is available to let you manually restore the databases to a consistent state.

Nested Transactions

UniVerse supports nested transactions—that is, within any transaction you can begin and end one or more subtransactions. Only one transaction is *active* at any time, although a transaction and several subtransactions may *exist* simultaneously. Each nested transaction exists on its own *transaction nesting level*. When no transaction currently exists, the program is at transaction nesting level 0.

An SQLFetch call must be issued at the same, or deeper, transaction nesting level as the SQLExecute or SQLExecDirect call associated with it; it cannot be issued at a higher nesting level. This is because the cursors opened within a transaction are closed when that transaction is committed or rolled back.

To preserve locking integrity, an **SQLFetch** call must be issued at the same transaction isolation level as the corresponding **SQLExecute** or **SQLExecDirect** call. If the nested transaction were running at an isolation level higher than the isolation level of the parent transaction, the **SQLFetch** would be executed in a different transaction environment from that of the **SQLExecute** call. If such a transaction were rolled back, it would release all of the locks acquired to support the **SQLFetch**. If an **SQLFetch** call is issued at a different transaction isolation level from the **SQLExecute** or **SQLExecDirect** call, SQL.ERROR is returned, and SQLSTATE is set to S1000.

UniVerse Data Sources

When the SQL Client Interface connects to a UniVerse data source, the application must maintain a correspondence between the transaction nesting level on the client and the transaction nesting level on the server. All connections and disconnections must be established outside of a transaction (at transaction nesting level 0) on the client.

Data definition statements (CREATE SCHEMA, CREATE TABLE, CREATE VIEW, CREATE INDEX, CREATE TRIGGER, ALTER TABLE, DROP SCHEMA, DROP TABLE, DROP VIEW, DROP INDEX, DROP TRIGGER, GRANT, and REVOKE) are not allowed within a transaction. All data definition statements executed by **SQLExecute** and **SQLExecDirect** must be issued at transaction nesting level 0 on the client.

ODBC Data Sources

When the SQL Client Interface connects to an ODBC data source, all **SQLExecute** and **SQLExecDirect** calls must be issued either outside of a transaction (at transaction nesting level 0) or at transaction nesting level 1 on the client. If one of these functions is issued at a transaction nesting level higher than level 1, SQL.ERROR is returned and SQLSTATE is set to IM983.

Detecting Errors

Any SQL Client Interface function call can generate errors. Use the SQLError call after any function call for which the returned status indicates an error condition.

ODBC data sources generally return a consistent error for the same error condition.

Each DBMS returns its own error code in response to SQL statements that generate errors. The same SQL statement generating the same type of error usually has a different error code, depending on the particular data source DBMS. For example, when you try to drop a nonexistent table, each DBMS generates a different native error code.

You can use the MAPERROR parameter in the configuration file to map DBMS error codes to a common SQLSTATE value. This makes it easier to write applications that are portable among supported DBMSs. For example, suppose an SQL statement returns an illegal option error code of 950999 from UniVerse, 1234 from ORACLE and –777 from INFORMIX-OnLine. Your program can test for this error condition by checking for an SQLSTATE value of S1009, regardless of which DBMS it is connected to, if the following MAPERROR statements are in the *uvodbc.config* file:

MAPERROR = S1009 = 950999(for UniVerse data sources)

The SQL Client Interface provides default mappings for the following common error types:

SQLSTATE	Error Described
S0001	Base table or view already exists
S0002	Base table not found

SQLSTATE Errors

For more information about error codes, see Appendix C, "Error Codes."

Errors can be detected by the SQL Client Interface, by the UniRPC, by the ODBC driver, or by the data source. See the SQLError function in Chapter 7, "SQL Client Interface Functions," for more information.

UniVerse Error and System Messages

When you are connected to a UniVerse data source, informational messages, such as "Creating column 1" returned by CREATE TABLE, and error conditions, such as "Table already exists" returned by CREATE TABLE, are suppressed.

Error conditions such as syntax errors are returned to the client program with an error code and an SQLSTATE value. Client programs should check the return status of every function call issued, including calls that request the UniVerse server to execute a programmatic SQL statement. The client program should use the **SQLError** call to determine the reason for the error.

SQLSTATE Values

The **SQLError** function provides an SQLSTATE value, a UniVerse error code, and the error message text. The following table shows all possible SQLSTATE values returned by **SQLError**, with examples of a UniVerse error code and text. For a complete list of SQLSTATE values, see Appendix C, "Error Codes."

UniVerse uses SQLSTATE values defined by the ODBC specification as well as a few defined in the SQL-1992 standard. All errors that cannot be matched to a more specific category return error S1000, which means "general error detected by the data source (for example, UniVerse)."

SQLSTATE	Description
21S01	The number of columns inserted does not match the number expected. For example: 950059 Number of columns inserted does not match number required.
21S02	The number of columns selected does not match the number defined in a CREATE VIEW statement. For example: 950415 More explicit column names than column selected.
22005	Data type does not match. For example: 950121 Column "ORDER.NOs" data type does not match insert value.
23000	An integrity constraint was violated. For example: 950110 Column referential integrity violation.

Client Program Error Codes

SQLSTATE	Description
3F000	The schema name is invalid. For example: 950342 Schema INVENTORY does not exist.
40000	The transaction was rolled back. For example: 950604 Fatal error: ISOLATION level cannot be changed during a transaction
40001	An SQL statement with NOWAIT encountered a conflicting lock.
42000	User lacks SQL privilege or operating system permissions for this operation. For example: 950076 Permission needed to insert records in table "INVENTORY".
IA000	Output from the EXPLAIN keyword.
S0001	The table or view already exists. For example: 950458 Table INVENTORY already exists in VOC.
S0002	The table or view was not found. For example: 950455 View EMPLOYEES does not exist.
S0021	The column already exists. For example: Duplicate column name.
S0022	The column was not found. For example: 950418 Table constraint has an undefined column ORDER.NO.
S1000	A general error was detected by the data source (server). For example: 950427 An association column must be multivalued.

Client Program Error Codes (Continued)

Fatal UniVerse errors that use error codes from 050000 through 050020 cause the application program to exit and return to the UniVerse prompt. Fatal UniVerse errors outside this range of error codes return the program to the previous execution environment and set the @SYSTEM.RETURN.CODE variable to –1.

Displaying Environment Variables in RAID

In RAID you can use the *variable*/ command to display environment variables. For example, the following command displays the SQL Client Interface environment variable DBCENV:

```
::DBCENV/
  This is an ODBC Environment variable.
  It has 2 connection environment(s).
```

The next two commands display two connection environment variables, ODBC1 and ODBC2:

```
::ODBC1/
  This is an ODBC connection environment variable.
  It is connected and has 1 statement environment(s).
::ODBC2/
  This is an ODBC connection environment variable.
  It is not connected.
```

The next command displays the SQL statement environment variable STMTENV:

```
::STMTENV/
  This is an ODBC statement environment variable.
```

If an SQL statement is attached to the statement environment, the *variable*/ command displays it. For example:

```
The SQL statement is select empnum, empname, grade, city
   from sqlcostaff
```

Chapter

5

Calling and Executing Procedures

What Can You Call as a UniVerse Procedure?. 5-3
Processing UniVerse Procedure Results. 5-5
    Print Result Set 5-5
    Affected-Row Count 5-6
Processing Errors from UniVerse Procedures 5-7
Calling and Executing ODBC Procedures 5-8

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*]…]

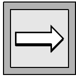
For example, to obtain a listing of the first three records in MYFILE, call the UniVerse LIST command by executing:

```
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** *(etc.) 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.
```

Calling and Executing ODBC Procedures

A BASIC client program that is connected to an ODBC data source can call and execute procedures stored on the server, provided that the ODBC driver and the server's database support a procedure call mechanism. The standard ODBC grammar for a procedure call statement is:

{ call *procedure* [([*parameter* [,*parameter*] …])] }

As when calling UniVerse procedures, you must execute (not simply prepare) a call statement before you can successfully use any of the following functions to inquire about the procedure's results:

- **SQLNumResultCols**
- **SQLColAttributes**
- SQLRowCount

Chapter

6

How to Write a UniVerse Procedure

Using UniVerse Paragraphs, Commands, and Procs as Procedures . . . 6-3
Writing UniVerse BASIC Procedures 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 6-14
Fetching Rows and Closing @HSTMT Within a Procedure 6-15
Hints for Debugging a Procedure 6-15

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.

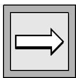
Using UniVerse Paragraphs, Commands, and Procs as Procedures

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	T.RDLBL
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	VI
MAIL	SP.TAPE	

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

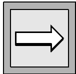
The SUBROUTINE statement at the beginning of a UniVerse BASIC subroutine procedure determines how many input and output parameters it requires. The calling client application program must supply the same number of parameters (or parameter markers for output parameters) in the same order as they are expected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

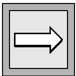
This section discusses how the programming of a UniVerse BASIC procedure determines which type of SQL result it produces.

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 multi-column 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 = [Ardent][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 ClearDiagnostics, 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 =
[Ardent][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, 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.

Chapter

7

SQL Client Interface Functions

Variable Names 7-5
Return Values 7-6
Error Codes 7-7
ClearDiagnostics 7-8
GetDiagnostics 7-9
SetDiagnostics 7-10
SQLAllocConnect 7-12
SQLAllocEnv 7-14
SQLAllocStmt 7-16
SQLBindCol 7-18
SQLBindParameter 7-21
SQLCancel 7-25
SQLColAttributes 7-27
SQLColumns 7-33
SQLConnect 7-36
SQLDescribeCol 7-38
SQLDisconnect 7-40
SQLError 7-42
SQLExecDirect 7-46
SQLExecute 7-50
SQLFetch 7-53
SQLFreeConnect 7-55
SQLFreeEnv 7-56
SQLFreeStmt 7-57
SQLGetInfo 7-59
SQLGetTypeInfo 7-65
SQLNumParams 7-69

SQLNumResultCols 7-71
SQLParamOptions 7-73
SQLPrepare 7-77
SQLRowCount 7-80
SQLSetConnectOption 7-82
SQLSetParam 7-88
SQLSpecialColumns. 7-89
SQLStatistics 7-94
SQLTables 7-99
SQLTransact 7-102

This chapter describes the SQL Client Interface functions in alphabetical order. The following table lists the SQL Client Interface functions according to how they are used.

Category	Function
Initializing	SQLAllocEnv
	SQLAllocConnect
	SQLSetConnectOption
	SQLConnect
	SQLAllocStmt
	SQLPrepare
Exchanging data	SQLTransact
	SQLBindParameter
	SQLSetParam
	SQLExecute
	SQLExecDirect
	SQLRowCount
	SQLNumResultCols
	SQLDescribeCol
	SQLColAttributes
	SQLBindCol
	SQLFetch
	SQLGetInfo
	SQLGetTypeInfo
	SQLNumParams
	SQLParamOptions
	SQLTables

Functions and Their Uses

Category	Function
Exchanging data	SQLColumns
	SQLSpecialColumns
	SQLStatistics
Processing errors	ClearDiagnostics
	GetDiagnostics
	SetDiagnostics
	SQLError
Disconnecting	SQLFreeStmt
	SQLCancel
	SQLDisconnect
	SQLFreeConnect
	SQLFreeEnv

Functions and Their Uses

The syntax diagram for each function includes the function name and any applicable input, output, and return variables. For example:

status = **SQLAllocConnect** (*bci.env, connect.env*)

You must call the **SQLAllocEnv** function before you use any other SQL Client Interface function.

Variable Names

In the previous syntax diagram, *status* is a return variable that the function returns upon completion. *bci.env* is an input variable whose value is provided by a previous function call. *connect.env* is an output variable containing a value output by the function. Names of return variables, input variables, and output variables are user-defined.

Return Values

SQL Client Interface functions return a value to the *status* variable. Return values are the following:

Return Value		Meaning
0	SQL.SUCCESS	Function completed successfully.
1	SQL.SUCCESS.WITH.INFO	Function completed successfully with a possible nonfatal error. Your program can call SQLError to get information about the error.
–1	SQL.ERROR	Function failed. Your program can call **SQLError** to get information about the error.
–2	SQL.INVALID.HANDLE	Function failed because the environment, connection, or SQL statement variable is invalid.
100	SQL.NO.DATA.FOUND	All rows from the result set were fetched (SQLFetch), or no error to report (**SQLError**).

Status Return Values

Error Codes

Any SQL Client Interface function call can generate errors. Use the SQLError function after any other function call for which the returned status indicates an error condition. For a list of SQL Client Interface error codes, see Appendix C, "Error Codes."

ClearDiagnostics

ClearDiagnostics clears diagnostics from the SQL diagnostics area.

Syntax

status = **ClearDiagnostics ( )**

Return Values

0 Success

1 Success: no errors to clear

Description

Use **ClearDiagnostics** in a procedure to clear any diagnostics from the SQL diagnostics area associated with @HSTMT. This removes any errors set by the procedure, allowing the procedure to return SQL.SUCCESS and a result to the calling program. No warning or error text is returned.

GetDiagnostics

GetDiagnostics returns the current warning or error text from the SQL diagnostics area.

Syntax

error = **GetDiagnostics ( )**

Return Value

Return Value	Description
error	The text of the pending error. If none is set, an empty string is returned.

GetDiagnostics Return Value

Description

Use **GetDiagnostics** in a procedure to return the current error text associated with @HSTMT.This text is the same as what would be returned to the output variable *error* by SQLError. The error text is removed from the SQL diagnostics area.

When an SQL statement that uses @HSTMT returns anything other than SQL.SUCCESS, the diagnostic information is associated with @HSTMT. This information comprises the SQLSTATE, the native error number, and the warning or error text.

You can use **SQLError** and GetDiagnostics to examine and clear this information. If multiple diagnostics are available, you need to call these functions continually until you have examined and cleared all diagnostics.

SetDiagnostics

SetDiagnostics adds an error to the SQL diagnostics area associated with @HSTMT.This text is available to the SQLError and GetDiagnostics functions.

Syntax

status = **SetDiagnostics (***text***)**

Input Variable

Input Variable	Description
text	An expression that evaluates to the error message text.

SetDiagnostics Input Variable

Return Values

Return Value	Description
0	Success
1	Error: text empty or text contains field marks or value marks.
2	Error: diagnostics area full.

SetDiagnostics Return Values

Description

A procedure can use **SetDiagnostics** to return an error message to the SQLExecute or SQLExecDirect function that called the procedure.

On exiting an SQL procedure, if an error has been set in the SQL diagnostics area associated with @HSTMT, the procedure returns SQL.ERROR to the SQLExecute or SQLExecDirect function that called it. The caller can then invoke SQLError or GetDiagnostics to determine the SQLSTATE, native error number, and the error text. The error text is stored in the SQL diagnostics area associated with @HSTMT and remains available until the next programmatic SQL statement is executed.

If **SetDiagnostics** sets multiple errors, they are returned to multiple calls to **SQLError** or **GetDiagnostics** in the order in which they are set.

Example

In the following example, the main program calls an SQL procedure named GivePctRaise to give a 25% raise to employee 12345:

```
STATUS = SQLExecDirect (HSTMT, "call GivePctRaise(12345,25)")
```

The procedure is a BASIC subroutine that limits raises to 20%:

```
SUBROUTINE GivePctRaise (EMPNUM, RAISEPCT)
IF RAISEPCT > 20 THEN SetDiagnostics ('Raise percent of
   ':RAISEPCT:' exceeds limit of 20.'); RETURN
      .
      .
      .
```

The main program checks the status of the called procedure and reports any errors:

```
IF STATUS # SQL.SUCCESS THEN GOSUB SHOW.ERR
   .
   .
   .
SHOW.ERR
PRINT 'Error from server.'
PRINT 'Status is ':STATUS
STATUS = SQLError(SQL.NULL.HENV, SQL.NULL.HDBC, HSTMT,
   SQLSTATE, NATIVE, TEXT)
PRINT 'SQLSTATE is ':SQLSTATE
PRINT 'Native error is ':NATIVE
PRINT 'Error text is: ':TEXT
RETURN
```

SQLAllocConnect

SQLAllocConnect allocates and initializes a connection environment in an SQL Client Interface environment.

Syntax

status = **SQLAllocConnect** (*bci.env, connect.env*)

Input Variable

The following table describes the input variable.

Input Variable	Description
bci.env	SQL Client Interface environment variable returned in an SQLAllocEnv call. For connections to a local UniVerse server, *bci.env* can be @HENV.

SQLAllocConnect Input Variable

Output Variable

The following table describes the output variable.

Output Variable	Description
connect.env	Variable that represents the allocated connection environment.

SQLAllocConnect Output Variable

Return Values

The following table describes the return values.

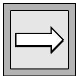
Return Value	Description
0	SQL.SUCCESS
-1	SQL.ERROR
-2	SQL.INVALID.HANDLE

SQLAllocConnect Return Values

Description

Use this function to create a connection environment to connect to a particular data source. One SQL Client Interface environment can have several connection environments, one for each data source. The function stores the internal representation of the connection environment in the *connect.env* variable.

BASIC programs running locally on a UniVerse server (*localuv*) can use the @variable @HDBC to refer directly to the connection environment. They do not need to allocate one.



***Note**: Use the connection environment variable only in SQL Client Interface calls that require it. Using it improperly can cause a run-time error and break communication with the data source.*

SQLAllocEnv

SQLAllocEnv creates an SQL Client Interface environment in which to execute SQL Client Interface calls.

Syntax

status = **SQLAllocEnv (***bci.env***)**

Output Variable

The following table describes the output variable.

Output Variable	Description
bci.env	Variable that represents the allocated SQL Client Interface environment.

SQLAllocEnv Output Variable

Return Values

The following table describes the return values.

Return Value	Description
0	SQL.SUCCESS
-1	SQL.ERROR

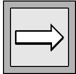
SQLAllocEnv Return Values

Description

Use this function to allocate memory for an SQL Client Interface environment. The function stores the address in the *bci.env* variable. **SQLAllocEnv** must be the first SQL Client Interface call issued in any application.

You can allocate more than one SQL Client Interface environment.

BASIC programs running locally on a UniVerse server (*localuv*) can use the @variable @HENV to refer directly to the SQL Client Interface environment. They do not need to allocate one.



***Note**: Use the SQL Client Interface environment variable only in SQL Client Interface calls that require it. Using it in any other context causes a run-time error or breaks communication with the data source.*

SQLAllocStmt

SQLAllocStmt creates an SQL statement environment in which to execute SQL statements.

Syntax

status = **SQLAllocStmt** (*connect.env, statement.env*)

Input Variable

The following table describes the input variable.

Input Variable	Description
connect.env	Connection environment used in SQLAllocConnect and SQLConnect calls. For connections to a local UniVerse server, *connect.env* can be @HDBC. If you have not established a connection to the data source using *connect.env*, an error is returned to the application.

SQLAllocStmt Input Variable

Output Variable

The following table describes the output variable.

Output Variable	Description
statement.env	Variable that represents an SQL statement environment.

SQLAllocStmt Output Variable

Return Values

The following table describes the return values.

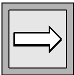
Return Value	Description
0	SQL.SUCCESS
-1	SQL.ERROR
-2	SQL.INVALID.HANDLE

SQLAllocStmt Return Values

Description

Use this function to allocate memory for an SQL statement environment.

BASIC programs running locally on a UniVerse server (*localuv*) can use the @variable @HSTMT to refer directly to the SQL statement environment. They do not need to allocate one.



***Note**: Use the SQL statement environment variable only in SQL Client Interface calls that require it. Using it in any other context causes a run-time error or breaks communication with the data source.*

SQLBindCol

SQLBindCol tells the system where to return column results when an SQLFetch call is issued to retrieve the next row of data.

Syntax

status = **SQLBindCol** (*statement.env, col#, data.type, column*)

Input Variables

The following table describes the input variables.

Input Variable	Description	
statement.env	SQL statement environment of the executed SQL statement. For connections to a local UniVerse server, *statement.env* can be @HSTMT.	
col#	Column number of result data, starting at 1. This value must be from 1 to the number of columns returned in an operation.	
data.type	BASIC data type into which to convert the incoming data. Possible values are the following:	
	SQL.B.CHAR	Character string data.
	SQL.B.BINARY	Bit string (raw) data.
	SQL.B.NUMBER	Numeric data (integer or double).
	SQL.B.DEFAULT	SQL data type determines the BASIC data type. For information about data conversion, see Appendix A, "Data Conversion."
	SQL.B.INTDATE	UniVerse date in internal format.
	SQL.B.INTTIME	UniVerse time in internal format.
	SQL.B.INTTIME	UniVerse time in internal format.
column	Variable that will contain column results obtained with SQLFetch.	

SQLBindCol Input Variables

Return Values

The following table describes the return values.

Return Value	Description
0	SQL.SUCCESS
-1	SQL.ERROR
-2	SQL.INVALID.HANDLE

SQLBindCol Return Values

Description

Use this function to tell the SQL Client Interface where to return the results of an **SQLFetch** call. **SQLBindCol** defines the name of the variable (*column*) to contain column results retrieved by **SQLFetch**, and specifies the data conversion (*data.type)* on the fetched data. **SQLBindCol** has no effect until **SQLFetch** is used.

Normally you call **SQLBindCol** once for each column of data in the result set. When **SQLFetch** is issued, data is moved from the result set at the data source and put into the variables specified in the **SQLBindCol** call, overwriting existing variable contents.

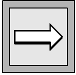
Data is converted from the SQL data type at the data source to the BASIC data type requested by the **SQLBindCol** call, if possible. If data cannot be converted to *data.type*, an error occurs. For information about data conversion types, see Appendix A, "Data Conversion."

Values are returned only for bound columns. Unbound columns are ignored and are not accessible. For example, if a SELECT command returns three columns, but **SQLBindCol** was called for only two columns, data from the third column is not accessible to your program. If you bind more variables than there are columns in the result set, an error is returned. If you bind no columns and an **SQLFetch** is issued, the cursor advances to the next row of results.

You need not use **SQLBindCol** with SQL statements that do not produce result sets.

UniVerse Data Sources

When multivalued data is fetched from a UniVerse data source, it is stored in the bound column as a dynamic array. You can use the SQL.COLUMN.MULTI-VALUED parameter of the SQLColAttributes function to determine whether the column is defined as multivalued or singlevalued. You can then use BASIC dynamic array functions to process the data.



***Note**: You cannot fetch raw multivalued data (that is, data from a column whose data type is SQL.BINARY, SQL.VARBINARY, or SQL.LONGVARBINARY) from a UniVerse data source unless you use dynamic normalization (see the UniVerse SQL Reference).*

SQLBindParameter

SQLBindParameter specifies where to find values for input parameter markers when an SQLExecute or SQLExecDirect call is issued. For output parameter markers, **SQLBindParameter** specifies where to find the return value of a called procedure.

Syntax

status = **SQLBindParameter (***statement.env*, *mrk#*, *data.type*, *sql.type*, *prec*, *scale*, *param* [,*param.type*])

Input Variables

The following table describes the input parameters.

Input Variable	Description	
statement.env	SQL statement environment associated with an SQL statement. For connections to a local UniVerse server, *statement.env* can be @HSTMT.	
mrk#	Number of the parameter marker in the SQL statement this call refers to. Parameter markers in the SQL statement are numbered left to right starting at 1.	
data.type	BASIC data type to bind to the parameter. *data.type* must be one of the following:	
	SQL.B.BASIC	Use with any *sql.type*.
	SQL.B.BINARY	Use only when *sql.type* is SQL.BINARY, SQL.VARBINARY, or SQL.LONGVARBINARY.
	SQL.B.INTDATE	Use only when *sql.type* is SQL.DATE.
	SQL.B.INTTIME	Use only when *sql.type* is SQL.TIME.

SQLBindParameter Input Variables

Input Variable	Description
sql.type	SQL data type to which the BASIC variable is converted. For information about converting BASIC data to SQL data types, see Appendix A, "Converting BASIC Data to SQL Data."
prec	Precision of the parameter, representing the width of the parameter. If *prec* is 0, default values are used based on the extended parameter settings in the *uvodbc.config* file (see Appendix D, "UniVerse Extended Parameters.")
scale	Scale of the parameter, used only when *sql.type* is SQL.DECIMAL or SQL.NUMERIC.
param	Variable that contains the data to use when SQLExecute or SQLExecDirect is called.
param.type	Type of parameter. *param.type* 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 output parameter in a procedure. SQL.PARAM.INPUT.OUTPUT — Use for an input/output parameter in a procedure. If you do not specify *param.type*, SQL.PARAM.INPUT is used.

SQLBindParameter Input Variables (Continued)

Return Values

The following table describes the return values.

Return Value	Description
0	SQL.SUCCESS
-1	SQL.ERROR
-2	SQL.INVALID.HANDLE

SQLBindParameter Return Values

Description

Parameter markers are placeholders in SQL statements. Input parameter markers let a program send user-defined values with the SQL statement when an **SQLExecute** or **SQLExecDirect** call is executed repeatedly. Output parameter markers receive values returned from a called procedure. The placeholder character is ? (question mark). For more information about parameter markers, see Using Parameter Markers in SQL Statements in Chapter 4, "Using the SQL Client Interface."

SQLBindParameter tells the system where to find the variables to substitute for parameter markers in the SQL statement and how to convert the data before sending it to the data source. You need to do only one **SQLBindParameter** for each parameter marker in the SQL statement, no matter how many times the statement is to be executed.

For example, consider the following SQL statement:

```
INSERT INTO T1 VALUES (?,?,?);
```

If you want to load 1000 rows, you need issue only three **SQLBindParameter** calls, one for each question mark.

Normally you specify *data.type* as SQL.B.BASIC. If you specify *sql.type* as SQL.DATE, however, you can specify *data.type* as SQL.B.INTDATE; if you specify *sql.type* as SQL.TIME, you can specify *data.type* as SQL.B.INTTIME. If you specify *sql.type* as SQL.BINARY, SQL.VARBINARY, or SQL.LONGVARBINARY, you can specify *data.type* as SQL.B.BINARY.

If you use SQL.B.INTDATE, the SQL Client Interface assumes the program variable holds a date in UniVerse internal date format and uses the DATEFORM conversion string to convert the internal date to an external format as required by the data source. To set or change the DATEFORM conversion string, see the SQLSetConnectOption function. For details about date and time conversions, see Appendix A, "Data Conversion."

If you specify *sql.type* as SQL.TIME and *data.type* as SQL.B.INTTIME, the SQL Client Interface assumes the program variable holds a time in UniVerse internal time format and does not convert the data.

SQLBindParameter uses the value of *prec* only for the following SQL data types:

SQL.CHAR
SQL.VARCHAR
SQL.LONGVARCHAR
SQL.WCHAR
SQL.WVARCHAR
SQL.WLONGVARCHAR
SQL.BINARY
SQL.VARBINARY
SQL.LONGVARBINARY
SQL.NUMERIC
SQL.DECIMAL

For all other data types, the extended parameters DBLPREC, FLOATPREC, and INTPREC determine the maximum length for strings representing double-precision numbers, floating-point numbers, and integers.

UniVerse Data Sources

The *prec* and *scale* parameters are not used if you are connected to a UniVerse data source.

SQLCancel

SQLCancel cancels the current SQL statement associated with an SQL statement environment and discards any pending results.

Syntax

status = **SQLCancel (***statement.env***)**

Input Variable

The following table describes the input variable.

Input Variable	Description
statement.env	SQL statement environment. For connections to a local UniVerse server, *statement.env* can be @HSTMT.

SQLCancel Input Variable

Return Values

The following table describes the return values.

Return Value	Description
0	SQL.SUCCESS
-1	SQL.ERROR
-2	SQL.INVALID.HANDLE

SQLCancel Return Values

Description

This function is equivalent to the SQLFreeStmt call with the SQL.CLOSE option. It closes any open cursor associated with the SQL statement environment and discards pending results at the data source.

It is good practice to issue **SQLCancel** 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. Issuing **SQLCancel** frees any locks that may be held at the data source.

SQLColAttributes

SQLColAttributes returns information about the columns available in a result set produced by an SQL SELECT statement.

Syntax

status = **SQLColAttributes** (*statement.env, col#, col.attribute, text.var, num.var*)

Input Variables

The following table describes the input variables.

Input Variable	Description
statement.env	SQL statement environment of the executed SQL statement. For connections to a local UniVerse server, *statement.env* can be @HSTMT.
col#	Column number to describe, starting with 1.
col.attribute	Attribute of the column that needs information. *col.attribute* values are listed in the following tables. These values are defined in the ODBC.H file. Appendix E, "The ODBC.H File," lists the contents of the ODBC.H file.

SQLColAttributes Input Variables

Output Variables

The following table describes the output variables.

Output Variable	Description
text.var	Contains column information for attributes returning text data.
num.var	Contains column information for attributes returning numeric data.

SQLColAttributes Output Variables

Return Values

The following table describes the return values.

Return Value	Description
0	SQL.SUCCESS
1	SQL.SUCCESS.WITH.INFO
-1	SQL.ERROR
-2	SQL.INVALID.HANDLE

SQLColAttributes Return Values

Description

Use this function to get information about a column. **SQLColAttributes** returns the specific information requested by the value of *col.attribute*.

Some DBMSs (such as SYBASE) do not make column information available until after the SQL statement is executed. In such cases, issuing an **SQLColAttributes** call before executing the statement produces an error.

The SQL.SUCCESS.WITH.INFO return occurs when you issue the call for a column that contains an unsupported data type or when *text.var* is truncated. The SQL data type returned is SQL.BINARY (–2).

The following table lists the column attributes you can use with both UniVerse and ODBC databases.

Column Attribute	Output	Description
SQL.COLUMN.AUTO.INCREMENT	*num.var*	1 – TRUE if the column values are incremented automatically. 0 – FALSE if the column values are not incremented automatically.
SQL.COLUMN.CASE.SENSITIVE	*num.var*	1 – TRUE for character data. 0 – FALSE for all other data.
SQL.COLUMN.COUNT	*num.var*	Number of columns in result set. The *col#* argument must be a valid column number in the result set.
SQL.COLUMN.DISPLAY.SIZE	*num.var*	Maximum number of characters required to display data from the column.
SQL.COLUMN.LABEL	*text.var*	Column heading.
SQL.COLUMN.LENGTH	*num.var*	Number of bytes transferred by an **SQLFetch** call.
SQL.COLUMN.NAME	*text.var*	Name of specified column.

Column Attributes

Column Attribute	Output	Description
SQL.COLUMN.NULLABLE	*num.var*	Column can contain null values. Can return one of the following: 0 – SQL.NO.NULLS 1 – SQL.NULLABLE 2 – SQL.NULLABLE.UNKNOWN
SQL.COLUMN.PRECISION	*num.var*	Column's precision.
SQL.COLUMN.SCALE	*num.var*	Column's scale.
SQL.COLUMN.SEARCHABLE	*num.var*	Always returns 3, SQL.SEARCHABLE.
SQL.COLUMN.TABLE.NAME	text.var	Name of the table to which the column belongs. If the column is an expression, an empty string is returned.
SQL.COLUMN.TYPE	*num.var*	Number representing the SQL type of this column. See Appendix E, "The ODBC.H File," for data type definitions. See Appendix A, "Data Conversion," for a list of data types.
SQL.COLUMN.TYPE.NAME	*text.var*	Data type name for column, specific to the data source.
SQL.COLUMN.UNSIGNED	*num.var*	1 – TRUE for nonnumeric data types. 0 – FALSE for all other data types.
SQL.COLUMN.UPDATABLE	*num.var*	As of Release 9, any expressions or computed columns return SQL.ATTR.READONLY, and stored data columns return SQL.ATTR.WRITE.

Column Attributes (Continued)

If you are connected to an ODBC database, SQL.COLUMN.NULLABLE always returns SQL.NULLABLE.UNKNOWN.

UniVerse Data Sources

The following table lists the column attributes you can use only with UniVerse databases.

Column Attribute	Output	Description
SQL.COLUMN.CONVERSION	text.var	UniVerse conversion code.
SQL.COLUMN.FORMAT	text.var	UniVerse format code.
SQL.COLUMN.MULTIVALUED	*num.var*	1 – TRUE if column is multivalued. 0 – FALSE if column is singlevalued.
SQL.COLUMN.PRINT.RESULT	num.var	1– TRUE if column is a one-column PRINT result set from a called procedure. 0 – FALSE if column contains no PRINT output from a called procedure. See "Processing UniVerse Procedure Results" on page 5 for details.

UniVerse Column Attributes

When you are connected to an ODBC data source, calling **SQLColAttributes** with one of the UniVerse-only column attributes returns a status of SQL.ERROR with SQLSTATE set to S1091.

ODBC Data Sources

The following table lists the column attributes you can use only with ODBC databases.

Column Attributes	Output	Description
SQL.COLUMN.MONEY	num.var	1 – TRUE if column is money data type. 0 – FALSE if column is not money data type.
SQL.COLUMN.OWNER.NAME	text.var	Owner of the table containing the column.
SQL.COLUMN.QUALIFIER.NAME	text.var	Qualifier of the table containing the column.

ODBC Column Attributes

SQLColumns

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

Syntax

status = **SQLColumns** (*statement.env, schema, owner, tablename, columnname*)

Input Variables

The following table describes the input variables.

Input Variable	Description
statement.env	SQL statement environment.
schema	Schema name search pattern.
owner	Table owner number search pattern.
tablename	Table name search pattern.
columnname	Column name search pattern.

SQLColumns Input Variables

Description

This function returns a result set in *statement.env* as a cursor of 12 columns describing those columns found by the search pattern. 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.

The result set contains 12 columns:

Column Name	Data Type
TABLE.SCHEMA	VARCHAR(128)
OWNER	INTEGER
TABLE.NAME	VARCHAR(128)
COLUMN.NAME	VARCHAR(128)
DATA.TYPE	SMALLINT
TYPE.NAME	VARCHAR(128)
NUMERIC.PRECISION	INTEGER
CHAR.MAX.LENGTH	INTEGER
NUMERIC.SCALE	SMALLINT
NUMERIC.PREC.RADIX	SMALLINT
NULLABLE	SMALLINT
REMARKS	VARCHAR(254)

Result Set Columns

The application is responsible for binding variables for the output columns and fetching the results using **SQLFetch**.

Return Values

Return Value	Description
0	SQL.SUCCESS
1	SQL.SUCCESS.WITH.INFO
-1	SQL.ERROR
-2	SQL.INVALID.HANDLE

SQLColumns Return Values

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.
S1008	Cancelled. Execution of the statement was stopped by an SQLCancel call.
S1010	Function sequence error. The *statement.env* 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 be caused by a variety of reasons. The native error code returned by the SQLError call indicates the specific UniVerse error that occurred.

SQLColumns SQLSTATE Values

SQLConnect

SQLConnect connects to a data source.

Syntax

status = **SQLConnect (***connect.env, data.source, logon1, logon2***)**

Input Variables

The following table describes input variables.

Input Variable	Description
connect.env	Connection environment assigned in a previous SQLAllocConnect. For connections to a local UniVerse server, *connect.env* can be @HDBC.
data.source	Data source name. For UniVerse data sources, this is the name of a valid data source defined in the *uvodbc.config* file. For ODBC data sources, this is the name of a data source specified by the data source management program you are using.
logon1	For a local UniVerse server, *logon1* is ignored. ■ For a remote UniVerse server, *logon1* is the name of the UniVerse account to connect to. The name can be the full path of the account directory, a schema name, or an account name as defined in the UV.ACCOUNT file. ■ For ODBC data sources, this is typically the user name for the remote database or operating system.
logon2	For local and remote UniVerse servers, *logon2* is ignored. For ODBC data sources, this is typically the password for the remote database or operating system.

SQLConnect Input Variables

For the specific information required for *logon1* and *logon2* when connecting to ODBC data sources, see the configuration for the specific driver used.

Return Values

The following table describes the return values.

Return Value	Description
0	SQL.SUCCESS
1	SQL.SUCCESS.WITH.INFO
-1	SQL.ERROR
-2	SQL.INVALID.HANDLE

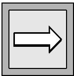
SQLConnect Input Variables

Description

Use this function to connect to the data source specified by *data.source*. UniVerse data sources must be defined in the configuration file. Use the *logon1* and *logon2* parameters to log on to the DBMS specified by *data.source*.

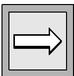
You cannot use **SQLConnect** within a transaction. An **SQLConnect** call issued within a transaction returns SQL.ERROR, and sets SQLSTATE to 25000, indicating that the **SQLConnect** function is illegal within a transaction.

As of Release 9.4.1, if you are connecting to a UniVerse server running with NLS enabled, you can use the SQLSetConnectOption call to specify the NLS locale information (SQL_NLSLOCALE, and so forth).



***Note**: 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. See "Connecting to a UniVerse Server with NLS Enabled" in Chapter 4, "Using the SQL Client Interface," for more information.*

A connection is established when the data source validates the user name and authorization.



***Note**: When connecting to a remote UniVerse server, before issuing* **SQLConnect**, *use SQLSetConnectOption calls to specify the user name and password for logging on to the server's operating system. If you do not do this, the connection fails.*

If you are connecting to a local UniVerse server (localuv), you connect directly to the UniVerse schema or account you are currently logged on to. You need not use **SQLSetConnectOption** *to specify the user name and password.*

SQLDescribeCol

SQLDescribeCol returns information about one column of a result set produced by an SQL SELECT statement.

Syntax

status = **SQLDescribeCol** (*statement.env, col#, col.name, sql.type, prec, scale, null*)

Input Variables

The following table describes the input variables.

Input Variable	Description
statement.env	SQL statement environment of the executed SQL statement. For connections to a local UniVerse server, *statement.env* can be @HSTMT.
col#	Column number to describe, starting with 1.

SQLDescribeCol Input Variables

Output Variables

The following table describes the output variables.

Output Variable	Description
col.name	Column name.
sql.type	SQL data type of the column, a numeric code defined in the ODBC.H file. See Appendix E, "The ODBC.H File," for more information.

SQLDescribeCol Output Variables

Output Variable	Description
prec	Precision of the column, or –1 if precision is unknown.
scale	Scale of the column, or –1 if scale is unknown.
null	One of the following: 0 SQL.NO.NULLS: field cannot contain NULL. 1 SQL.NULLABLE: field can contain NULL. 2 SQL.NULLABLE.UNKNOWN: not known whether field can contain NULL.

SQLDescribeCol Output Variables (Continued)

Return Values

The following table describes the return values.

Return Value	Description
0	SQL.SUCCESS
1	SQL.SUCCESS.WITH.INFO
-1	SQL.ERROR
-2	SQL.INVALID.HANDLE

SQLDescribeCol Return Values

Description

Use this function to get information about the column described by *col#*.

The SQL.SUCCESS.WITH.INFO return occurs when you issue the call for a column that contains an unsupported data type, or if *col.name* is truncated. The SQL data type returned is SQL.BINARY (–2).

SQLDisconnect

SQLDisconnect disconnects a connection environment from a data source.

Syntax

status = **SQLDisconnect** (*connect.env*)

Input Variable

The following table describes the input variable.

Input Variable	Description
connect.env	Connection environment.

SQLDisconnect Input Variable

Return Values

The following table describes the return values.

Return Value	Description
0	SQL.SUCCESS
1	SQL.SUCCESS.WITH.INFO
-1	SQL.ERROR
-2	SQL.INVALID.HANDLE

SQLDisconnect Return Values

Description

You cannot use **SQLDisconnect** within a transaction. An **SQLDisconnect** call issued within a transaction returns SQL.ERROR, and sets SQLSTATE to 25000. You must commit or roll back active transactions before disconnecting, and you must be in autocommit mode. If there is no active transaction, **SQLDisconnect** frees all SQL statement environments owned by this connection before disconnecting.

SQLDisconnect returns SQL.SUCCESS.WITH.INFO if an error occurs but the disconnect succeeds.

SQLError

SQLError returns error status information about one of the three environments you use.

Syntax

status = **SQLError (***bci.env, connect.env, statement.env, sqlstate, dbms.code, error***)**

Input Variables

The following table describes the input variables.

Input Variable	Description
bci.env	SQL Client Interface environment or the constant SQL.NULL.HENV. For connections to a local UniVerse server, *bci.env* can be @HENV.
connect.env	Connection environment or the constant SQL.NULL.HDBC. For connections to a local UniVerse server, *connect.env* can be @HDBC.
statement.env	SQL statement environment or the constant SQL.NULL.HSTMT. For connections to a local UniVerse server, *statement.env* can be @HSTMT.

SQLError Input Variables

Output Variables

Output Variable	Description
sqlstate	SQLSTATE code. This code describes the SQL Client Interface Client error associated with the environment passed. *sqlstate* is always a five-character string. For a list of SQLSTATE codes and their meanings, see Appendix C, "Error Codes."
dbms.code	Error code specific to the data source. *dbms.code* contains an integer error code from the data source. If *dbms.code* is 0, the error was detected by the SQL Client Interface. For the meanings of specific error codes, see the documentation provided for the data source.
error	Text describing the error in more detail.

SQLError Output Variables

Return Values

The following table describes the return values.

Return Value	Description
0	SQL.SUCCESS
1	SQL.SUCCESS.WITH.INFO
-1	SQL.ERROR
-2	SQL.INVALID.HANDLE
100	SQL.NO.DATA.FOUND

SQLError Return Values

Description

Use **SQLError** when a function returns a status value other than SQL.SUCCESS or SQL.INVALID.HANDLE. **SQLError** returns a value in *sqlstate* when the SQL Client Interface detects an error condition. The *dbms.code* field contains information from the data source that identifies the error.

Each environment type maintains its own error status. **SQLError** returns errors for the rightmost nonnull environment. For example, to get errors associated with a connection environment, specify input variables and constants in the following order:

bci.env, *connect.env*, SQL.NULL.HSTMT

To get errors associated with a particular SQL statement environment, specify the following:

bci.env, *connect.env*, *statement.env*

If all arguments are null, **SQLError** returns a status of SQL.NO.DATA.FOUND and sets SQLSTATE to 00000.

Since multiple errors can be returned for a variable, you should call **SQLError** until it returns a status of SQL.NO.DATA.FOUND. This ensures that all errors are reported.

UniVerse Data Sources

When a program is connected to a UniVerse server, errors can be detected by the SQL Client Interface, by the UniRPC middleware, or by the UniVerse server. When the error is returned, the source of the error is indicated by bracketed items in the *error* output variable, as shown in the following examples.

Errors detected by the SQL Client Interface software:

```
[IBM][SQL Client] An illegal configuration option was found
```

For information about errors detected by the SQL Client Interface, see Appendix C, "Error Codes."

Errors detected by the UniRPC middleware:

```
[IBM][SQL Client][RPC] Connect error, subcode: . . .
```

Errors detected by the UniVerse server:

```
[IBM][SQL Client][UNIVERSE] Universe/SQL: Table ORDERS does not
exist.
```

ODBC Data Sources

When a program is connected to an ODBC server, errors can be detected by the SQL Client Interface, by the ODBC driver, or by the data source. When the error is returned, the source of the error is indicated by bracketed items in the *error* output variable, as shown in the following examples.

Errors detected by the SQL Client Interface software:

```
[IBM][SQL Client] An illegal configuration option was found
```

For information about errors detected by the SQL Client Interface, see Appendix C, "Error Codes."

Errors detected by the ODBC driver:

```
SQLConnect error:   Status = -1   SQLState = S1000   Natcode =
9352

[ODBC] [INTERSOLV][ODBC Oracle driver][Oracle]ORA-09352: Windows
32-bit

Two-Task driver unable to spawn new ORACLE task
```

For information about errors detected by the ODBC driver manager, see the *Microsoft ODBC 2.0 Programmer's Reference and SDK Guide*.

Errors detected by the data source:

```
[IBM][SQL Client][INFORMIX] Database not found or no system
   permissions.
```

For information about errors detected by the data source, see the documentation provided for the DBMS running on the data source.

SQLExecDirect

SQLExecDirect accepts an SQL statement or procedure call and delivers it to the data source for execution. It uses the current values of any SQL statement parameter markers.

Syntax

status = **SQLExecDirect (***statement.env, statement***)**

Input Variables

The following table describes the input variables.

Input Variable	Description	
statement.env	SQL statement environment from a previous SQLAllocStmt. For connections to a local UniVerse server, *statement.env* can be @HSTMT.	
statement	Either an SQL statement or a call to an SQL procedure, to be executed at the data source. If you are connected to a UniVerse server, it treats the SQL statement case-sensitively; all keywords must be in uppercase letters. If you are connected to an ODBC data source, it may treat identifiers and keywords in the SQL statement case-sensitively. To call an SQL procedure, use one of the following syntaxes: [{] CALL *procedure* [([*parameter* [, *parameter*] …])] [}] CALL *procedure* [*argument* [*argument*] …] If you are connected to an ODBC data source, use the first syntax and enclose the entire **call** statement in braces.	
	procedure	Name of the procedure. If the procedure name contains characters other than alphabetic or numeric, enclose the name in double quotation marks. To embed a single double quotation mark in the procedure name, use two consecutive double quotation marks.
	parameter	Either a literal value or a parameter marker that marks where to insert values to send to or receive from the data source. Programmatic SQL uses a ? (question mark) as a parameter marker. 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. For an ODBC data source, parameters should be of the same data type as the procedure requires.
	argument	Any valid keyword, literal, or other token you can use in a UniVerse command line.

SQLExecDirect Input Variables

Return Values

The following table describes the return values.

Return Value	Description
0	SQL.SUCCESS
1	SQL.SUCCESS.WITH.INFO
-1	SQL.ERROR
-2	SQL.INVALID.HANDLE

SQLExecDirect Return Values

Description

SQLExecDirect differs from **SQLExecute** in that it does not require a call to SQLPrepare. **SQLExecDirect** prepares the SQL statement or procedure call implicitly. Use **SQLExecDirect** when you do not need to execute the same SQL statement or procedure repeatedly.

You can use parameter markers in the SQL statement or procedure call as long as you have resolved each marker with an SQLBindParameter call. For information about parameter markers, see "Using Parameter Markers in SQL Statements" in Chapter 4, "Using the SQL Client Interface.".

After an **SQLExecDirect** call you can use SQLNumResultCols, SQLDescribeCol, SQLRowCount, or SQLColAttributes to get information about the resulting columns. You can use **SQLNumResultCols** to determine if the SQL statement or procedure call created a result set.

If the executed SQL statement or procedure produces a set of results, you must use an SQLFreeStmt call with the SQL.CLOSE option before you execute another SQL statement or procedure call using the same SQL statement environment. The SQL.CLOSE option cancels any pending results still waiting at the data source.

Your application programs should not try to issue transaction control statements directly to the data source (for instance, by issuing a COMMIT statement with **SQLExecDirect** or SQLPrepare). Programs should use only BASIC transaction control statements. The SQL Client Interface issues the correct combination of transaction control statements and middleware transaction control function calls that are appropriate for the DBMS you are using. Trying to use **SQLExecDirect** to execute explicit transaction control statements on ODBC data sources can cause unexpected results and errors.

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.

UniVerse Data Sources

If your application is connected to a UniVerse data source, it must execute data definition statements (CREATE SCHEMA, CREATE TABLE, CREATE VIEW, CREATE INDEX, ALTER TABLE, DROP SCHEMA, DROP TABLE, DROP VIEW, DROP INDEX, GRANT, and REVOKE) outside a transaction (at transaction level 0). If a DDL statement is executed within a transaction, **SQLExecDirect** returns SQL.ERROR and sets SQLSTATE to S1000, indicating that DDL statements are illegal in a transaction.

SQL statements can refer to UniVerse files as well as to SQL tables.

ODBC Data Sources

If your application is connected to an ODBC data source, it must issue **SQLExecDirect** calls either outside a transaction (transaction level 0) or at transaction level 1. An **SQLExecDirect** call issued within a nested transaction returns SQL.ERROR and sets SQLSTATE to IM983, indicating that the call is not allowed at the current nesting level.

If you execute a stored procedure or enter a command batch with multiple SELECT statements, the results of only the first SELECT statement are returned.

SQLExecute

SQLExecute tells the data source to execute a prepared SQL statement or a called procedure, using the current values of any parameter markers used in the statement. Using **SQLExecute** with an SQLBindParameter call is the most efficient way to execute a statement repeatedly, since the statement does not have to be parsed by the data source each time it is issued.

Syntax

status = **SQLExecute (***statement.env***)**

Input Variable

The following table describes the input variable.

Input Variable	Description
statement.env	SQL statement environment associated with a prepared SQL statement. For connections to a local UniVerse server, *statement.env* can be @HSTMT.

SQLExecute Input Variable

Return Values

The following table describes the return values.

Return Value	Description
0	SQL.SUCCESS
1	SQL.SUCCESS.WITH.INFO
-1	SQL.ERROR
-2	SQL.INVALID.HANDLE

SQLExecute Return Values

Description

Use this function to repeatedly execute an SQL statement, using different values for parameter markers. You must use an SQLPrepare call 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 call for each marker in the SQL statement before you use **SQLExecute**. After you load the parameter marker variables with data to send to the data source, you can issue the **SQLExecute** call. 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 **SQLBindParameter** call for the parameter markers. See Appendix A, "Data Conversion," for details.

If the SQL statement executed produces a set of results, you must use an SQLFreeStmt call with the SQL.CLOSE option before you execute another SQL statement using the same SQL statement environment. The SQL.CLOSE option cancels any pending results still waiting at the data source.

Your application programs should not try to issue transaction control statements directly to the data source (for instance, by issuing a COMMIT statement with SQLExecDirect or SQLPrepare). Programs should use only BASIC transaction control statements. The SQL Client Interface issues the correct combination of transaction control statements and middleware transaction control function calls that are appropriate for the DBMS you are using. Trying to use **SQLExecute** to execute explicit transaction control statements on ODBC data sources can cause unexpected results and errors.

UniVerse Data Sources

If your application is connected to a UniVerse data source, it must execute data definition statements (CREATE SCHEMA, CREATE TABLE, CREATE VIEW, CREATE INDEX, ALTER TABLE, DROP SCHEMA, DROP TABLE, DROP VIEW, DROP INDEX, GRANT, and REVOKE) outside a transaction (at transaction level 0). If a DDL statement is executed within a transaction, **SQLExecute** returns SQL.ERROR and sets SQLSTATE to S1000, indicating that DDL statements are illegal in a transaction.

ODBC Data Sources

If your application is connecting to an ODBC data source, it must issue **SQLExecute** calls either outside a transaction (transaction level 0) or at transaction level 1. An **SQLExecute** call issued within a nested transaction returns SQL.ERROR and SQLSTATE is set to IM983, indicating that the call is not allowed at the current nesting level.

SQLFetch

SQLFetch returns the next row of data from the result set pending at the data source.

Syntax

status = **SQLFetch (***statement.env***)**

Input Variable

The following table describes the input variable.

Input Variable	Description
statement.env	SQL statement environment of the executed SQL statement. For connections to a local UniVerse server, *statement.env* can be @HSTMT.

SQLFetch Input Variable

Return Values

The following table describes the return values.

Return Value	Description
0	SQL.SUCCESS
-1	SQL.ERROR
-2	SQL.INVALID.HANDLE
1	SQL.SUCCESS.WITH.INFO
100	SQL.NO.DATA.FOUND

SQLFetch Return Values

Description

Use this function to retrieve the next row's column values from the result set at the data source and put them into the variables specified with SQLBindCol. **SQLFetch** performs any required data conversions (see Appendix A, "Data Conversion," for details).

SQLFetch returns SQL.SUCCESS.WITH.INFO if numeric data is truncated or rounded when converting SQL values to UniVerse values.

SQLFetch logically advances the cursor to the next row in the result set. Unbound columns are ignored and are not available to the application. When no more rows are available, **SQLFetch** returns a status of 100.

Your application must issue an **SQLFetch** call at the same transaction nesting level (or deeper) as the corresponding SQLExecDirect or SQLExecute call. Also, an **SQLFetch** call must be executed at the same transaction isolation level as the SELECT statement that generates the data. If it does not, **SQLFetch** returns SQL.ERROR and sets SQLSTATE to S1000.

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

SQLFreeConnect

SQLFreeConnect releases a connection environment and its resources.

Syntax

status = **SQLFreeConnect** (*connect.env*)

Input Variable

The following table describes the input variable.

Input Variable	Description
connect.env	Connection environment.

SQLFreeConnect Input Variable

Return Values

The following table describes the return values.

Return Value	Description
0	SQL.SUCCESS
-1	SQL.ERROR
-2	SQL.INVALID.HANDLE

SQLFreeConnect Return Values

Description

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

SQLFreeEnv

SQLFreeEnv releases an SQL Client Interface environment and its resources.

Syntax

status = **SQLFreeEnv (***bci.env***)**

Input Variable

The following table describes the input variable.

Input Variable	Description
bci.env	SQL Client Interface environment.

SQLFreeEnv Input Variable

Return Values

The following table describes the return values.

Return Value	Description
0	SQL.SUCCESS
-1	SQL.ERROR
-2	SQL.INVALID.HANDLE

SQLFreeEnv

Description

You must use **SQLFreeConnect** to release all connection environments attached to the SQL Client Interface environment before you release the SQL Client Interface environment with **SQLFreeEnv**, otherwise an error is returned.

SQLFreeStmt

SQLFreeStmt frees some or all resources associated with an SQL statement environment.

Syntax

status = **SQLFreeStmt (***statement.env, option***)**

Input Variable

The following table describes the input variable.

Input Variable	Description
statement.env	SQL statement environment. For connections to a local UniVerse server, *statement.env* can be @HSTMT.

SQLFreeStmt Input Variable

option is one of the following:

Option	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 calls.
SQL.UNBIND	Releases all bound column variables defined in **SQLBindCol** for this SQL statement environment.
SQL.RESET.PARAMS	Releases all parameter marker variables set by **SQLBindParameter** for this SQL statement environment.
SQL.DROP	Releases the SQL statement environment. This option terminates all access to the SQL statement environment. SQL.DROP also closes cursors, discards pending results, unbinds columns, and resets parameter marker variables.

SQLFreeStmt Options

Options are defined in the ODBC.H file. See Appendix E, "The ODBC.H File," for more information.

Return Values

The following table describes the return values.

Return Value	Description
0	SQL.SUCCESS
-1	SQL.ERROR
-2	SQL.INVALID.HANDLE

SQLFreeStmt Return Values

Description

Use this function at the end of processing to free resources used by the SQL statement environment.

If your program uses the same SQL statement environment to execute different SQL statements, use **SQLFreeStmt** with the SQL.DROP option, then use SQLAllocStmt to reallocate a new SQL statement environment. This unbinds all bound columns and resets all parameter marker variables.

It is good practice to issue **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. Issuing **SQLFreeStmt** with the SQL.CLOSE option frees any locks that may be held at the data source.

SQLGetInfo

SQLGetInfo returns general information about the ODBC driver and the data source.

Syntax

status = **SQLGetInfo (***connect.env, info.type, info.value***)**

Input Variables

The following table describes the input variables.

Input Variable	Description
connect.env	Connection environment.
info.type	The specific information requested. For a list of values, see the following tables.

SQLGetInfo Input Variables

Output Variable

The following table describes the output variable.

Output Variable	Description
info.value	The information returned by **SQLGetInfo**.

SQLGetInfo Output Variable

Description

This function supports all of the possible requests for information defined in the ODBC 2.0 specification. The *#define*s for *info.type* are contained in the ODBC.H include file. In addition, **SQLGetInfo** also returns NLS information from UniVerse data sources.

ODBC info.type Values

The following table lists the valid ODBC values for *info.type*. For more detailed information about information types, see the *Microsoft ODBC 2.0 Programmer's Reference and SDK Guide*.

Driver Information	
SQL.ACTIVE.CONNECTIONS	SQL.DRIVER.VER
SQL.ACTIVE.STATEMENTS	SQL.FETCH.DIRECTION
SQL.DATA.SOURCE.NAME	SQL.FILE.USAGE
SQL.DRIVER.HDBC	SQL.GETDATA.EXTENSIONS
SQL.DRIVER.HENV	SQL.LOCK.TYPES
SQL.DRIVER.HLIB	SQL.ODBC.API.CONFORMANCE
SQL.DRIVER.HSTMT	SQL.ODBC.SAG.CLI.CONFORMANCE
SQL.DRIVER.NAME	SQL.ODBC.VER
SQL.DRIVER.ODBC.VER	SQL.POS.OPERATIONS
SQL.ROW.UPDATES	SQL.SERVER.NAME
SQL.SEARCH.PATTERN.ESCAPE	
DBMS Product Information	
SQL.DATABASE.NAME	SQL.DBMS.VER
SQL.DBMS.NAME	
Data Source Information	
SQL.ACCESSIBLE.PROCEDURES	SQL.OWNER.TERM
SQL.ACCESSIBLE.TABLES	SQL.PROCEDURE.TERM
SQL.BOOKMARK.PERSISTENCE	SQL.QUALIFIER.TERM

ODBC *info.type* Values

SQL.CONCAT.NULL.BEHAVIOR	SQL.SCROLL.CONCURRENCY
SQL.CURSOR.COMMIT.BEHAVIOR	SQL.SCROLL.OPTIONS
SQL.DATA.SOURCE.READ.ONLY	SQL.STATIC.SENSITIVITY
SQL.DEFAULT.TXN.ISOLATION	SQL.TABLE.TERM
SQL.MULT.RESULT.SETS	SQL.TXN.CAPABLE
SQL.MULTIPLE.ACTIVE.TXN	SQL.TXN.ISOLATION.OPTION
SQL.NEED.LONG.DATA.LEN	SQL.USER.NAME
SQL.NULL.COLLATION	
Supported SQL	
SQL.ALTER.TABLE	SQL.ODBC.SQL.OPT.IEF
SQL.COLUMN.ALIAS	SQL.ORDER.BY.COLUMNS.IN.SELECT
SQL.CORRELATION.NAME	SQL.OUTER.JOINS
SQL.EXPRESSIONS.IN.ORDER.BY	SQL.OWNER.USAGE
SQL.GROUP.BY	SQL.POSITIONED.STATEMENTS
SQL.IDENTIFIER.CASE	SQL.PROCEDURES
SQL.IDENTIFIER.QUOTE.CHAR	SQL.QUALIFIER.LOCATION
SQL.KEYWORDS	SQL.QUALIFIER.NAME.SEPARATOR
SQL.LIKE.ESCAPE.CLAUSE	SQL.QUALIFIER.USAGE
SQL.NON.NULLABLE.COLUMNS	SQL.QUOTED.IDENTIFIER.CASE
SQL.ODBC.SQL.CONFORMANCE	SQL.SPECIAL.CHARACTERS
SQL.SUBQUERIES	SQL.UNION
SQL Limits	
SQL.MAX.BINARY.LITERAL.LEN	SQL.MAX.OWNER.NAME.LEN

ODBC *info.type* Values

SQL.MAX.CHAR.LITERAL.LEN	SQL.MAX.PROCEDURE.NAME.LEN
SQL.MAX.COLUMN.NAME.LEN	SQL.MAX.QUALIFIER.NAME.LEN
SQL.MAX.COLUMNS.IN.GROUP.BY	SQL.MAX.ROW.SIZE
SQL.MAX.COLUMNS.IN.ORDER.BY	SQL.MAX.ROW.SIZE.INCLUDES.LONG
SQL.MAX.COLUMNS.IN.INDEX	SQL.MAX.STATEMENT.LEN
SQL.MAX.COLUMNS.IN.SELECT	SQL.MAX.TABLE.NAME.LEN
SQL.MAX.COLUMNS.IN.TABLE	SQL.MAX.TABLES.IN.SELECT
SQL.MAX.CURSOR.NAME.LEN	SQL.MAX.USER.NAME.LEN
SQL.MAX.INDEX.SIZE	
Scalar Function Information	
SQL.CONVERT.FUNCTIONS	SQL.TIMEDATE.ADD.INTERVALS
SQL.NUMERIC.FUNCTIONS	SQL.TIMEDATE.DIFF.INTERVALS
SQL.STRING.FUNCTIONS	SQL.TIMEDATE.FUNCTIONS
SQL.SYSTEM.FUNCTIONS	
Conversion Information	
SQL.CONVERT.BIGINT	SQL.CONVERT.LONGVARCHAR
SQL.CONVERT.BINARY	SQL.CONVERT.NUMERIC
SQL.CONVERT.BIT	SQL.CONVERT.REAL
SQL.CONVERT.CHAR	SQL.CONVERT.SMALLINT
SQL.CONVERT.DATE	SQL.CONVERT.TIME
SQL.CONVERT.DECIMAL	SQL.CONVERT.TIMESTAMP
SQL.CONVERT.DOUBLE	SQL.CONVERT.TINYINT

ODBC *info.type* Values

SQL.CONVERT.FLOAT	SQL.CONVERT.VARBINARY
SQL.CONVERT.INTEGER	SQL.CONVERT.VARCHAR
SQL.CONVERT.LONGVARBINARY	

ODBC *info.type* Values

UniVerse NLS info.type Values

The following table lists the valid UniVerse NLS values for *info.type*. For information about UniVerse NLS, see the *UniVerse NLS Guide*.

Data Source Information	
SQL.UVNLS.FIELD.MARK	SQL.UVNLS.LC.NUMERIC
SQL.UVNLS.ITEM.MARK	SQL.UVNLS.LC.TIME
SQL.UVNLS.MAP	SQL.UVNLS.SQL.NULL
SQL.UVNLS.LC.ALL	SQL.UVNLS.SUBVALUE.MARK
SQL.UVNLS.LC.COLLATE	SQL.UVNLS.TEXT.MARK
SQL.UVNLS.LC.CTYPE	SQL.UVNLS.VALUE.MARK
SQL.UVNLS.LC.MONETARY	

UniVerse NLS info.type Values

Return Values

The following table describes the return values.

Return Value	Description
0	SQL.SUCCESS

SQLGetInfo Return Values

Return Value	Description
1	SQL.SUCCESS.WITH.INFO
-1	SQL.ERROR
-2	SQL.INVALID.HANDLE

SQLGetInfo Return Values (Continued)

SQLGetTypeInfo

SQLGetTypeInfo returns information about an SQL on the data source. You can use **SQLGetTypeInfo** only against ODBC data sources.

Syntax

status = **SQLGetTypeInfo** **(***statement.env*, *sql.type***)**

Input Variables

The following table describes the input variables.

Input Variable	Description	
statement.env	SQL statement environment.	
sql.type	A driver-specific SQL data type, or one of the following:	
	SQL.B.BINARY	SQL.LONGVARCHAR
	SQL.BIGINT	SQL.NUMERIC
	SQL.BINARY	SQL.REAL
	SQL.BIT	SQL.SMALLINT
	SQL.C.BINARY	SQL.TIME
	SQL.CHAR	SQL.TIMESTAMP
	SQL.DATE	SQL.TINYINT
	SQL.DECIMAL	SQL.VARBINARY
	SQL.DOUBLE	SQL.VARCHAR
	SQL.FLOAT	SQL.WCHAR
	SQL.INTEGER	SQL.WLONGVARCHAR
	SQL.LONGVARBINARY	SQL.WVARCHAR

SQLGetTypeInfo Input Variables

Description

SQLGetTypeInfo returns a standard result set ordered by DATA.TYPE and TYPE.NAME. The following table lists the columns in the result set. For more detailed information about data type information, see the *Microsoft ODBC 2.0 Programmer's Reference and SDK Guide.*

Column Name	Data Type	Description
TYPE.NAME	Varchar	Data-source-dependent data type name.
DATA.TYPE	Smallint	Driver-dependent or SQL data type.
PRECISION	Integer	Maximum precision of the data type on the data source.
LITERAL.PREFIX	Varchar(128)	Characters used to prefix a literal.
LITERAL.SUFFIX	Varchar (128)	Characters used to terminate a literal.
CREATE.PARAMS	Varchar(128)	Parameters for a data type definition.
NULLABLE	Smallint	Data type accepts null values. Returns one of the following: SQL.NO.NULLS SQL.NULLABLE SQL.NULLABLE.UNKNOWN
CASE.SENSITIVE	Smallint	Character data type is case-sensitive. Returns one of the following: TRUE if data type is a character data type and is case-sensitive FALSE if data type is not a character data type and is not case-sensitive

SQLGetTypeInfo Results

Column Name	Data Type	Description
SEARCHABLE	Smallint	How the WHERE clause uses the data type. Returns one of the following: SQL.UNSEARCHABLE if data type cannot be used SQL.LIKE.ONLY if data type can be used only with the LIKE predicate SQL.ALL.EXCEPT.LIKE if data type can be used with all comparison operators except LIKE SQL.SEARCHABLE if data type can be used with any comparison operator
UNSIGNED.ATTRIBUTE		
	Smallint	Data type is unsigned. Returns one of the following: TRUE if data type is unsigned FALSE if data type is signed NULL if attribute is not applicable to the data type or the data type is not numeric
MONEY	Smallint	Data type is a money data type. Returns one of the following: TRUE if data type is a money data type FALSE if it is not
AUTO.INCREMENT	Smallint	Data type is autoincrementing. Returns one of the following: TRUE if data type is autoincrementing FALSE if it is not NULL if attribute is not applicable to the data type or the data type is not numeric

SQLGetTypeInfo Results (Continued)

Column Name	Data Type	Description
LOCAL.TYPE.NAME	Varchar(128)	Localized version of TYPE.NAME.
MINIMUM.SCALE	Smallint	Minimum scale of the data type on the data source.
MAXIMUM.SCALE	Smallint	Maximum scale of the data type on the data source.

SQLGetTypeInfo Results (Continued)

Return Values

The following table describes the return values.

Return Value	Description
0	SQL.SUCCESS
1	SQL.SUCCESS.WITH.INFO
-1	SQL.ERROR
-2	SQL.INVALID.HANDLE

SQLGetTypeInfo Return Values

SQLNumParams

SQLNumParams returns the number of parameters in an SQL statement.

Syntax

status = **SQLNumParams (***statement.env*, *parameters***)**

Input Variable

The following table describes the input variable.

Input Variable	Description
statement.env	SQL statement environment containing the prepared or executed SQL statement. For connections to a local UniVerse server, *statement.env* can be @HSTMT.

SQLNumParams Input Variable

Output Variable

The following table describes the output variable.

Output Variable	Description
parameters	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 *statement.env* contains no parameters, *parameters* is set to 0.

Return Values

The following table describes the return values.

Return Value	Description
0	SQL.SUCCESS
-1	SQL.ERROR
-2	SQL.INVALID.HANDLE

SQLNumParams Return Values

SQLNumResultCols

SQLNumResultCols returns the number of columns in a result set.

Syntax

status = **SQLNumResultCols (***statement.env, cols***)**

Input Variable

The following table describes the input variable.

Input Variable	Description
statement.env	SQL statement environment containing the executed SQL statement. For connections to a local UniVerse server, *statement.env* can be @HSTMT.

SQLNumResultCols Input Variable

Output Variable

The following table describes the output variable.

Output Variable	Description
cols	Number of report columns generated.

SQLNumResultCols Output Variable

Return Values

The following table describes the return values.

Return Value	Description
0	SQL.SUCCESS
-1	SQL.ERROR
-2	SQL.INVALID.HANDLE

SQLNumResultCols Return Values

Description

Use this function after executing an SQL statement to find the number of columns in the result set. If the executed statement was not a SELECT statement or a called procedure that produced a result set, the number of result columns returned is 0.

Use this function when the number of columns to be bound to application variables is unknown—as, for example, when your program is processing SQL statements entered by users.

SQLParamOptions

SQLParamOptions lets applications specify an array of values for each parameter assigned by **SQLBindParameter**.

Syntax

status = **SQLParamOptions** (*statement.env*, *option*, *value*)

Variables

The following table describes the SQLParamOptions variables.

Variable	Description
statement.env	SQL statement environment. For connections to a local UniVerse server, *statement.env* can be @HSTMT.
option	One of the following, followed by a value: SQL.PARAMOPTIONS.SET *value* is an input variable containing the number of rows to process. It can be an integer from 1 through 1024. SQL.PARAMOPTIONS.READ *value* is an output variable containing the number of parameter rows processed by **SQLExecDirect** or **SQLExecute**. As each set of parameters is processed, *value* is updated to the current row number. If **SQLExecDirect** or **SQLExecute** encounters an error, *value* contains the number of the row that failed.

SQLParamOptions Variables

Return Values

The following table describes the return values.

Return Value	Description
0	SQL.SUCCESS
1	SQL.SUCCESS.WITH.INFO
-1	SQL.ERROR
-2	SQL.INVALID.HANDLE

SQLParamOptions Return Values

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 values for each parameter associated with an INSERT statement, then execute the INSERT statement once to perform the twenty insertions.

You can use **SQLParamOptions** before or after you issue SQLBindParameter calls. You need only one **SQLParamOptions** call with SQL.PARAMOPTIONS.SET, no matter how many parameters are bound.

SQLParamOptions requires that the *param* argument specified in **SQLBindParameter** point to the first element of an array of parameter values.

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.

After you issue an **SQLParamOptions** call with SQL.PARAMOPTIONS.SET, the client program must supply all parameters as arrays until an SQLFreeStmt call drops the statement environment or resets all parameter marker variables.

UniVerse Data Sources

After an **SQLParamOptions** call, **SQLExecute** and **SQLExecDirect** can execute only the following statements until the statement environment is dropped or the parameter marker variables are reset:

- INSERT
- UPDATE
- DELETE

SQLParamOptions works only for input parameter types.

ODBC Data Sources

SQLParamOptions works for all parameter types—output and input/output parameters as well as the more usual input parameters.

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.

```
  1   $include UNIVERSE.INCLUDE ODBC.H
arrsize = 20
dim p1(arrsize)
dim p2(arrsize)
SQLINS1 = "INSERT INTO TAB1 VALUES(?,?)"
rowindex = 0

status = SQLAllocEnv(henv)
status = SQLAllocConnect(henv, hdbc)
status = SQLConnect(hdbc, "odbcds", "dbuid", "dbpwd")
status = SQLAllocStmt(hdbc, hstmt)

status = SQLPrepare(hstmt, SQLINS1)
status = SQLBindParameter(hstmt, 1, SQL.B.BASIC, SQL.INTEGER,
0, 0, p1(1),
SQL.PARAM.INPUT)

status = SQLBindParameter(hstmt, 2, SQL.B.BASIC, SQL.CHAR, 30,
0, p2(1),
SQL.PARAM.INPUT)

status = SQLParamOptions(hstmt, SQL.PARAMOPTIONS.SET, arrsize)
for index = 1 to arrsize
p1(index) = index
```

```
p2(index) = "This is row ":index
next index

* now execute, delivering 20 sets of parameters in one network
operation

stexec = SQLExecute(hstmt)
status = SQLParamOptions(hstmt, SQL.PARAMOPTIONS.READ,
rowindex)

if stexec = SQL.ERROR then
print "Error in parameter row number ":rowindex
end else
print rowindex:" parameter marker sets were processed"
end
```

SQLPrepare

SQLPrepare passes an SQL statement or procedure call to the data source in order to prepare it for execution by **SQLExecute**.

Syntax

status = **SQLPrepare (***statement.env, statement***)**

Input Variables

The following table describes the input variables.

Input Variable	Description
statement.env	SQL statement environment from a previous SQLAllocStmt. For connections to a local UniVerse server, *statement.env* can be @HSTMT.
statement	Either an SQL statement or a call to an SQL procedure, to be executed at the data source. If you are connected to a UniVerse server, it treats the SQL statement case-sensitively; all keywords must be in uppercase letters. If you are connected to an ODBC data source, it may treat identifiers and keywords in the SQL statement case-sensitively. To call an SQL procedure, use one of the following syntaxes: [{] CALL *procedure* [([*parameter* [, *parameter*] …])] [}] CALL *procedure* [*argument* [*argument*] …] If you are connected to an ODBC data source, use the first syntax and enclose the entire **call** statement in braces.

SQLPrepare Input Variables

Input Variable	Description	
	procedure	Name of the procedure. If the procedure name contains characters other than alphabetic or numeric, enclose the name in double quotation marks. To embed a single double quotation mark in the procedure name, use two consecutive double quotation marks.
	parameter	Either a literal value or a parameter marker that marks where to insert values to send to or receive from the data source. Programmatic SQL uses a ? (question mark) as a parameter marker. Use parameters only if the procedure is a subroutine. The number and order of parameters must correspond to the subroutine arguments. For an ODBC data source, parameters should be of the same data type as the procedure requires.
	argument	Any valid keyword, literal, or other token you can use in a UniVerse command line.

SQLPrepare Input Variables (Continued)

Return Values

The following table describes the return values.

Return Value	Description
0	SQL.SUCCESS
1	SQL.SUCCESS.WITH.INFO
-1	SQL.ERROR
-2	SQL.INVALID.HANDLE

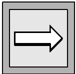
SQLPrepare Return Values

Description

Use this function to deliver either an SQL statement or a call to an SQL procedure to the data source where it can prepare to execute the passed SQL statement or the procedure. The application subsequently uses SQLExecute to tell the data source to execute the prepared SQL statement or procedure.

What happens when the data source executes the **SQLPrepare** call depends on the data source. In many cases the data source parses the SQL statement and generates an execution plan that allows rapid, efficient execution of the SQL statement.

Use **SQLPrepare** and **SQLExecute** functions when you are issuing SQL statements or calling a procedure repeatedly. You can supply values to a prepared INSERT or UPDATE statement and issue an **SQLExecute** call each time you change the values of 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 an SQLBindParameter call, otherwise* **SQLExecute** *returns an error.*

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.

UniVerse Data Sources

If you are connected to a UniVerse server, SQL statements can refer to UniVerse files as well as to SQL tables.

ODBC Data Sources

If you execute a stored procedure or enter a command batch with multiple SELECT statements, the results of only the first SELECT statement are returned.

SQLRowCount

SQLRowCount returns the number of rows changed by UPDATE, INSERT, or DELETE statements, or by a called procedure that executes one of these statements.

Syntax

status = **SQLRowCount (***statement.env, rows***)**

Input Variable

The following table describes the input variable.

Input Variable	Description
statement.env	SQL statement environment containing the executed SQL statement. For connections to a local UniVerse server, *statement.env* can be @HSTMT.

SQLRowCount Input Variable

Output Variable

The following table describes the output variable.

Output Variable	Description
rows	Number of rows affected by the operation.

SQLRowCount Output Variable

Return Values

The following table describes the return values.

Return Value	Description
0	SQL.SUCCESS
-1	SQL.ERROR
-2	SQL.INVALID.HANDLE

SQLRowCount Return Values

Description

Statements such as GRANT and CREATE TABLE, which do not update rows in the database, return 0 in *rows*.

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.

The value of *rows* returned after executing a stored procedure at the data source may not be accurate. It is accurate for a single UPDATE, INSERT, or DELETE statement.

SQLSetConnectOption

SQLSetConnectOption controls some aspects of the connection to a data source, such as operating system login information and whether empty string data is returned as null values.

Syntax

status = **SQLSetConnectOption** (*connect.env, option, value*)

Input Variables

The following table describes the input variable.

Input Variable	Description
connect.env	Connection environment returned from a previous SQLAllocConnect. For connections to a local UniVerse server, *connect.env* can be @HDBC.

SQLSetConnect Option

Options

option is one of the following, followed by a value:

Option	Description	
SQL.AUTOCOMMIT	*value* is one of the following:	
	SQL.AUTOCOMMIT.ON	Puts a private connection into autocommit mode.
	SQL.AUTOCOMMIT.OFF	Puts a private connection into manual commit mode.

SQLSetConnect Option

Option	Description
	Use this option only with ODBC data sources. When you use it, the connection must already be established, and the SQL.PRIVATE.TX option must be set to SQL.PRIVATE.TX.ON
SQL.EMPTY.NULL	A value that helps control whether the SQL Client Interface interprets empty strings on a UniVerse data source as equivalent to the null value. *value* is one of the following: SQL.EMPTY.NULL.OFF (default) SQL.EMPTY.NULL.ON Use this option only with UniVerse data sources. You can set this option before or after you connect to the data source.
SQL.LIC.DEV.SUBKEY	A string of up to 24 characters, used to uniquely identify client devices for UniVerse licensing when an application connects to a UniVerse server through a multiple-tier connection. You must set this option before you connect to the data source.
SQL.OS.PWD	Specifies the user's password when connecting to the data source operating system. *value* is a character string. If the string is NULL or empty, the operating system password is an empty string. Use this option only with UniVerse data sources, and you must set it before you connect to the data source.
SQL.OS.UID	Specifies the user's login name when connecting to the data source operating system. *value* is a character string. If the string is NULL or empty, SQLConnect tries to connect to the operating system using the user name specified by the *logon1* variable in the **SQLConnect** call. Use this option only with UniVerse data sources, and you must set it before you connect to the data source.
SQL.PRIVATE.TX	*value* is one of the following:

SQLSetConnect Option (Continued)

Option	Description	
	SQL.PRIVATE.TX.ON	Specifies that transaction processing is controlled directly by the application instead of by the UniVerse transaction manager.
	SQL.PRIVATE.TX.OFF	Specifies that transaction processing is controlled by the UniVerse transaction manager instead of by the application directly.
	Use this option only with ODBC data sources, and only after you connect to the data source.	
SQL.TXN.ISOLATION	*value* is one of the following:	
	SQL.TXN.READ. UNCOMMITTED	Sets the server's isolation level to 1.
	SQL.TXN.READ. COMMITTED	Sets the server's isolation level to 2.
	SQL.TXN.REPEATABLE. READ	Sets the server's isolation level to 3.
	SQL.TXN. SERIALIZABLE	Sets the server's isolation level to 4.
	SQL.TXN.VERSIONING	No UniVerse equivalent.
	Use this option only with ODBC data sources. When you use it, the connection must already be established, the SQL.PRIVATE.TX option must be set to SQL.PRIVATE.TX.ON, and no transactions can be active.	

SQLSetConnect Option (Continued)

Option	Description
SQL.UVNLS.LC.ALL	A value that specifies all components of a locale. *value* is a slash-separated list of five values, as set up in the server's NLS.LC tables: *value1*/*value2*/*value3*/*value4*/*value5* Use this option only with UniVerse data sources running with NLS enabled. You can set this option before or after you connect to the data source.
SQL.UVNLS.LC.COLLATE	A value that specifies the name of a locale whose sort order to use. Use this option only with UniVerse data sources running with NLS enabled. You can set this option before or after you connect to the data source.
SQL.UVNLS.LC.CTYPE	A value that specifies the name of a locale whose character type to use. Use this option only with UniVerse data sources running with NLS enabled. You can set this option before or after you connect to the data source.
SQL.UVNLS.LC.MONETARY	A value that specifies the name of a locale whose monetary conventions to use. Use this option only with UniVerse data sources running with NLS enabled. You can set this option before or after you connect to the data source.
SQL.UVNLS.LC.NUMERIC	A value that specifies the name of a locale whose numeric conventions to use. Use this option only with UniVerse data sources running with NLS enabled. You can set this option before or after you connect to the data source.
SQL.UVNLS.LC.TIME	A value that specifies the name of a locale whose time conventions to use. Use this option only with UniVerse data sources running with NLS enabled. You can set this option before or after you connect to the data source.
SQL.UVNLS.LOCALE	A value that specifies the name of a locale, all of whose conventions are to be used. Use this option only with UniVerse data sources running with NLS enabled. You can set this option before or after you connect to the data source.

SQLSetConnect Option (Continued)

Return Values

The following table describes the return values.

Return Value	Description
0	SQL.SUCCESS
-1	SQL.ERROR
-2	SQL.INVALID.HANDLE

SQLSetConnect Return Values

Description

The SQL.OS.UID and SQL.OS.PWD options let you specify user names and passwords for the operating system of a UniVerse server.

Converting Empty Strings to Null Values

If you want your client program to return empty strings as null values from a UniVerse data source, and to convert null values to empty strings when inserting or updating data on a UniVerse data source, complete the following steps:

- 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 to SQL.EMPTY.NULL.ON.

SQL.EMPTY.NULL.OFF keeps empty strings and null values as distinct values. SQL.EMPTY.NULL.ON forces empty strings to be treated as null values in those files whose dictionary contains the appropriate @EMPTY.NULL entry.

Private Transactions

SQL.PRIVATE.TX.ON frees the connection from being managed by the UniVerse transaction manager. When a connection is made private, the application can use the SQL.AUTOCOMMIT option to put the connection into either autocommit mode or manual commit mode. By default, private connections are in autocommit mode, in which each SQL statement is treated as a separate transaction, committed after the statement is executed.

In manual commit mode the application can do either of the following:

- Use the SQLTransact function to commit or roll back changes to the database.
- Set the SQL.AUTOCOMMIT option of **SQLSetConnectOption** to SQL.AUTOCOMMIT.ON. This commits any outstanding transactions and returns the connection to autocommit mode.

You must return the connection to autocommit mode before using SQLDisconnect to close the connection. You can do this in two ways:

- Set the SQL.AUTOCOMMIT option of **SQLSetConnectOption** to SQL.AUTOCOMMIT.ON
- Set the SQL.PRIVATE.TX option of **SQLSetConnectOption** to SQL.PRIVATE.TX.OFF

When a connection is private, SQL.TXN.ISOLATION lets the application define the default transaction isolation level at which to execute server operations. To determine what isolation levels the data source supports, use the SQL.TXN.ISOLATION.OPTION option of the SQLGetInfo function. This returns a bitmap of the options the data source supports. The application can then use the BASIC BIT functions to determine whether a particular bit is set in the bitmap.

Use **SQLSetConnectOption** with the SQL.TXN.ISOLATION option only in the following two places:

- Immediately following an SQLConnect function call
- Immediately following an SQLTransact call to commit or roll back an operation

Whenever you execute an SQL statement, a new transaction exists, which makes setting the SQL.TXN.ISOLATION option illegal. If a transaction is active when the SQL.TXN.ISOLATION.OPTION is set, the SQL Client Interface returns SQL.ERROR and sets SQLSTATE to S1C00.

SQLSetParam

SQLSetParam has been superseded by SQLBindParameter.

SQLSpecialColumns

SQLSpecialColumns gets information about columns in a table. Use this function only when you are connected to an ODBC data source.

Syntax

status = **SQLSpecialColumns** (*statement.env*, *col.type*, *schema*, *owner*, *tablename*, *IDscope*, *null*)

Input Variables

The following table describes the input variables.

Input Variable	Description	
statement.env	SQL statement environment.	
col.type	Type of column to return. *col.type* is one of the following:	
	SQL.BEST.ROWID	Returns the best column or set of columns that uniquely identifies a row in a table.
	SQL.ROWVER	Returns the column or columns that are automatically updated when any value in the row is updated by a transaction.
schema	Qualifier name for *tablename*. If a driver supports qualifiers for some tables but not others, use an empty string for tables that do not have qualifiers.	
owner	Name of the owner of the table. If a driver supports owners for some table but not others, use an empty string for tables that do not have owners.	
tablename	Name of the table.	
IDScope	Minimum required scope of the row ID. *IDscope* is one of the following:	

SQLSpecialColumns Input Variables

Input Variable	Description	
	SQL.SCOPE.CURROW	Row ID is guaranteed to be valid only while positioned on that row.
	SQL.SCOPE.TRANSACTION	Row ID is guaranteed to be valid for the duration of the current transaction.
	SQL.SCOPE.SESSION	Row ID is guaranteed to be valid for the duration of the session.
null	Can be one of the following:	
	SQL.NO.NULLS	Excludes special columns that can have null values.
	SQL.NULLABLE	Returns special columns even if they can have null values.

SQLSpecialColumns Input Variables (Continued)

Return Values

The following table describes the return values.

Return Value	Description
0	SQL.SUCCESS
1	SQL.SUCCESS.WITH.INFO
-1	SQL.ERROR
-2	SQL.INVALID.HANDLE

SQLSpecialColumns Return Values

Description

Results are returned as a standard result set ordered by SCOPE. The following table lists the columns in the result set. The lengths of VARCHAR columns are maximums; the actual lengths depend on the data source. To get the length of the COLUMN.NAME column, use the SQL.MAX.COLUMN.NAME.LEN option of the SQLGetInfo function.

Column Name	Data Type	Description
SCOPE	Smallint	Actual scope of the row ID. It contains one of the following: SQL.SCOPE.CURROW SQL.SCOPE.TRANSACTION SQL.SCOPE.SESSION The null value is returned when *col.type* is SQL.ROWVER.
COLUMN.NAME	Varchar(128) Not null	Column identifier.
DATA.TYPE	Smallint Not null	Either an ODBC SQL data type or a driver-specific SQL data type.
TYPE.NAME	Varchar(128) Not null	Data-source-dependent data type name.
PRECISION	Integer	Precision of the column on the data source. The null value is returned for data types where precision does not apply.

SQLSpecialColumns Results

Column Name	Data Type	Description
LENGTH	Integer	The length in bytes of data transferred on an **SQLGetData** or **SQLFetch** if SQL.C.DEFAULT is specified. For numeric data, this size can differ from the size of the data stored on the data source. This value is the same as the PRECISION column for character or binary data.
SCALE	Smallint	The scale of the column on the data source. The null value is returned for data types where scale does not apply.
PSEUDO.COLUMN	Smallint	Indicates whether the column is a pseudo-column: SQL.PC.UNKNOWN SQL.PC.PSEUDO SQL.PC.NOT.PSEUDO Pseudo-columns should not be quoted with the identifier quote character returned by **SQLGetInfo**.

SQLSpecialColumns Results (Continued)

SQLSpecialColumns lets applications scroll forward and backward in a result set to get the most recent data from a set of rows. Columns returned for column type SQL.BEST.ROWID are guaranteed not to change while positioned on that row. Columns of the row ID can remain valid even when the cursor is not positioned on the row. The application can determine this by checking the SCOPE column in the result set.

Once the application gets values for SQL.BEST.ROWID, it can use these values to reselect that row within the defined scope. The SELECT statement is guaranteed to return either no rows or one row.

Columns returned for SQL.BEST.ROWID can always be used in an SQL select expression or WHERE clause. However, SQLColumns does not necessarily return these columns. If no columns uniquely identify each row in the table, **SQLSpecialColumns** returns a row set with no rows; a subsequent call to SQLFetch returns SQL.NO.DATA.FOUND.

Columns returned for column type SQL.ROWVER let applications check if any columns in a given row have been updated while the row was reselected using the row ID.

If *col.type*, *IDscope*, or *null* specifies characteristics not supported by the data source, **SQLSpecialColumns** returns a result set with no rows, and a subsequent call to **SQLFetch** returns SQL.NO.DATA.FOUND.

For complete details about the **SQLSpecialColumns** function, see the *Microsoft ODBC 2.0 Programmer's Reference and SDK Guide*.

SQLStatistics

SQLStatistics gets a list of statistics about a single table and its indexes. Use this function only when you are connected to an ODBC data source.

Syntax

status = **SQLStatistics** (*statement.env*, *schema*, *owner*, *tablename*, *index.type*, *accuracy*)

Input Variables

The following table describes the input variables.

Input Variable	Description	
statement.env	SQL statement environment.	
schema	Qualifier name for *tablename*. If a driver supports qualifiers for some tables but not others, use an empty string for tables that do not have qualifiers.	
owner	Name of the owner of the table. If a driver supports owners for some table but not others, use an empty string for tables that do not have owners.	
tablename	Name of the table.	
index.type	One of the following: SQL.INDEX.UNIQUE SQL.INDEX.ALL	
accuracy	The importance of the CARDINALITY and PAGES columns in the result set:	
	SQL.ENSURE	The driver unconditionally gets the statistics.
	SQL.QUICK	The driver gets results only if they are readily available from the server. The driver does not ensure that the values are current.

SQLStatistics Input Variables

Return Values

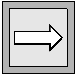
The following table describes the return values.

Return Value	Description
0	SQL.SUCCESS
1	SQL.SUCCESS.WITH.INFO
-1	SQL.ERROR
-2	SQL.INVALID.HANDLE

SQLStatistics Return Values

Description

SQLStatistics returns information as a standard result set ordered by NON.UNIQUE, TYPE, INDEX.QUALIFIER, INDEX.NAME, and SEQ.IN.INDEX. The result set combines statistics for the table with statistics for each index. The following table lists the columns in the result set.



***Note*: SQLStatistics** *might not return all indexes. For example, a driver might return only the indexes in files in the current directory. Applications can use any valid index regardless of whether it is returned by* **SQLStatistics**.

The lengths of VARCHAR columns are maximums; the actual lengths depend on the data source. Use SQLGetInfo to determine the actual lengths of the TABLE.QUALIFIER, TABLE.OWNER, TABLE.NAME, and COLUMN.NAME columns.

Column Name	Data Type	Description
TABLE.QUALIFIER	Varchar(128)	Table qualifier identifier (schema) of the table. The null value is returned if it is not applicable to the data source. If a driver supports qualifiers for some tables but not others, it returns an empty string for tables without qualifiers.
TABLE.OWNER	Varchar(128)	Name of the owner of the table. The null value is returned if it is not applicable to the data source. If a driver supports owners for some tables but not others, it returns an empty string for tables without owners.
TABLE.NAME	Varchar(128) Not null	Name of the table.
NON.UNIQUE	Smallint	The index prohibits duplicate values: TRUE if the index values can be nonunique. FALSE if the index values must be unique. NULL if TYPE is SQL.TABLE.STAT.
INDEX.QUALIFIER	Varchar(128)	Index qualifier identifier used by the DROP INDEX statement. The null value is returned if the data source does not support index qualifiers or if TYPE is SQL.TABLE.STAT. If a nonnull value is returned, it must be used to qualify the index name in a DROP INDEX statement, otherwise the TABLE.OWNER name should be used to qualify the index name.
INDEX.NAME	Varchar(128)	Name of the index. The null value is returned if TYPE is SQL.TABLE.STAT.

SQLStatistics Results

Column Name	Data Type	Description
TYPE	Smallint Not null	Type of information returned: SQL.TABLE.STAT indicates a statistic for the table. SQL.INDEX.CLUSTERED indicates a clustered index. SQL.INDEX.HASHED indicates a hashed index. SQL.INDEX.OTHER indicates another type of index.
SEQ.IN.INDEX	Smallint	Column sequence number in index, starting with 1. The null value is returned if TYPE is SQL.TABLE.STAT.
COLUMN.NAME	Varchar(128)	Name of a column. If the column is based on an expression, the expression is returned. If the expression cannot be determined, an empty string is returned. If the index is filtered, each column in the filter condition is returned (this may require more than one row). The null value is returned if TYPE is SQL.TABLE.STAT.
COLLATION	Char(1)	Sort sequence for the column: A indicates ascending. B indicates descending. The null value is returned if the data source does not support column sort sequence.

SQLStatistics Results (Continued)

Column Name	Data Type	Description
CARDINALITY	Integer	Number of rows in the table if TYPE is SQL.TABLE.STAT. Number of unique values in the index if TYPE is not SQL.TABLE.STAT. The null value is returned if the value is not available from the data source or if it is not applicable to the data source.
PAGES	Integer	Number of pages for the table if TYPE is SQL.TABLE.STAT. Number of pages for the index if TYPE is not SQL.TABLE.STAT. The null value is returned if the value is not available from the data source or if it is not applicable to the data source.
FILTER. CONDITION	Varchar(128)	If the index is filtered, the filter condition, or an empty string if the filter condition cannot be determined. The null value is returned if the index is not filtered, or if it cannot be determined that the index is filtered, or TYPE is SQL.TABLE.STAT.

SQLStatistics Results (Continued)

If the row in the result set corresponds to a table, the driver sets TYPE to SQL.TABLE.STAT and sets the following columns to NULL:

- NON.UNIQUE
- INDEX.QUALIFIER
- INDEX.NAME
- SEQ.IN.INDEX
- COLUMN.NAME
- COLLATION

If CARDINALITY or PAGES are not available from the data source, the driver sets them to NULL.

For complete details about the **SQLStatistics** function, see the *Microsoft ODBC 2.0 Programmer's Reference and SDK Guide*.

SQLTables

SQLTables returns a result set listing the tables matching the search patterns. Use this function only when you are connected to an ODBC data source.

Syntax

status = **SQLTables (***statement.env, schema, owner, tablename, type***)**

Input Variables

The following table describes the input variables.

Input Variable	Description
statement.env	SQL statement environment.
schema	Schema name search pattern.
owner	Table owner number search pattern.
tablename	Table name search pattern.
type	Table type (one of the following: BASE TABLE, VIEW, ASSOCIATION, or TABLE) search pattern.

SQLTables Input Variables

Description

This function returns *statement.env* as a standard result set of five columns containing the schemas, owners, names, types, and remarks for all tables found by the search. The search criteria arguments can contain a literal, 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. *owner* cannot specify a LIKE pattern. You can access the result set with SQLFetch. These five columns have the following descriptors:

TABLE.SCHEMA	VARCHAR(128)
TABLE.OWNER	VARCHAR(128)
TABLE.NAME	VARCHAR(128)
TABLE.TYPE	VARCHAR(128)
REMARKS	VARCHAR(254)

Special Search Criteria

Three special search criteria combinations enable an application to enumerate the set of schemas, owners, and tables:

Table Qualifier	Table Owner	Table Name	Table Type	Return is...
%	empty string	empty string	*ignored*	Set of distinct schema names
empty string	%	empty string	*ignored*	Set of distinct table owners
empty string	empty string	empty string	%	Set of distinct table types

Search Criteria Combinations

The ability to obtain information about tables does not imply that you have any privileges on those tables.

Return Values

The following table describes the return values.

Return Value	Description
0	SQL.SUCCESS
1	SQL.SUCCESS.WITH.INFO
-1	SQL.ERROR
-2	SQL.INVALID.HANDLE

SQLTables Return Values

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.
S1008	Cancelled. Execution of the statement was stopped by an SQLCancel call.
S1010	Function sequence error. *statement.env* 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 be caused by a variety of reasons. The native error code returned by the SQLError call indicates the specific UniVerse error that occurred.

SQLTables SQLSTATE Values

SQLTransact

SQLTransact requests a COMMIT or ROLLBACK for all SQL statements associated with a connection or all connections associated with an environment. Use this function only when you are connected to an ODBC data source.

Syntax

status = **SQLTransact (***bci.env, connect.env, type***)**

Input Variables

The following table describes the input variables.

Input Variables	Description	
bci.env	SQL Client Interface environment.	
connect.env	Connection environment or SQL.NULL.HDBC.	
type	One of the following:	
	SQL.COMMIT	Writes all modified data to the data source, releases all lock acquired by the current transaction, and terminates the transaction.
	SQL.ROLLBACK	Discards any changes written during the transaction and terminates it.

SQLTransact Input Variables

Description

This function provides the same transaction functions as the UniVerse BASIC statements COMMIT, and ROLLBACK. When *connect.env* is a valid connection environment with SQL.AUTOCOMMIT set to OFF, **SQLTransact** commits or rolls back the connection.

To use **SQLTransact**, all of the following conditions must be met:

- The SQL.PRIVATE.TX option of SQLSetConnectOption is set to SQL.PRIVATE.TX.ON.
- The SQL.AUTOCOMMIT option is set to SQL.AUTOCOMMIT.OFF.
- The connection is active.

Setting *bci.env* to a valid environment handle and *connect.env* to SQL.NULL.HDBC requests the server to try to execute the requested action on all *hdbc*s 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 connection environment in turn.

If you call **SQLTransact** with a *type* of SQL.COMMIT or SQL.ROLLBACK when no transaction is active, SQL.SUCCESS is returned.

Return Values

The following table describes the return values.

Return Value	Description
0	SQL.SUCCESS
-1	SQL.ERROR
-2	SQL.INVALID.HANDLE

SQLTransact Return Values

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.

SQLTransact SQLSTATE Values

SQLSTATE	Description
S1012	*type* did not contain SQL.COMMIT or SQL.ROLLBACK.
08003	No connection is active on *connect.env*.
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 (Continued)

Data Conversion

This appendix describes the UniVerse BASIC and ODBC SQL data types you can specify with the SQLBindParameter and SQLBindCol calls. It also explains how data is converted.

You use **SQLBindParameter** to convert BASIC data to an ODBC SQL data type that can be sent to the data source. You specify the BASIC data type as SQL.B.BASIC, except in two cases:

- If you are sending a date or a time in internal format to a UniVerse data source, you specify the BASIC data type as SQL.B.INTDATE or SQL.B.INTTIME.
- If you are sending raw data as a bit string to any data source, you specify the BASIC data type as SQL.B.BINARY.

You use **SQLBindCol** to convert SQL data retrieved from the data source, to a BASIC data type. You can specify that data retrieved from the data source be stored locally as:

- A character string (SQL.B.CHAR)
- A bit string (SQL.B.BINARY)
- A number (SQL.B.NUMBER)
- An internal date (SQL.B.INTDATE)
- An internal time (SQL.B.INTTIME)

You can also specify SQL.B.DEFAULT. This lets the ODBC SQL data type determine the BASIC data type to which to convert data from the data source. The NULL data type requires no conversion.

The following table shows the BASIC SQL Client Interface data types and their UniVerse BASIC counterparts.

BASIC SQL Client Interface Data Type	BASIC Data Type	Uses in Calls
SQL.B.CHAR	Character string.	**SQLBindCol**
SQL.B.BINARY	Bit string (raw data)	SQLBindCol SQLBindParameter
SQL.B.NUMBER	Integer. Double.	**SQLBindCol**
SQL.B.DEFAULT	SQL data type at the source determines how to store locally.	**SQLBindCol**
SQL.B.INTDATE	Internal date.	**SQLBindCol** **SQLBindParameter**
SQL.B.INTTIME	Internal time	**SQLBindCol** **SQLBindParameter**
SQL.B.BASIC	The data determines the BASIC data type. The data must be string, integer, or double.	**SQLBindParameter**

SQL Client Interface and BASIC Data Types

The following table shows the ODBC SQL data types and their SQL definitions. You can use these ODBC SQL data types when you connect to UniVerse data sources and to ODBC data sources.

ODBC SQL Data Type	SQL Data Type	Description and SQL Client Interface Defaults
SQL.CHAR	CHAR (*n*)	Fixed-length character string precision = *n* where 1 <= *n* <= 254
SQL.VARCHAR SQL.LONGVARCHAR	VARCHAR (*n*)	Variable-length character string precision = *n* where 1 <= *n* <= 65535
SQL.WCHAR	NCHAR (*n*)	Fixed-length national character string precision = *n* where 1 <= *n* <= 254
SQL.WVARCHAR SQL.WLONGVARCHAR	NVARCHAR (*n*)	Variable-length national character string precision = *n* where 1 <= *n* <= 65535
SQL.BINARY	BIT (*n*)	Fixed-length bit string (raw data) precision = *n* where 1 <= *n* <= 2032
SQL.VARBINARY SQL.LONGVARBINARY	VARBIT (*n*)	Variable-length bit string (raw data) precision = *n* where 1 <= *n* <= 524,280
SQL.DECIMAL	DECIMAL (*p*,*s*)	Signed exact numeric precision = *p*, scale = *s*
SQL.NUMERIC	NUMERIC (*p*,*s*)	Signed exact numeric precision = *p*, scale = *s*
SQL.SMALLINT	SMALLINT	Signed exact numeric precision = 5, scale = 0
SQL.INTEGER	INTEGER	Signed exact numeric precision = 10, scale = 0
SQL.REAL	REAL	Signed approximate numeric precision = 7
SQL.FLOAT	FLOAT	Signed approximate numeric precision = 15

UniVerse SQL Data Types

ODBC SQL Data Type	SQL Data Type	Description and SQL Client Interface Defaults
SQL.DOUBLE	DOUBLE PRECISION	Signed approximate numeric precision = 15
SQL.DATE	DATE	Data source dependent
SQL.TIME	TIME	Internal time.

UniVerse SQL Data Types (Continued)

The following table represents the ODBC SQL data types currently supported by the BASIC SQL Client Interface. Some data types encountered on ODBC data sources cannot be mapped into any of these types. Your programs cannot fetch data from such column types. If you try to execute an SQL SELECT statement that produces a column with an unsupported data type, an application error (SQLSTATE code S1004) is returned to the execution call. When this happens, use the SQLDescribeCol or SQLColAttributes function to determine which column you requested contains an unsupported data type. The ODBC SQL data type for these columns is returned as SQL.BINARY.

The following table lists ODBC SQL data types you can use when connected to certain ODBC data sources. Do not use these data types when connected to a UniVerse data source unless your connection uses UniVerse ODBC.

ODBC SQL Data Type	Description
SQL.TIMESTAMP[a]	Date and time information in the formats *yyyy-mm-dd* and *hh*:*mm*:*ss*.
SQL.BIGINT[b]	Signed integer up to 19 digits, unsigned integer up to 20 digits.
SQL.TINYINT[c]	Signed integer from 0 through 255; unsigned integer from 128 through +127.
SQL.BIT	Data values of 0 or 1.

ODBC SQL Data Types

a. ODBC data sources vary in compliance with the ODBC specification regarding what to do with a missing component. For safety, always supply both components of a timestamp parameter.

b. Valid only for data sources such as ORACLE that support integers longer than 10 digits.

c. Use the **SQLGetTypeInfo** function to determine if the column is signed or unsigned.

Converting BASIC Data to SQL Data

Use the SQLBindParameter call to specify the ODBC SQL data type to which to convert outgoing BASIC data supplied for a parameter marker. This section describes how the BASIC SQL Client Interface converts outgoing data.

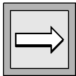
SQLBindParameter verifies that the BASIC data type is either SQL.B.BASIC, SQL.B.BINARY, SQL.B.INTDATE, or SQL.B.INTTIME. It does not check whether the data type is consistent with a data conversion that may occur later.

Both SQLExecute and SQLExecDirect calls check the data type and convert the data. Data from locations indicated in **SQLBindParameter** must be one of the following:

- A number (integer or double data types)
- A character string
- A bit string (raw data)
- A subroutine
- A null value

Any other kind of data returns an error to the **SQLExecute** or **SQLExecDirect** call.

All data sent to UniVerse data sources is sent as character string data. Multivalued data is sent to UniVerse data sources as multivalued dynamic arrays.



***Note**: You cannot send multivalued bit string values to a UniVerse data source, because the UniVerse system delimiters cannot be recognized in a stream of raw data bits.*

When converting BASIC character string data to numeric SQL data types, all numbers are rounded to 15 digits. The SQLSetConnectOption flag, SQL.TRUNC.ROUND, is ignored.

Precision and Scale

Precision and scale specified by the **SQLBindParameter** function are ignored if the data source is UniVerse.

For ODBC data sources, precision is observed for the following ODBC SQL data types:

SQL.CHARSQL.WCHAR
SQL.VARCHARSQL.WVARCHAR
SQL.LONGVARCHARSQL.WLONGVARCHAR
SQL.BINARYSQL.NUMERIC
SQL.VARBINARYSQL.DECIMAL
SQL.LONGVARBINARY

Scale is observed for the following data types:

SQL.DECIMAL
SQL.NUMERIC

All data sent to ODBC data sources is checked by the ODBC driver and database engine. Thus, conversion failures can be detected either by the BASIC SQL Client Interface, by the ODBC driver, or by the underlying DBMS.

UniVerse Data Storage Format

UniVerse files sometimes use an internal format to store DECIMAL and NUMERIC data types. In such cases, the file dictionary contains a code that converts the stored representation of numbers into a format suitable for output. For example, a file can store the number 123.45 as 12345. The file dictionary uses an MD2 conversion to insert the decimal point when the number is retrieved. You must ensure that BASIC variables sent to the data source contain the correct numeric values whether they are read from UniVerse files or not.

BASIC to SQL Character Types

No data conversion is necessary when converting BASIC character string data to the following ODBC SQL character data types:

SQL.CHARSQL.WCHAR
SQL.VARCHARSQL.WVARCHAR
SQL.LONGVARCHARSQL.WLONGVARCHAR

When the data is a BASIC subroutine, the subroutine name is used as the string value.

UniVerse Data Sources

No precision is checked. Data is sent to the server as a character string. Multivalued data is passed through to the server.

ODBC Data Sources

If the string is longer than the precision of the column as specified in the **SQLBindParameter** or SQLSetParam call, the BASIC SQL Client Interface returns SQL.ERROR, sets SQLSTATE to 01004. Other errors can be returned from the data source if the string exceeds the width of the column as defined in the data column.

BASIC to SQL Binary Types

No data conversion is done when converting BASIC bit string data to the following ODBC SQL binary data types:

SQL.BITSQL.BINARY
SQL.VARBINARY
SQL.LONGVARBINARY

For SQL.BIT binding, the application should use only the values 0 and 1. ODBC drivers vary as to how they handle other values. Some may return SQL.ERROR, others may return SQL.SUCCESS.WITH.INFO and deliver the integer part of a fractional number.

UniVerse Data Sources

No precision is checked. Data is sent to the server as a raw bit string. Multivalued data cannot be sent to the server.

ODBC Data Sources

If the string is longer than the precision of the column as specified in the **SQLBindParameter** or SQLSetParam call, the BASIC SQL Client Interface returns SQL.ERROR, sets SQLSTATE to 01004. Other errors can be returned from the data source if the string exceeds the width of the column as defined in the data column.

BASIC to SQL.DECIMAL and SQL.NUMERIC

If a BASIC string contains invalid nonnumeric characters, the BASIC SQL Client Interface returns SQL.ERROR and sets SQLSTATE to 22005.

UniVerse Data Sources

Precision and scale are not checked. Data is sent to the server as a character string.

ODBC Data Sources

If the string's precision and scale are consistent with the arguments of the **SQLBindParameter** or **SQLSetParam** call, the string is sent to the ODBC driver for further checks.

If the number of significant digits (digits to the left of the decimal point) is greater than the specified precision, SQL.ERROR is returned with an SQLSTATE of 22003.

If the number of insignificant digits (digits to the right of the decimal point) is greater than the column's defined scale, SQL.SUCCESS.WITH.INFO is returned with an SQLSTATE of 01004, indicated that fractional truncation has occurred.

BASIC to SQL Integer Types

The ODBC SQL integer types are:

SQL.INTEGERSQL.SMALLINT
SQL.BIGINTSQL.TINYINT

If a BASIC string is nonnumeric, the BASIC SQL Client Interface returns SQL.ERROR and sets SQLSTATE to 22005.

UniVerse Data Sources

No precision is checked. Data is sent to the server as a character string.

ODBC Data Sources

If the string represents a number that falls outside the limits of the SQL data type of the column, SQL.ERROR is returned with an SQLSTATE of 22003.

Some data sources such as SYBASE treat the TINYINT data type as unsigned, others treat it as signed. Use SQLGetTypeInfo to determine whether the type is signed or unsigned.

BASIC to SQL.REAL, SQL.FLOAT, and SQL.DOUBLE

If a BASIC string is nonnumeric, the BASIC SQL Client Interface returns SQL.ERROR and sets SQLSTATE to 22005.

UniVerse Data Sources

No precision is checked. Data is sent to the server as a character string.

ODBC Data Sources

These data types are passed directly to the ODBC driver and the data source for handling. Drivers and data sources vary as to how they handle numbers trying to represent too many digits in the number. Generally data sources represent 15 digits of precision. Digits beyond 15 may be lost, perhaps silently.

BASIC to SQL.DATE

The **SQLBindParameter** call can contain SQL.B.BASIC or SQL.B.INTDATE as the BASIC data type. If a BASIC string is a date in external format, use SQL.B.BASIC. If the string is a date in internal format, use SQL.B.INTDATE.

If you specify a BASIC data type of SQL.B.INTDATE and the ODBC SQL data type is not SQL.DATE, the BASIC SQL Client Interface returns an error and sets SQLSTATE to 07006. If the date is invalid, the BASIC SQL Client Interface returns SQL.ERROR and sets SQLSTATE to 22008.

UniVerse Data Sources

Dates are sent to UniVerse data sources in internal format. Dates already in internal format (SQL.B.INTDATE) need no conversion. The BASIC SQL Client Interface accepts dates in any valid UniVerse external format, including the ODBC format:

yyyy-mm-dd

ODBC Data Sources

Dates are sent to ODBC data sources in the following format:

yyyy-mm-dd

Dates already in external format (SQL.B.BASIC) need no conversion. The BASIC SQL Client Interface accepts dates in any valid UniVerse external format and converts them to the correct format for the ODBC driver.

BASIC to SQL.TIME

The SQLBindParameter call can contain SQL.B.BASIC or SQL.B.INTTIME as the BASIC data type. If the string is a time in external format (*hh*:*mm*:*ss*), use SQL.B.BASIC. If the string is a time in internal format, use SQL.B.INTTIME.

All times in external format (SQL.B.BASIC) must be specified using the following format:

[*h*]*h*:[*m*]*m*:[*s*]*s*

If the hour value is greater than 24, the value is divided by 24 and the remainder is used for the hour. So 50:01:02 is equivalent to 02:01:02. The minutes and seconds values must be from 0 to 59; if they are not, SQL.ERROR is returned and SQLSTATE is set to 22008.

When sending times to a UniVerse data source, the MTS conversion code converts times in external format to internal format.

If you specify a BASIC data type of SQL.B.INTTIME, the value is interpreted as the number of seconds since midnight. The value should be from 0 (midnight) to 86399. On UniVerse data sources, values outside this range can produce unusual results. On ODBC data sources, if a value is outside this range, the BASIC SQL Client Interface returns SQL.ERROR and sets SQLSTATE to 22008.

Times in internal format (SQL.B.INTTIME) need no conversion when they are sent to a UniVerse data source.

BASIC to SQL.TIMESTAMP

The TIMESTAMP data type is available only for ODBC data sources.

The **SQLBindParameter** call should contain SQL.B.BASIC as the BASIC data type and SQL.TIMESTAMP as the SQL data type. The format for timestamp data is as follows:

yyyy-mm-dd hh:*mm*:*ss*

Be sure to supply both parts of this string, because ODBC drivers vary in how they handle a timestamp lacking a date or a time.

If either the date or the time is invalid, SQL.ERROR is returned and SQLSTATE is set to 22008.

Converting SQL Data to BASIC Data

Use the SQLBindCol call to specify the BASIC data type to which to convert incoming SQL data. This section describes how the BASIC SQL Client Interface converts incoming data.

You can specify six BASIC data types:

- SQL.B.CHARConverts to character string data
- SQL.B.BINARYConverts to bit string (raw) data
- SQL.B.NUMBERConverts to integer or double
- SQL.B.DEFAULTUses the ODBC SQL data type to determine how to convert
- SQL.B.INTDATEConverts a date to UniVerse internal date format
- SQL.B.INTTIMEConverts a time to UniVerse internal time format

Use SQL.B.INTDATE only with the SQL types DATE and TIMESTAMP; use SQL.B.INTTIME only with the SQL types TIME and TIMESTAMP. If you use them with any other SQL type, the SQLFetch function returns SQL.ERROR and sets SQLSTATE to 07006.

Unlike **SQLBindParameter**, which does not send truncated data to the data source, the **SQLFetch** function associated with **SQLBindCol** delivers rounded or truncated data to the BASIC program.

Conversions using the SQL.B.DEFAULT data type follow these rules:

- For the character string and bit string SQL data types, SQL.B.DEFAULT is equivalent to using SQL.B.CHAR.
- For the numeric SQL data types, SQL.B.DEFAULT is equivalent to specifying SQL.B.NUMBER at the **SQLBindCol** call. Data is stored in either integer or double form.
- For UniVerse data sources, SQL.B.DEFAULT is always equivalent to SQL.B.CHAR.

UniVerse files sometimes use an internal format to store DECIMAL and NUMERIC data types. In such cases, the file dictionary contains a code that converts the stored representation of the numbers into a format suitable for output. For example, a file can store the number 123.45 as 12345. The file dictionary uses an MD2 conversion to insert the decimal point when the number is retrieved. For data coming from the data source, the BASIC program may have to use the ICONV function to convert incoming data to the proper UniVerse internal storage format.

UniVerse servers return data to the client in a stripped external format. For example, if a money column has a conversion code of MD22$, , the value $4.50 is stored as the integer 450. The UniVerse server returns the value 4.50 to a bound column, which is the correct numeric value in external format, stripped of any text-formatting symbols such as $ (dollar sign) or , (comma).

Converting SQL Character Types to BASIC Data Types

There are six ODBC SQL character data types:

- SQL.CHAR
- SQL.VARCHAR
- SQL.LONGVARCHAR
- SQL.WCHAR
- SQL.WVARCHAR
- SQL.WLONGVARCHAR

SQL Character Data Types to SQL.B.CHAR and SQL.B.DEFAULT

Data does not need to be converted. Space is allocated and the string is stored in the BASIC datum. Trailing spaces are deleted. Multivalued data is passed through with value marks.

SQL Character Data Types to SQL.B.NUMBER

Nonnumeric SQL data returns SQL.ERROR and sets SQLSTATE to 22005.

Numeric SQL data is rounded to 15 digits. Multivalued data is transferred from UniVerse data sources as SQL.B.CHAR (see next section). If the number of significant digits (excluding trailing zeros) exceeds 15, the BASIC SQL Client Interface returns SQL.ERROR and sets SQLSTATE to 01004. If there is a fractional part and the number of digits without trailing zeros exceeds 15, SQLSTATE is set to 22001. Otherwise SQL.SUCCESS is returned.

SQL Data	BASIC Result	SQLSTATE
JONES	----	22005
123456789	123456789	00000
1234567890123456	----	01004
123456789012345	123456789012345	00000
123456789012345000	123456789012345000	00000
1.2e18	1200000000000000000	00000
123e-11	0.00000000123	00000
1234.567890123456789	1234.56789012346	22001
123456789012345.1	123456789012345	22001
12345678901234.1	12345678901234.1	00000
12345678901234.6	12345678901234.6	00000
12345678901234.567	12345678901234.6	22001

SQLSTATEs

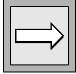
Converting SQL Binary Types to BASIC Data Types

There are three ODBC SQL binary data types:

- SQL.BINARY
- SQL.VARBINARY
- SQL.LONGVARBINARY

SQL Binary Data Types to SQL.B.BINARY and SQL.B.DEFAULT

Raw data is not converted. Space is allocated and the bit string is stored in the BASIC datum.



Note*: Multivalued data is passed through with value marks, but because the data is passed as a bit string, the value marks cannot be distinguished as meaningful delimiters. To fetch multivalued bit string data from a UniVerse data source, use dynamic normalization (see the UniVerse SQL Reference).*

Converting SQL Numeric Types to BASIC Data Types

There are seven ODBC SQL numeric data types:

- SQL.DECIMAL
- SQL.NUMERIC
- SQL.SMALLINT
- SQL.INTEGER
- SQL.REAL
- SQL.FLOAT
- SQL.DOUBLE

SQL Numeric Types to SQL.B.CHAR

The number is put in the BASIC variable in ASCII format.

Data returned from a UniVerse data source does not need to be converted. Space is allocated and the string is stored in the BASIC datum. Trailing spaces are deleted. Multivalued data is passed through with value marks.

SQL Numeric Types to SQL.B.NUMBER and SQL.B.DEFAULT

SMALLINT and INTEGER types are stored as BASIC integers. All others are stored as doubles.

SQL Data	SQL Type	BASIC Result	SQLSTATE
12345	SMALLINT	12345	00000
123456789	INTEGER	123456789	00000
123456789.	FLOAT	123456789.	00000
12345678901234567.25	DOUBLE	12345678901234600	00000
1234.37218738172312	DOUBLE	1234.3721873817	00000

Data returned from a UniVerse data source to SQL.B.DEFAULT does not need to be converted. Space is allocated and the string is stored in the BASIC datum. Trailing spaces are deleted. If the conversion results in an integer part loss, the BASIC SQL Client Interface returns SQL.ERROR and sets SQLSTATE to 01004. If the conversion results in a fractional part loss, SQL.ERROR is returned and SQLSTATE is set to 22001.

Multivalued numeric data returned from a UniVerse data source to SQL.B.NUMBER is transferred as a dynamic array. Values are delimited by value marks. No conversion or data checking occurs. SUCCESS.WITH.INFO is returned, and SQLSTATE indicates that the data is multivalued and a single result was returned.

Converting SQL Date, Time, and Timestamp Types to BASIC Types

The BASIC SQL Client Interface returns an SQL date or time in two ways:

- From UniVerse servers, as a character string or as an internal date or time
- From ODBC servers, as a character string

You cannot convert any of these SQL data types to SQL.B.NUMBER. If you try, **SQLFetch** generates SQL.ERROR and sets SQLSTATE to 07006.

SQL DATE Data to SQL.B.INTDATE

UniVerse Data Sources

No conversion is required, because dates are transferred in internal format.

ODBC Data Sources

The BASIC SQL Client Interface accepts dates in external format and converts them to dates in internal format.

SQL DATE and TIME Data to SQL.B.CHAR and SQL.B.DEFAULT

Dates from both UniVerse and ODBC data sources are returned in the following format:

yyyy-mm-dd

Times from both UniVerse and ODBC data sources are returned in the following format:

hh:*mm*:*ss*

If the date or time is not valid, **SQLFetch** returns SQL.ERROR and sets SQLSTATE to 22008.

SQL TIME Data to SQL.B.INTTIME

Times from both UniVerse and ODBC data sources are converted to UniVerse times in internal format through the MTS conversion code. The resulting integer is the data returned in the bound variable.

SQL TIMESTAMP Data to SQL.B.CHAR and SQL.B.DEFAULT

The TIMESTAMP data type is available only from ODBC data sources. Timestamp data is returned in the following formats:

> *yyyy-mm-dd*
>
> *hh*:*mm*:*ss*

SQL TIMESTAMP Data to SQL.B.INTDATE and SQL.B.INTTIME

The date part of a TIMESTAMP value is converted to a UniVerse date in internal format through a D2 conversion code. The time part of a TIMESTAMP value is converted to a UniVerse time in internal format via an MTS conversion code.

The resulting integer is the data returned in the bound variable.

Appendix B

SQL Client Interface Demonstration Program

This appendix describes a demonstration program that shows how to use the SQL Client Interface. The program does the following:

- Gathers information to log on to a data source
- Connects to the data source
- Creates a local UniVerse table and populates it with data
- Drops and creates the tables on the data source
- Reads the UniVerse file and inserts the data into the data source table
- Selects the file from the data source and displays it on the screen

The demonstration program is called SQLBCIDEMO. It is in the BP file of the UV account. For information about how to run the Running the Demonstration Program, see Chapter 2, "Getting Started."

Main Program

First the program includes the SQL Client Interface definitions from the ODBC.H file:

```
* Include the ODBC definitions

$INCLUDE UNIVERSE.INCLUDE ODBC.H

form = "T##########"
dash =  "----------"
Expect = ""
```

The program creates an ODBC environment and a connection environment. Use the SQLSetConnectOption call to specify the operating system user name and password.

```
STATUS = SQLAllocEnv(DBCENV)
IF STATUS <> SQL.SUCCESS THEN STOP "Failed to allocate an ODBC
environment"

STATUS = SQLAllocConnect(DBCENV,CONENV)
IF STATUS <> SQL.SUCCESS THEN STOP "Failed to allocate a CONNECT
environment"
```

The next section gathers the name of the data source, the user name and password for the server operating system, and information for the data source. The DBMS information is often a user name and a password.

```
PRINT "Please enter the target data source ":
INPUT SOURCE
UID=""
PWD=""
gosub testodbc

if toodbc = 0
then
    PRINT "Please enter the username for the server operating
system login ":
    INPUT OSUID
    PRINT "Please enter the operating system password for user
":OSUID:" ":
    ECHO OFF
    INPUT OSPWD
    PRINT ""
    ECHO ON
    PRINT "Enter name or path of remote schema/account (hit return
if local)":
    INPUT UID
    PWD = ""
    PRINT "";PRINT ""
```

```
STATUS = SQLSetConnectOption(CONENV, SQL.OS.UID, OSUID)
    STATUS = SQLSetConnectOption(CONENV, SQL.OS.PWD, OSPWD)
end
else if toodbc = 1
then
    PRINT "Enter the first DBMS connection parameter: ":
    input UID
    PRINT "Enter the second DBMS connection parameter: ":
    echo off
    input PWD
    echo on
    PRINT "";PRINT ""
end
```

The following lines make a connection to the data source:

```
PRINT "Connecting to data source: ": SOURCE
Fn = "SQLConnect"
STATUS = SQLConnect(CONENV,SOURCE,UID,PWD)
GOSUB CKCONENV
```

After making the connection, the program creates some local UniVerse files and loads them with data:

```
gosub CREATEFILES
gosub LOADFILES
```

The following lines create an SQL statement environment for executing SQL statements:

```
Fn = "SQLAllocStmt"
STATUS = SQLAllocStmt(CONENV,STMTENV)
GOSUB CKCONENV
```

Next the program creates some SQL tables on the data source, then loads the tables with data by reading records from UniVerse files:

```
gosub CREATETABLES
gosub LOADTABLES
```

After loading the SQL tables, the program reads them back and displays them:

```
gosub SELECTFILES
```

The following lines free up the statement, connection, and ODBC environments, and exit the program:

```
Fn = "SQLFreeStmt"
STATUS = SQLFreeStmt(STMTENV, SQL.DROP)
GOSUB CKSTMTENV

Fn = "SQLDisconnect"
STATUS = SQLDisconnect(CONENV)
GOSUB CKCONENV

Fn = "SQLFreeConnect"
STATUS = SQLFreeConnect(CONENV)
GOSUB CKCONENV

Fn = "SQLFreeEnv"
STATUS = SQLFreeEnv(DBCENV)
IF STATUS <> SQL.SUCCESS THEN STOP "Failed to release ODBC
environment"

STOP "Exiting SQLBCIDEMO"
```

Creating Local UniVerse Files

The following subroutine creates a set of local UniVerse files. These files contain data to be uploaded into a data source.

```
CREATEFILES:

CREATE.STAFF = "CREATE.FILE SQLCOSTAFF 2 1 1"
DIM DICT(8)
f = @FM
DICT(2) =  "EMPNUM":
f:"D":f:0:f:f:f:"10L":f:"S":f:f:"CHARACTER,10":f
DICT(3) =  "EMPNAME":
f:"D":f:1:f:f:f:"10L":f:"S":f:f:"CHARACTER,10":f
DICT(4) =
"EMPGRADE":f:"D":f:2:f:"MD0":f:f:"10R":f:"S":f:f:"INTEGER":f
DICT(5) =  "EMPCITY":
f:"D":f:3:f:f:f:"15L":f:"S":f:f:"CHARACTER,15":f
DICT(6) =  "@REVISE": f: "PH":f:f:f:f:f:f:f:f
DICT(7) =  "@":f:"PH":f:"ID.SUP EMPNUM EMPNAME EMPGRADE
EMPCITY":f:f:f:f:f:f:f
DICT(8) =  "@KEY":f:"PH":f:"EMPNUM":f:f:f:f:f:f:f
```

First the program creates a table in the UniVerse account:

```
OPEN "SQLCOSTAFF" TO STAFFVAR THEN
  CLOSE STAFFVAR
  PRINT "Deleting local SQLCOSTAFF file"
  EXECUTE "DELETE.FILE SQLCOSTAFF"
  PRINT ""
END

EXECUTE CREATE.STAFF
PRINT ""
```

Now the program populates the dictionary with definitions that would have been put in with the following SQL statement:

```
*  CREATE TABLE SQLCOSTAFF (TYPE 2, MODULO 1, SEPARATION 1,
*  EMPNUM CHAR(10) NOT NULL PRIMARY KEY,
*  EMPNAME CHAR(10), EMPGRADE INTEGER, EMPCITY CHAR(15) );

OPEN "DICT", "SQLCOSTAFF"  TO STAFFVAR ELSE STOP "Failed to open
DICT SQLCOSTAFF"
REC = ""
FOR INDEX = 2 TO 8
  ID = DICT(INDEX)<1>
  FOR I = 2 TO 9
    REC<I-1> = DICT(INDEX)<I>
```

```
  NEXT I
  WRITE REC TO STAFFVAR, ID
NEXT INDEX

CLOSE STAFFVAR
RETURN
```

Inserting Data into Local UniVerse Tables

The following subroutine inserts data into a set of local UniVerse tables:

```
LOADFILES:

*
* Setup data to insert into UniVerse tables and data source'
tables
*

DIM EMPDATA(5)
EMPDATA(1) = "E1":@FM:"Alice":@FM: 12:@FM:"Deale"
EMPDATA(2) = "E2":@FM:"Betty":@FM: 10:@FM:"Vienna"
EMPDATA(3) = "E3":@FM:"Carmen":@FM: 13:@FM:"Vienna"
EMPDATA(4) = "E4":@FM:"Don":@FM: 12:@FM:"Deale"
EMPDATA(5) = "E5":@FM:"Ed":@FM: 13:@FM:"Akron"

*
* Clear the files and then load them up
*

EXECUTE "CLEAR.FILE SQLCOSTAFF"
OPEN "SQLCOSTAFF"  TO STAFFVAR ELSE STOP "Failed to open
SQLCOSTAFF File"
FOR INDEX = 1 TO 5
  REC = EMPDATA(INDEX)
  ID = REC<1>
  DREC = REC<2>:@FM:REC<3>:@FM:REC<4>
  WRITE DREC TO STAFFVAR, ID
NEXT INDEX
CLOSE STAFFVAR

RETURN
```

Creating Tables on the Data Source

The following subroutine creates tables on the chosen data source:

```
CREATETABLES:

* Create Table statement to build the test table. These are in
upper case
* because UniVerse systems are often case sensitive.  Because this
program
* can be run using the local server on UniVerse the table name on
the server
* must be different than the file name on the client.

CTBL1 = "CREATE TABLE TSQLCOSTAFF( EMPNUM CHAR(10) NOT NULL
PRIMARY KEY, EMPNAME CHAR(10), GRADE INT, CITY CHAR(15))"

* Drop table statements to alwasy drop the target table before re-
creating
* them.

DTBL1 = "DROP TABLE TSQLCOSTAFF"

* Now create the tables needed for testing on the host DBMS

PRINT "Dropping TSQLCOSTAFF table at ":SOURCE
Fn = "SQLExecDirect"; Expect = "S0002"
STATUS = SQLExecDirect(STMTENV, DTBL1)
GOSUB CKSTMTENV
Expect = ""

PRINT ""; PRINT "Creating TSQLCOSTAFF table at ":SOURCE
STATUS = SQLExecDirect(STMTENV,CTBL1)
GOSUB CKSTMTENV

RETURN
```

Inserting Data into the Data Source Table

The following subroutine loads data into the table on the data source. The parameter markers make it easy to load multiple rows of data.

```
LOADTABLES:

INST1 = "INSERT INTO TSQLCOSTAFF VALUES ( ?, ?, ?, ?)"

ROWNUM = 0
Fn = "SQLBindParameter"
PRINT ""; PRINT "Setting values for the parameter markers"

STATUS = SQLBindParameter(STMTENV, 1, SQL.B.BASIC, SQL.CHAR, 10,
0, EMPNUM)
GOSUB CKSTMTENV

STATUS = SQLBindParameter(STMTENV, 2, SQL.B.BASIC, SQL.CHAR, 10,
0, EMPNAME)
GOSUB CKSTMTENV

STATUS = SQLBindParameter(STMTENV, 3, SQL.B.BASIC, SQL.INTEGER, 0,
0, GRADE)
GOSUB CKSTMTENV

STATUS = SQLBindParameter(STMTENV, 4, SQL.B.BASIC, SQL.CHAR, 15,
0, CITY)
GOSUB CKSTMTENV

PRINT ""; PRINT "Prepare the SQL statement to load TSQLCOSTAFF
table"
Fn = "SQLPrepare"
STATUS = SQLPrepare(STMTENV, INST1)
GOSUB CKSTMTENV
```

Now the program opens the local UniVerse SQLCOSTAFF table, reads values from it, puts the values into the **SQLBindParameter** variables, and executes the prepared INSERT statement:

```
OPEN "SQLCOSTAFF" TO FILEVAR ELSE STOP "Failed to open SQLCOSTAFF
file"
SELECT FILEVAR

NEXTID:
ROWNUM = ROWNUM+1
READNEXT ID ELSE GOTO EOD1
READ REC FROM FILEVAR,ID  ELSE STOP "Error reading local
SQLCOSTAFF file"
EMPNUM = ID
EMPNAME = REC<1>
GRADE = REC<2>
```

```
CITY = REC<3>

PRINT "Loading row ":ROWNUM:" of SQLCOSTAFF"
Fn = "SQLExecute"
STATUS = SQLExecute(STMTENV)
GOSUB CKSTMTENV
GOTO NEXTID

EOD1:
CLOSE FILEVAR
ROWNUM = 0

RETURN
```

Selecting Data from the Data Source

The following subroutine selects data from the data source:

```
SELECTFILES:

* Select statements to retrieve data from sqlcostaff table

SEL01 = "SELECT EMPNUM, EMPNAME, GRADE, CITY FROM TSQLCOSTAFF"


* Now select the data back & list it on the terminal

PRINT "Execute a SELECT statement against the TSQLCOSTAFF table"
PRINT ""

Fn = "SQLExecDirect"
STATUS = SQLExecDirect(STMTENV,SEL01)
GOSUB CKSTMTENV

PRINT ""; PRINT "Bind columns to program variables"
Fn = "SQLBindCol"
STATUS = SQLBindCol(STMTENV, 1, SQL.B.CHAR, EMPNUM.RET)
GOSUB CKSTMTENV

STATUS = SQLBindCol(STMTENV, 2, SQL.B.CHAR, EMPNAME.RET)
GOSUB CKSTMTENV

STATUS = SQLBindCol(STMTENV, 3, SQL.B.NUMBER, GRADE.RET)
GOSUB CKSTMTENV

STATUS = SQLBindCol(STMTENV, 4, SQL.B.CHAR, CITY.RET)
GOSUB CKSTMTENV

PRINT "EMPNUM" form:"EMPNAME" form:"GRADE" form :"CITY" form
PRINT dash form:dash form:dash form :dash form
STATUS = 0
LOOP
Fn = "SQLFetch"
WHILE STATUS <> SQL.NO.DATA.FOUND DO
    STATUS = SQLFetch(STMTENV)
    GOSUB CKSTMTENV
    IF STATUS <> SQL.NO.DATA.FOUND
    THEN
      PRINT EMPNUM.RET form:EMPNAME.RET form:GRADE.RET
form:CITY.RET
    END
REPEAT

STATUS = SQLFreeStmt(STMTENV, SQL.UNBIND)
GOSUB CKSTMTENV
RETURN
```

Checking for Errors

The following two subroutines check to see if an error occurred. The first subroutine is used for calls issued when there is a connection environment but no SQL statement environment:

```
CKCONENV:

COUNT = -1
    IF STATUS EQ -2 THEN STOP "INVALID CONNECTION HANDLE"
    IF STATUS NE 0
    THEN
201*
      ST = SQLERROR(-1,CONENV,-1,STATE,NATCODE,ERRTXT)
      IF ST <> SQL.NO.DATA.FOUND
      THEN
        PRINT "****************************************"
        COUNT = 1
        IF Expect NE 0 AND STATE = Expect AND ST <>
SQL.NO.DATA.FOUND
        THEN
          PRINT "Allowed error of ":STATE:" returned for func ":Fn
          GOTO 299
        END
        ELSE
          PRINT "Status for ":Fn:" call is: ":STATUS
          PRINT "SQLSTATE,NATCOD are:" : STATE:"  ":NATCODE
          PRINT "Error text is"
          PRINT "    " : ERRTXT
        END
        IF ST = SQL.SUCCESS THEN GOTO 201
      END
    IF STATUS = -1 AND COUNT = 1 THEN STOP "EXITING CKCONENV"
    END
299*
    IF STATUS <> 0 THEN PRINT
"****************************************"
    RETURN
```

The second subroutine is used for calls issued when there is an SQL statement environment:

```
CKSTMTENV:
    IF STATUS EQ -2 THEN STOP "INVALID STATEMENT HANDLE"

    IF STATUS EQ 100 THEN RETURN
    IF STATUS NE 0
    THEN
301*
      ST = SQLERROR(-1,-1,STMTENV,STATE,NATCODE,ERRTXT)
      IF ST <> SQL.NO.DATA.FOUND
```

```
      THEN
        PRINT "****************************************"
        COUNT = 1
        IF Expect NE 0 AND STATE = Expect AND ST <>
SQL.NO.DATA.FOUND
        THEN
          PRINT "Allowed error of ":STATE:" returned for func ":Fn
          GOTO 399
        END
        ELSE
          PRINT "Status for ":Fn:" call is: ":STATUS
          PRINT "SQLSTATE,NATCOD are:" : STATE:"  ":NATCODE
          PRINT "Error text is "
          PRINT "    " : ERRTXT
        END
        IF ST = 0 THEN GOTO 301
      END
    IF STATUS = -1 AND COUNT = 1 THEN STOP "EXITING CKSTMTENV"
    END
399*
  IF STATUS <> 0 THEN PRINT
"****************************************"
  RETURN
```

Appendix

C

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

Error Codes

SQLSTATE	Message
22005	Nonnumeric data was found where numeric is required
22008	Illegal date/time value
23000	Integrity constraint violation
24000	Invalid cursor state
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	ODBC is not installed on the system
IM977	Multivalued parameter finding for CALL not allowed
IM978	SQLBindMvCol/SQLBindMvParam illegal on 1NF 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

Error Codes (Continued)

SQLSTATE	Message
IM982	Only a single environment variable can be allocated
IM983	Nested transactions to non-UniVerse databases not allowed
IM985	Error in RPC interface
IM986	Improper SQLTYPE option
IM987	Improper MAPERROR option
IM988	Row exceeds maximum allowable width
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
S1009	An illegal option value was specified
S1010	Function call is illegal at this point

Error Codes (Continued)

SQLSTATE	Message
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 Interface 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

Error Codes (Continued)

Appendix

D

UniVerse Extended Parameters

The following table lists data source and DBMS-type extended parameters with their current default settings.

Parameter Name	Description	Modifiable
AUTOINC	Set to YES if data source can return whether a column is an autoincrement column. Defaults:UNIVERSENO ODBC*Not used in this release.*	No
BIGINTPREC	Precision for SQL_BIGINT data. Represents the number of digits and SQL_BIGINT value can have. Defaults:UNIVERSEN/A ODBC38	
CASE	Set to YES if data source can determine if a column is case-sensitive to collations. Defaults:UNIVERSEYES ODBC*Not used in this release.*	No
DATEFORM	Date format. Converts dates between UniVerse and the format required by the data source. Defaults should be appropriate. Defaults:UNIVERSED–YMD[4,2,2] ODBC*Not used in this release.*	Yes

Extended Parameters

Parameter Name	Description	Modifiable
DATEPREC	Date field width. Controls how many characters of the complete DATE value are used when converting an SQL date to a date in internal UniVerse format. The DATE value from some systems may include timestamp information as well as date information. Defaults:UNIVERSEN/A ODBC38	Yes
DBLPREC	Precision for SQL_DOUBLE data. Represents the number of digits an SQL_DOUBLE value can have. Defaults:UNIVERSEN/A ODBC38	Yes
DSPSIZE	Set to YES if data source supplies a display size value. Defaults:UNIVERSEYES ODBC*Not used in this release.*	No
FLOATPREC	Precision for SQL_FLOAT data. Represents the number of a digits an SQL_FLOAT value can have. Defaults:UNIVERSEN/A ODBC38	Yes
INTPREC	Precision for SQL_INTEGER data. Represents the number of digits an SQL_INTEGER value can have. Defaults:UNIVERSEN/A ODBC38	Yes

Extended Parameters (Continued)

Parameter Name	Description	Modifiable
MAPERROR	Lets you map a data source error code to a BASIC SQL Client Interface error code. Common SQL Client Interface errors are already mapped. Defaults:UNIVERSE MAPERROR = 08001 = 930129 MAPERROR = 08004 = 930133 MAPERROR = 08004 = 930127 MAPERROR = 08004 = 930137 MAPERROR = 21S01 = 950059 MAPERROR = 21S02 = 950415 MAPERROR = 21S02 = 950417 MAPERROR = 22005 = 950043 MAPERROR = 22005 = 950121 MAPERROR = 22005 = 950122 MAPERROR = 22005 = 950169 MAPERROR = 22005 = 950617	Yes

Extended Parameters (Continued)

Parameter Name	Description	Modifiable
MAPERROR (continued)	MAPERROR = 23000 = 923012	Yes
	MAPERROR = 23000 = 923013	
	MAPERROR = 23000 = 950136	
	MAPERROR = 23000 = 950568	
	MAPERROR = 23000 = 950645	
	MAPERROR = 40000 = 040065	
	MAPERROR = 40000 = 909046	
	MAPERROR = 40000 = 950604	
	MAPERROR = 42000 = 001397	
	MAPERROR = 42000 = 001422	
	MAPERROR = 42000 = 001423	
	MAPERROR = 42000 = 001424	
	MAPERROR = 42000 = 020142	
	MAPERROR = 42000 = 036010	
	MAPERROR = 42000 = 950072	
	MAPERROR = 42000 = 950076	
	MAPERROR = 42000 = 950078	
	MAPERROR = 42000 = 950131	
	MAPERROR = 42000 = 950303	
	MAPERROR = 42000 = 950304	
	MAPERROR = 42000 = 950305	
	MAPERROR = 42000 = 950306	
	MAPERROR = 42000 = 950338	
	MAPERROR = 42000 = 950343	
	MAPERROR = 42000 = 950350	
	MAPERROR = 42000 = 950352	
	MAPERROR = 42000 = 950361	
	MAPERROR = 42000 = 950362	
	MAPERROR = 42000 = 950365	
	MAPERROR = 42000 = 950391	
	MAPERROR = 42000 = 950392	
	MAPERROR = 42000 = 950393	
	MAPERROR = 42000 = 950394	
	MAPERROR = 42000 = 950395	
	MAPERROR = 42000 = 950398	
	MAPERROR = 42000 = 950405	

Parameter Name	Description	Modifiable
MAPERROR (continued)	MAPERROR = 42000 = 950534	Yes
	MAPERROR = 42000 = 950538	
	MAPERROR = 42000 = 950539	
	MAPERROR = 42000 = 950540	
	MAPERROR = 42000 = 950541	
	MAPERROR = 42000 = 950546	
	MAPERROR = 42000 = 950548	
	MAPERROR = 42000 = 950563	
	MAPERROR = 42000 = 950588	
	MAPERROR = 42000 = 950590	
	MAPERROR = 42000 = 950607	
	MAPERROR = 42000 = 950609	
	MAPERROR = S0001 = 950458	
	MAPERROR = S0001 = 950459	
	MAPERROR = S0001 = 950528	
	MAPERROR = S0001 = 950529	
	MAPERROR = S0002 = 950311	
	MAPERROR = S0002 = 950313	
	MAPERROR = S0002 = 950316	
	MAPERROR = S0002 = 950390	
	MAPERROR = S0002 = 950455	
	MAPERROR = S0002 = 950545	
	MAPERROR = S0002 = 950596	
	MAPERROR = S0002 = 950597	
	MAPERROR = S0002 = 950598	
	MAPERROR = S0002 = 950599	
	MAPERROR = S0021 = 950416	
	MAPERROR = S0021 = 950570	
	MAPERROR = S0022 = 950418	
	MAPERROR = S0022 = 950425	
	MAPERROR = S0022 = 950428	
	MAPERROR = S0022 = 950522	
	MAPERROR = S0022 = 950523	
	MAPERROR = S1008 = 50003	

Parameter Name	Description	Modifiable
	ODBC*None*	No
MARKERNAME	Set to YES if data source uses names instead of question marks for parameter markers. Defaults:UNIVERSEYES ODBCYES	No
MAXCHAR	Maximum width of the data source's character data type. Defaults:UNIVERSEN/A ODBC255	Yes
MAXFETCHBUFF	For UniVerse servers only. Maximum buffer size allocated 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. Default:UNIVERSE8192 bytes	Yes
MAXFETCHCOLS	For UniVerse servers only. Maximum number of column values the server can put in the buffer before sending it 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. Default:UNIVERSE400 column values	Yes
MAXLONGVARCHAR		Yes
	Maximum width of the data source's long character varying data type. Defaults:UNIVERSEN/A ODBC4,096	

Extended Parameters (Continued)

Parameter Name	Description	Modifiable
MAXVARCHAR	Maximum width of the data source's character varying data type. Defaults:UNIVERSEN/A ODBC255	Yes
MONEY	Set to YES if data source reports if a column is a money data type. Defaults:UNIVERSENO ODBCNO	No
NLSMAP	For UniVerse servers only. Name of the server's NLS map table.	Yes
NLSLCALL	For UniVerse servers only. Slash-separated list of five locales to use for each of the five locale categories.	Yes
NLSLCCOLLATE	For UniVerse servers only. Name of the locale whose sort order to use.	Yes
NLSLCCTYPE	For UniVerse servers only. Name of the locale whose character type to use.	Yes
NLSLOCALE	For UniVerse servers only. Name of the locale to use for all locale categories.	Yes
NLSLCMONETARY	For UniVerse servers only. Name of the locale whose monetary conventions to use.	Yes
NLSLCNUMERIC	For UniVerse servers only. Name of the locale whose numeric conventions to use.	Yes
NLSLCTIME	For UniVerse servers only. Name of the locale whose time conventions to use.	Yes

Extended Parameters (Continued)

Parameter Name	Description	Modifiable
NULLABLE	Set to YES if data source reports how a column handles SQL null values. Defaults:UNIVERSEYES ODBCYES	No
PRECISION	Set to YES if data source reports a column's precision. Defaults:UNIVERSEYES ODBCYES	No
REALPREC	Precision for SQL_REAL data. Represents the number of digits an SQL_REAL value can have. Defaults:UNIVERSEN/A ODBC38	Yes
SCALE	Set to YES if data source reports a column's scale. Defaults:UNIVERSEYES ODBCYES	No
SEARCH	Set to YES if data source can report a column's disposition in a WHERE clause. Defaults:UNIVERSEYES ODBC*Not used in this release.*	No
SMINTPREC	Precision for SQL_SMALLINT data. Represents the number of digits an SQL_SMALLINT value can have. Defaults:UNIVERSEN/A ODBC38	Yes

Extended Parameters (Continued)

Parameter Name	Description	Modifiable
TYPENAME	Set to YES if data source can report the data source's data type name for a column. Defaults:UNIVERSEYES ODBCYES	No
UNSIGNED	Set to YES if data source can report if a column is unsigned. Defaults:UNIVERSEYES ODBCYES	No
UPDATE	Set to YES if data source can report if a column is updatable. Defaults:UNIVERSEYES ODBCYES	No

Extended Parameters (Continued)

The parameters in the next table control internal aspects of the software's behavior with respect to transaction control. Do not modify them without specific instructions to do so from IBM support personnel.

Transaction Parameters

Parameter Name	Description		
TXBEHAVIOR	Defaults:	UNIVERSE ODBC	1 1
TXCOMMIT	Defaults:	UNIVERSE ODBC	empty empty
TXROLL	Defaults:	UNIVERSE ODBC	empty empty
TXSTART	Defaults:	UNIVERSE ODBC	empty empty
USETGITX	Defaults:	UNIVERSE ODBC	NO NO

Parameter Name	Description		
TXBEHAVIOR	Defaults:	UNIVERSE	1
		ODBC	1
TXCOMMIT	Defaults:	UNIVERSE	empty
		ODBC	empty
TXROLL	Defaults:	UNIVERSE	empty
		ODBC	empty
TXSTART	Defaults:	UNIVERSE	empty
		ODBC	empty
USETGITX	Defaults:	UNIVERSE	NO
		ODBC	NO

Transaction Parameters

Appendix

The ODBC.H File

E

This appendix lists the contents of the ODBC.H file. The ODBC.H file defines the values of column attributes.

```
*******************************************************
******************
*
*   Header file for ODBC BASIC programs
*
*     Module  %M%      Version %I%         Date    %H%
*
*       (c) Copyright 2005 IBM Corporation. - All
Rights Reserved
*       This is unpublished proprietary source code of
Ardent Software Inc.
*       The copyright notice above does not evidence
any actual or intented
*       publication of such source code.
*
*******************************************************
******************

* SQL Error RETCODES and defines.

EQU SQL.ERROR                             TO
-1
EQU SQL.INVALID.HANDLE                    TO
-2
EQU SQL.NEED.DATA                         TO
99
EQU SQL.NO.DATA.FOUND                     TO
100
EQU SQL.SUCCESS                          TO
0
EQU SQL.SUCCESS.WITH.INFO                 TO
1

EQU SQL.NULL.HENV                         TO
-1
EQU SQL.NULL.HDBC                         TO
-1
```

```
EQU SQL.NULL.HSTMT                       TO               -1

EQU SQL.NULL.DATA                        TO               -1

* SQLColAttributes defines

EQU SQL.COLUMN.COUNT                     TO                1
EQU SQL.COLUMN.NAME                      TO                2
EQU SQL.COLUMN.TYPE                      TO                3
EQU SQL.COLUMN.LENGTH                    TO                4
EQU SQL.COLUMN.PRECISION                 TO                5
EQU SQL.COLUMN.SCALE                     TO                6
EQU SQL.COLUMN.DISPLAYSIZE               TO                7
EQU SQL.COLUMN.DISPLAY.SIZE              TO                7
EQU SQL.COLUMN.NULLABLE                  TO                8
EQU SQL.COLUMN.UNSIGNED                  TO                9
EQU SQL.COLUMN.MONEY                     TO               10
EQU SQL.COLUMN.UPDATABLE                 TO               11
EQU SQL.COLUMN.AUTO.INCREMENT            TO               12
EQU SQL.COLUMN.CASE.SENSITIVE            TO               13
EQU SQL.COLUMN.SEARCHABLE                TO               14
EQU SQL.COLUMN.TYPE.NAME                 TO               15
EQU SQL.COLUMN.TABLE.NAME                TO               16
EQU SQL.COLUMN.OWNER.NAME                TO               17
EQU SQL.COLUMN.QUALIFIER.NAME            TO               18
EQU SQL.COLUMN.LABEL                     TO               19
EQU SQL.COLUMN.MULTIVALUED               TO             1001
EQU SQL.COLUMN.FORMAT                    TO             1002
EQU SQL.COLUMN.CONVERSION                TO             1003
EQU SQL.COLUMN.PRINT.RESULT              TO             1004

* SQLColAttributes subdefines for SQL.COLUMN.UPDATABLE

EQU SQL.ATTR.READONLY                    TO                0
EQU SQL.ATTR.WRITE                       TO                1
EQU SQL.ATTR.READWRITE.UNKNOWN           TO                2

* SQLColAttributes subdefines for SQL.COLUMN.SEARCHABLE

EQU SQL.UNSEARCHABLE                     TO                0
EQU SQL.LIKE.ONLY                        TO                1
EQU SQL.ALL.EXCEPT.LIKE                  TO                2
EQU SQL.SEARCHABLE                       TO                3

* SQLSetConnectOption defines

EQU SQL.AUTOCOMMIT                       TO              102
EQU SQL.USE.ODBC.PRECISION               TO              999
EQU SQL.TRUNC.ROUND                      TO              998
EQU SQL.SEND.TRUNC.ROUND                 TO              997
EQU SQL.OS.UID                           TO              996
EQU SQL.OS.PWD                           TO              995
EQU SQL.DATEFORM                         TO              994
EQU SQL.DATEPREC                         TO              993
```

```
EQU SQL.AUTOCOMMIT.OFF                  TO              0
EQU SQL.AUTOCOMMIT.ON                   TO              1

EQU SQL.EMPTY.NULL                      TO           1003
EQU SQL.EMPTY.NULL.ON                   TO              1
EQU SQL.EMPTY.NULL.OFF                  TO              0

EQU SQL.TX.PRIVATE                      TO           1004
EQU SQL.TX.PRIVATE.ON                   TO              1
EQU SQL.TX.PRIVATE.OFF                  TO              0

EQU SQL.UVNLS.MAP                       TO           1005
EQU SQL.UVNLS.LOCALE                    TO           1006
EQU SQL.UVNLS.LC.TIME                   TO           1007
EQU SQL.UVNLS.LC.NUMERIC                TO           1008
EQU SQL.UVNLS.LC.MONETARY               TO           1009
EQU SQL.UVNLS.LC.CTYPE                  TO           1010
EQU SQL.UVNLS.LC.COLLATE                TO           1011
EQU SQL.UVNLS.LC.ALL                    TO           1012

EQU SQL.UVNLS.SQL.NULL                  TO           1013
EQU SQL.UVNLS.TEXT.MARK                 TO           1014
EQU SQL.UVNLS.SUBVALUE.MARK             TO           1015
EQU SQL.UVNLS.VALUE.MARK                TO           1016
EQU SQL.UVNLS.FIELD.MARK                TO           1017
EQU SQL.UVNLS.ITEM.MARK                 TO           1018
EQU SQL.LIC.DEV.SUBKEY                  TO           1019


* SQLFreeStmt option defines

EQU SQL.CLOSE                           TO              1
EQU SQL.DROP                            TO              2
EQU SQL.UNBIND                          TO              3
EQU SQL.RESET.PARAMS                    TO              4

* Define all SQL data types
* and those that we support

EQU SQL.CHAR                            TO              1
EQU SQL.NUMERIC                         TO              2
EQU SQL.DECIMAL                         TO              3
EQU SQL.INTEGER                         TO              4
EQU SQL.SMALLINT                        TO              5
EQU SQL.FLOAT                           TO              6
EQU SQL.REAL                            TO              7
EQU SQL.DOUBLE                          TO              8
EQU SQL.DATE                            TO              9
EQU SQL.TIME                            TO             10
EQU SQL.TIMESTAMP                       TO             11
EQU SQL.VARCHAR                         TO             12
EQU SQL.LONGVARCHAR                     TO             -1
EQU SQL.BINARY                          TO             -2
```

```
EQU SQL.VARBINARY                        TO              -3
EQU SQL.LONGVARBINARY                    TO              -4
EQU SQL.BIGINT                           TO              -5
EQU SQL.TINYINT                          TO              -6
EQU SQL.BIT                              TO              -7
EQU SQL.WCHAR                            TO              -8
EQU SQL.WVARCHAR                         TO              -9
EQU SQL.WLONGVARCHAR                     TO              -10
EQU NUM.SQL.TYPES                        TO              22

* Define ODBC conception of display size
* for the various data types

EQU SQL.CHAR.DSPSIZE                     TO               0
EQU SQL.VARCHAR.DSPSIZE                  TO               0
EQU SQL.DECIMAL.DSPSIZE                  TO               2
EQU SQL.NUMERIC.DSPSIZE                  TO               2
EQU SQL.SMALLINT.DSPSIZE                 TO               6
EQU SQL.INTEGER.DSPSIZE                  TO              11
EQU SQL.REAL.DSPSIZE                     TO              13
EQU SQL.FLOAT.DSPSIZE                    TO              22
EQU SQL.DOUBLE.DSPSIZE                   TO              22
EQU SQL.DATE.DSPSIZE                     TO              10
EQU SQL.TIME.DSPSIZE                     TO               8

* Define ODBC conception of precision
* for the various data types

EQU SQL.CHAR.PRECISION                   TO             254
EQU SQL.VARCHAR.PRECISION                TO             254
EQU SQL.DECIMAL.PRECISION                TO              15
EQU SQL.NUMERIC.PRECISION                TO              15
EQU SQL.SMALLINT.PRECISION               TO               5
EQU SQL.INTEGER.PRECISION                TO              10
EQU SQL.REAL.PRECISION                   TO               7
EQU SQL.FLOAT.PRECISION                  TO              15
EQU SQL.DOUBLE.PRECISION                 TO              15
EQU SQL.DATE.PRECISION                   TO              10
EQU SQL.TIME.PRECISION                   TO               8

* Valid BASIC data types

EQU SQL.B.BASIC                          TO             100
EQU SQL.B.INTDATE                        TO             101
EQU SQL.B.NUMBER                         TO             102
EQU SQL.B.INTTIME                        TO             103
EQU SQL.B.CHAR                           TO               1
EQU SQL.B.DEFAULT                        TO              99

* Define return valued for
* Describe and ColAttributes

EQU SQL.NO.NULLS                         TO               0
EQU SQL.NULLABLE                         TO               1
```

```
EQU SQL.NULLABLE.UNKNOWN                TO              2

* Define parameter types for SQLBindParameter (SQLSetParam)

EQU SQL.PARAM.INPUT                     TO              1
EQU SQL.PARAM.INPUT.OUTPUT              TO              2
EQU SQL.PARAM.OUTPUT                    TO              4

* DTM Added for BCI/Datastage - SQLGetInfo

EQU SQL.ACTIVE.CONNECTIONS              TO              0
EQU SQL.ACTIVE.STATEMENTS               TO              1
EQU SQL.DATA.SOURCE.NAME                TO              2
EQU SQL.DRIVER.HDBC                     TO              3
EQU SQL.DRIVER.HENV                     TO              4
EQU SQL.DRIVER.HSTMT                    TO              5
EQU SQL.DRIVER.NAME                     TO              6
EQU SQL.DRIVER.VER                      TO              7
EQU SQL.FETCH.DIRECTION                 TO              8
EQU SQL.ODBC.API.CONFORMANCE            TO              9
EQU SQL.ODBC.VER                        TO             10
EQU SQL.ROW.UPDATES                     TO             11
EQU SQL.ODBC.SAG.CLI.CONFORMANCE        TO             12
EQU SQL.SERVER.NAME                     TO             13
EQU SQL.SEARCH.PATTERN.ESCAPE           TO             14
EQU SQL.ODBC.SQL.CONFORMANCE            TO             15
EQU SQL.DATABASE.NAME                   TO             16
EQU SQL.DBMS.NAME                       TO             17
EQU SQL.DBMS.VER                        TO             18
EQU SQL.ACCESSIBLE.TABLES               TO             19
EQU SQL.ACCESSIBLE.PROCEDURES           TO             20
EQU SQL.PROCEDURES                      TO             21
EQU SQL.CONCAT.NULL.BEHAVIOR            TO             22
EQU SQL.CURSOR.COMMIT.BEHAVIOR          TO             23
EQU SQL.CURSOR.ROLLBACK.BEHAVIOR        TO             24
EQU SQL.DATA.SOURCE.READ.ONLY           TO             25
EQU SQL.DEFAULT.TXN.ISOLATION           TO             26
EQU SQL.EXPRESSIONS.IN.ORDERBY          TO             27
EQU SQL.IDENTIFIER.CASE                 TO             28
EQU SQL.IDENTIFIER.QUOTE.CHAR           TO             29
EQU SQL.MAX.COLUMN.NAME.LEN             TO             30
EQU SQL.MAX.CURSOR.NAME.LEN             TO             31
EQU SQL.MAX.OWNER.NAME.LEN              TO             32
EQU SQL.MAX.PROCEDURE.NAME.LEN          TO             33
EQU SQL.MAX.QUALIFIER.NAME.LEN          TO             34
EQU SQL.MAX.TABLE.NAME.LEN              TO             35
EQU SQL.MULT.RESULT.SETS                TO             36
EQU SQL.MULTIPLE.ACTIVE.TXN             TO             37
EQU SQL.OUTER.JOINS                     TO             38
EQU SQL.OWNER.TERM                      TO             39
EQU SQL.PROCEDURE.TERM                  TO             40
EQU SQL.QUALIFIER.NAME.SEPARATOR        TO             41
EQU SQL.QUALIFIER.TERM                  TO             42
EQU SQL.SCROLL.CONCURRENCY              TO             43
```

```
EQU SQL.SCROLL.OPTIONS                  TO                44
EQU SQL.TABLE.TERM                      TO                45
EQU SQL.TXN.CAPABLE                     TO                46
EQU SQL.USER.NAME                       TO                47
EQU SQL.CONVERT.FUNCTIONS               TO                48
EQU SQL.NUMERIC.FUNCTIONS               TO                49
EQU SQL.STRING.FUNCTIONS                TO                50
EQU SQL.SYSTEM.FUNCTIONS                TO                51
EQU SQL.TIMEDATE.FUNCTIONS              TO                52
EQU SQL.CONVERT.BIGINT                  TO                53
EQU SQL.CONVERT.BINARY                  TO                54
EQU SQL.CONVERT.BIT                     TO                55
EQU SQL.CONVERT.CHAR                    TO                56
EQU SQL.CONVERT.DATE                    TO                57
EQU SQL.CONVERT.DECIMAL                 TO                58
EQU SQL.CONVERT.DOUBLE                  TO                59
EQU SQL.CONVERT.FLOAT                   TO                60
EQU SQL.CONVERT.INTEGER                 TO                61
EQU SQL.CONVERT.LONGVARCHAR             TO                62
EQU SQL.CONVERT.NUMERIC                 TO                63
EQU SQL.CONVERT.REAL                    TO                64
EQU SQL.CONVERT.SMALLINT                TO                65
EQU SQL.CONVERT.TIME                    TO                66
EQU SQL.CONVERT.TIMESTAMP               TO                67
EQU SQL.CONVERT.TINYINT                 TO                68
EQU SQL.CONVERT.VARBINARY               TO                69
EQU SQL.CONVERT.VARCHAR                 TO                70
EQU SQL.CONVERT.LONGVARBINARY           TO                71
EQU SQL.TXN.ISOLATION.OPTION            TO                72
EQU SQL.ODBC.SQL.OPT.IEF                TO                73
EQU SQL.CORRELATION.NAME                TO                74
EQU SQL.NON.NULLABLE.COLUMNS            TO                75
EQU SQL.DRIVER.HLIB                     TO                76
EQU SQL.DRIVER.ODBC.VER                 TO                77
EQU SQL.LOCK.TYPES                      TO                78
EQU SQL.POS.OPERATIONS                  TO                79
EQU SQL.POSITIONED.STATEMENTS           TO                80
EQU SQL.GETDATA.EXTENSIONS              TO                81
EQU SQL.BOOKMARK.PERSISTENCE            TO                82
EQU SQL.STATIC.SENSITIVITY              TO                83
EQU SQL.FILE.USAGE                      TO                84
EQU SQL.NULL.COLLATION                  TO                85
EQU SQL.ALTER.TABLE                     TO                86
EQU SQL.COLUMN.ALIAS                    TO                87
EQU SQL.GROUP.BY                        TO                88
EQU SQL.KEYWORDS                        TO                89
EQU SQL.ORDER.BY.COLUMNS.IN.SELECT      TO                90
EQU SQL.OWNER.USAGE                     TO                91
EQU SQL.QUALIFIER.USAGE                 TO                92
EQU SQL.QUOTED.IDENTIFIER.CASE          TO                93
EQU SQL.SPECIAL.CHARACTERS              TO                94
EQU SQL.SUBQUERIES                      TO                95
EQU SQL.UNION                           TO                96
EQU SQL.MAX.COLUMNS.IN.GROUP.BY         TO                97
```

```
EQU SQL.MAX.COLUMNS.IN.INDEX            TO              98
EQU SQL.MAX.COLUMNS.IN.ORDER.BY         TO              99
EQU SQL.MAX.COLUMNS.IN.SELECT           TO             100
EQU SQL.MAX.COLUMNS.IN.TABLE            TO             101
EQU SQL.MAX.INDEX.SIZE                  TO             102
EQU SQL.MAX.ROW.SIZE.INCLUDES.LONG      TO             103
EQU SQL.MAX.ROW.SIZE                    TO             104
EQU SQL.MAX.STATEMENT.LEN               TO             105
EQU SQL.MAX.TABLES.IN.SELECT            TO             106
EQU SQL.MAX.USER.NAME.LEN               TO             107
EQU SQL.MAX.CHAR.LITERAL.LEN            TO             108
EQU SQL.TIMEDATE.ADD.INTERVALS          TO             109
EQU SQL.TIMEDATE.DIFF.INTERVALS         TO             110
EQU SQL.NEED.LONG.DATA.LEN              TO             111
EQU SQL.MAX.BINARY.LITERAL.LEN          TO             112
EQU SQL.LIKE.ESCAPE.CLAUSE              TO             113
EQU SQL.QUALIFIER.LOCATION              TO             114

* SQL_ALTER_TABLE bitmasks *
EQU     SQL.AT.ADD.COLUMN               TO               1
EQU     SQL.AT.DROP.COLUMN              TO               2

* SQL_BOOKMARK_PERSISTENCE bitmasks *
EQU     SQL.BP.CLOSE                    TO               1
EQU     SQL.BP.DELETE                   TO               2
EQU     SQL.BP.DROP                     TO               4
EQU     SQL.BP.TRANSACTION              TO               8
EQU     SQL.BP.UPDATE                   TO              16
EQU     SQL.BP.OTHER.HSTMT              TO              32
EQU     SQL.BP.SCROLL                   TO              64

* SQL_CONCAT_NULL_BEHAVIOR values *
EQU     SQL.CB.NULL                     TO               0
EQU     SQL.CB.NON.NULL                 TO               1

* SQL_CURSOR_COMMIT_BEHAVIOR values *
* SQL_CURSOR_ROLLBACK_BEHAVIOR values *
EQU     SQL.CB.DELETE                   TO               0
EQU     SQL.CB.CLOSE                    TO               1
EQU     SQL.CB.PRESERVE                 TO               2

* SQL_CORRELATION_NAME values *
EQU     SQL.CN.NONE                     TO               0
EQU     SQL.CN.DIFFERENT                TO               1
EQU     SQL.CN.ANY                      TO               2

* SQL_CONVERT_<.> bitmasks *
EQU     SQL.CVT.CHAR                    TO               1
EQU     SQL.CVT.NUMERIC                 TO               2
EQU     SQL.CVT.DECIMAL                 TO               4
EQU     SQL.CVT.INTEGER                 TO               8
EQU     SQL.CVT.SMALLINT                TO              16
EQU     SQL.CVT.FLOAT                   TO              32
EQU     SQL.CVT.REAL                    TO              64
```

```
EQU     SQL.CVT.DOUBLE                TO              128
EQU     SQL.CVT.VARCHAR               TO              256
EQU     SQL.CVT.LONGVARCHAR           TO              512
EQU     SQL.CVT.BINARY                TO             1024
EQU     SQL.CVT.VARBINARY             TO             2048
EQU     SQL.CVT.BIT                   TO             4096
EQU     SQL.CVT.TINYINT               TO             8192
EQU     SQL.CVT.BIGINT                TO            16384
EQU     SQL.CVT.DATE                  TO            32768
EQU     SQL.CVT.TIME                  TO            65536
EQU     SQL.CVT.TIMESTAMP             TO           131072
EQU     SQL.CVT.LONGVARBINARY         TO           262144

* SQL_FETCH_DIRECTION bitmask *
EQU SQL.FD.FETCH.NEXT                 TO                1
EQU SQL.FD.FETCH.FIRST                TO                2
EQU SQL.FD.FETCH.LAST                 TO                4
EQU SQL.FD.FETCH.PRIOR                TO                8
EQU SQL.FD.FETCH.ABSOLUTE             TO               16
EQU SQL.FD.FETCH.RELATIVE             TO               32
EQU SQL.FD.FETCH.RESUME               TO               64
EQU SQL.FD.FETCH.BOOKMARK             TO              128

* SQL_FILE_USAGE values *
EQU     SQL.FILE.NOT.SUPPORTED        TO                0
EQU     SQL.FILE.TABLE                TO                1
EQU     SQL.FILE.QUALIFIER            TO                2

* SQL_CONVERT_FUNCTIONS bitmask *
EQU     SQL.FN.CVT.CONVERT            TO                1

* SQL_NUMERIC_FUNCTIONS bitmask *
EQU SQL.FN.NUM.ABS                    TO                1
EQU SQL.FN.NUM.ACOS                   TO                2
EQU SQL.FN.NUM.ASIN                   TO                4
EQU SQL.FN.NUM.ATAN                   TO                8
EQU SQL.FN.NUM.ATAN2                  TO               16
EQU SQL.FN.NUM.CEILING                TO               32
EQU SQL.FN.NUM.COS                    TO               64
EQU SQL.FN.NUM.COT                    TO              128
EQU SQL.FN.NUM.EXP                    TO              256
EQU SQL.FN.NUM.FLOOR                  TO              512
EQU SQL.FN.NUM.LOG                    TO             1024
EQU SQL.FN.NUM.MOD                    TO             2048
EQU SQL.FN.NUM.SIGN                   TO             4096
EQU SQL.FN.NUM.SIN                    TO             8192
EQU SQL.FN.NUM.SQRT                   TO            16384
EQU SQL.FN.NUM.TAN                    TO            32768
EQU SQL.FN.NUM.PI                     TO            65536
EQU SQL.FN.NUM.RAND                   TO           131072
EQU SQL.FN.NUM.DEGREES                TO           262144
EQU SQL.FN.NUM.LOG10                  TO           524288
EQU SQL.FN.NUM.POWER                  TO          1048576
EQU SQL.FN.NUM.RADIANS                TO          2097152
```

```
EQU SQL.FN.NUM.ROUND                    TO          4194304
EQU SQL.FN.NUM.TRUNCATE                 TO          8388608

* SQL_STRING_FUNCTIONS bitmask *
EQU SQL.FN.STR.CONCAT                   TO                1
EQU SQL.FN.STR.INSERT                   TO                2
EQU SQL.FN.STR.LEFT                     TO                4
EQU SQL.FN.STR.LTRIM                    TO                8
EQU SQL.FN.STR.LENGTH                   TO               16
EQU SQL.FN.STR.LOCATE                   TO               32
EQU SQL.FN.STR.LCASE                    TO               64
EQU SQL.FN.STR.REPEAT                   TO              128
EQU SQL.FN.STR.REPLACE                  TO              256
EQU SQL.FN.STR.RIGHT                    TO              512
EQU SQL.FN.STR.RTRIM                    TO             1024
EQU SQL.FN.STR.SUBSTRING                TO             2048
EQU SQL.FN.STR.UCASE                    TO             4096
EQU SQL.FN.STR.ASCII                    TO             8192
EQU SQL.FN.STR.CHAR                     TO            16384
EQU SQL.FN.STR.DIFFERENCE               TO            32768
EQU SQL.FN.STR.LOCATE.2                 TO            65536
EQU SQL.FN.STR.SOUNDEX                  TO           131072
EQU SQL.FN.STR.SPACE                    TO           262144

* SQL_SYSTEM_FUNCTIONS bitmask *
EQU SQL.FN.SYS.USERNAME                 TO                1
EQU SQL.FN.SYS.DBNAME                   TO                2
EQU SQL.FN.SYS.IFNULL                   TO                4

* SQL_TIMEDATE bitmask *
EQU SQL.FN.TD.NOW                       TO                1
EQU SQL.FN.TD.CURDATE                   TO                2
EQU SQL.FN.TD.DAYOFMONTH                TO                4
EQU SQL.FN.TD.DAYOFWEEK                 TO                8
EQU SQL.FN.TD.DAYOFYEAR                 TO               16
EQU SQL.FN.TD.MONTH                     TO               32
EQU SQL.FN.TD.QUARTER                   TO               64
EQU SQL.FN.TD.WEEK                      TO              128
EQU SQL.FN.TD.YEAR                      TO              256
EQU SQL.FN.TD.CURTIME                   TO              512
EQU SQL.FN.TD.HOUR                      TO             1024
EQU SQL.FN.TD.MINUTE                    TO             2048
EQU SQL.FN.TD.SECOND                    TO             4096
EQU SQL.FN.TD.TIMESTAMPADD              TO             8192
EQU SQL.FN.TD.TIMESTAMPDIFF             TO            16384
EQU SQL.FN.TD.DAYNAME                   TO            32768
EQU SQL.FN.TD.MONTHNAME                 TO            65536

* SQL_TIMEDATE_ADD_INTERVALS bitmask *
* SQL_TIMEDATE_DIFF_INTERVALS bitmask *
EQU SQL.FN.TSI.FRAC.SECOND              TO                1
EQU SQL.FN.TSI.SECOND                   TO                2
EQU SQL.FN.TSI.MINUTE                   TO                4
EQU SQL.FN.TSI.HOUR                     TO                8
```

```
EQU SQL.FN.TSI.DAY                       TO                 16
EQU SQL.FN.TSI.WEEK                      TO                 32
EQU SQL.FN.TSI.MONTH                     TO                 64
EQU SQL.FN.TSI.QUARTER                   TO                128
EQU SQL.FN.TSI.YEAR                      TO                256

* SQL_GROUP_BY values *
EQU     SQL.GB.NOT.SUPPORTED             TO                  0
EQU     SQL.GB.GROUP.BY.EQUALS.SELECT    TO                  1
EQU     SQL.GB.GROUP.BY.CONTAINS.SELECT TO                  2
EQU     SQL.GB.NO.RELATION               TO                  3

* SQL_GETDATA_EXTENSIONS values *
EQU     SQL.GD.ANY.COLUMN                TO                  1
EQU     SQL.GD.ANY.ORDER                 TO                  2
EQU     SQL.GD.BLOCK                     TO                  4
EQU     SQL.GD.BOUND                     TO                  8

* SQL_IDENTIFIER_CASE values *
* SQL_QUOTED_IDENTIFIER values *
EQU     SQL.IC.UPPER                     TO                  1
EQU     SQL.IC.LOWER                     TO                  2
EQU     SQL.IC.SENSITIVE                 TO                  3
EQU     SQL.IC.MIXED                     TO                  4

* SQL_LOCK_TYPES bitmask *
EQU     SQL.LCK.NO.CHANGE                TO                  1
EQU     SQL.LCK.EXCLUSIVE                TO                  2
EQU     SQL.LCK.UNLOCK                   TO                  4

* SQL_NULL_COLLATION values *
EQU     SQL.NC.HIGH                      TO                  0
EQU     SQL.NC.LOW                       TO                  1
EQU     SQL.NC.START                     TO                  2
EQU     SQL.NC.END                       TO                  4

* SQL_NON_NULLABLE_COLUMNS values *
EQU     SQL.NNC.NULL                     TO                  0
EQU     SQL.NNC.NON.NULL                 TO                  1

* SQL_ODBC_API_CONFORMANCE *
EQU     SQL.OAC.NONE                     TO                  0
EQU     SQL.OAC.LEVEL1                   TO                  1
EQU     SQL.OAC.LEVEL2                   TO                  2

* SQL_ODBC_SQL_CONFORMANCE values *
EQU     SQL.OSC.MINIMUM                  TO                  0
EQU     SQL.OSC.CORE                     TO                  1
EQU     SQL.OSC.EXTENDED                 TO                  2

* SQL_ODBC_SAG_CLI_CONFORMANCE values *
EQU     SQL.OSCC.NOT.COMPLIANT           TO                  0
EQU     SQL.OSCC.COMPLIANT               TO                  1
```

```
* SQL_OWNER_USAGE bitmask *
EQU     SQL.OU.DML.STATEMENTS          TO                 1
EQU     SQL.OU.PROCEDURE.INVOCATION    TO                 2
EQU     SQL.OU.TABLE.DEFINITION        TO                 4
EQU     SQL.OU.INDEX.DEFINITION        TO                 8
EQU     SQL.OU.PRIVILEGE.DEFINITION    TO                16

* SQL_POS_OPERATIONS *
EQU SQL.POS.POSITION                   TO                 1
EQU SQL.POS.REFRESH                    TO                 2
EQU SQL.POS.UPDATE                     TO                 4
EQU SQL.POS.DELETE                     TO                 8
EQU SQL.POS.ADD                        TO                16

* SQL_POSITIONED_STATEMENTS bitmask *
EQU     SQL.PS.POSITIONED.DELETE       TO                 1
EQU     SQL.PS.POSITIONED.UPDATE       TO                 2
EQU     SQL.PS.SELECT.FOR.UPDATE       TO                 4

* SQL_DEFAULT_TXN_ISOLATION bitmask *
* SQL_TXN_ISOLATION_OPTION bitmask *
EQU     SQL.TXN.READ.UNCOMMITTED       TO                 1
EQU     SQL.TXN.READ.COMMITTED         TO                 2
EQU     SQL.TXN.REPEATABLE.READ        TO                 4
EQU     SQL.TXN.SERIALIZABLE           TO                 8
EQU     SQL.TXN.VERSIONING             TO                16
EQU     SQL.TXN.CURRENT                TO                42

* SQL_QUALIFIER_LOCATION values *
EQU     SQL.QL.START                   TO                 1
EQU     SQL.QL.END                     TO                 2

* SQL_QUALIFIER_USAGE bitmask *
EQU     SQL.QU.DML.STATEMENTS          TO                 1
EQU     SQL.QU.PROCEDURE.INVOCATION    TO                 2
EQU     SQL.QU.TABLE.DEFINITION        TO                 4
EQU     SQL.QU.INDEX.DEFINITION        TO                 8
EQU     SQL.QU.PRIVILEGE.DEFINITION    TO                16

* SQL_SCROLL_CONCURRENCY bitmask *
EQU     SQL.SCCO.READ.ONLY             TO                 1
EQU     SQL.SCCO.LOCK                  TO                 2
EQU     SQL.SCCO.OPT.ROWVER            TO                 4
EQU     SQL.SCCO.OPT.VALUES            TO                 8

* SQL_SCROLL_OPTIONS bitmask *
EQU     SQL.SO.FORWARD.ONLY            TO                 1
EQU     SQL.SO.KEYSET.DRIVEN           TO                 2
EQU     SQL.SO.DYNAMIC                 TO                 4
EQU     SQL.SO.MIXED                   TO                 8
EQU     SQL.SO.STATIC                  TO                16

* SQL_STATIC_SENSITIVITY bitmask *
EQU     SQL.SS.ADDITIONS               TO                 1
```

```
EQU     SQL.SS.DELETIONS                TO                2
EQU     SQL.SS.UPDATES                  TO                4

* SQL_SUBQUERIES bitmask *
EQU SQL.SQ.COMPARISON                   TO                1
EQU SQL.SQ.EXISTS                       TO                2
EQU SQL.SQ.IN                           TO                4
EQU SQL.SQ.QUANTIFIED                   TO                8
EQU SQL.SQ.CORRELATED.SUBQUERIES        TO               16

* SQL_TXN_CAPABLE values *
EQU SQL.TC.NONE                         TO                0
EQU SQL.TC.DML                          TO                1
EQU SQL.TC.ALL                          TO                2
EQU SQL.TC.DDL.COMMIT                   TO                3
EQU SQL.TC.DDL.IGNORE                   TO                4

* SQL_UNION values *
EQU SQL.U.UNION                         TO                1
EQU SQL.U.UNION.ALL                     TO                2

* Additions for SQLSpecialColumns

EQU     SQL.BEST.ROWID                  TO                1
EQU     SQL.ROWVER                      TO                2
EQU     SQL.SCOPE.CURROW                TO                0
EQU     SQL.SCOPE.TRANSACTION           TO                1
EQU     SQL.SCOPE.SESSION               TO                2
EQU     SQL.PC.UNKNOWN                  TO                0
EQU     SQL.PC.PSEUDO                   TO                1
EQU     SQL.PC.NOT.PSEUDO               TO                2

* Additions for SQLStatistics
EQU     SQL.INDEX.UNIQUE                TO                0
EQU     SQL.INDEX.ALL                   TO                1
EQU     SQL.QUICK                       TO                0
EQU     SQL.ENSURE                      TO                1
EQU     SQL.TABLE.STAT                  TO                0
EQU     SQL.INDEX.CLUSTERED             TO                1
EQU     SQL.INDEX.HASHED                TO                2
EQU     SQL.INDEX.OTHER                 TO                3

* Additions for SQLParamOptions
EQU     SQL.PARAMOPTIONS.SET            TO                0
EQU     SQL.PARAMOPTIONS.READ           TO                1

* Additions for SQLTransact
EQU     SQL.COMMIT                      TO                1
EQU     SQL.ROLLBACK                    TO                2
```

Glossary

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 transactions are not logged. Each update to a database is committed immediately.

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**.

connection environment — Memory allocated and initialized with data necessary to describe and maintain a connection between the SQL Client Interface and the data source. Several connection environments can connect to a single ODBC environment.

cursor — A virtual pointer to the set of results produced by a query. The SQL Client Interface cursor points to one row of data at a time. An application can advance the cursor only one row at a time.

DDL — Data definition language. A subset of SQL statements used for creating, altering, and dropping schemas, tables, and views.

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.
data source	The data you want to access, the DBMS it is associated with, and the server the DBMS resides on. For example, a data source ORA could refer to an ORACLE DBMS on node SERVER1, with ORACLE SID ALPHA. ODBC data sources do not contain user ID and password information.
driver	A program that processes function calls, submits SQL requests to a specific data source, and returns results to the client application program.
driver manager	A program that loads and initializes drivers on behalf of a client application program.
dynamic normalization	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 global information concerning the state of the application, including the valid connection handles and the current active connection handle.
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.
middleware	Software that acts as a bridge between two different database systems. The UniRPC is middleware for UniVerse and UniData servers. Other servers use various ODBC drivers as middleware.
multivalued column	A column that can contain more than one value for each row in a table.

null value — A special value representing an unknown value. Not the same as 0 (zero), a blank, or an empty string.

ODBC — Open Database Connectivity. The Microsoft version of the SQL Access Group Call Level Interface (SAG CLI). Microsoft adds extensions to the SAG CLI that target it at the Windows marketplace.

ODBC driver — Software that acts as a bridge between a client system and a specific database system.

ODBC driver manager — Software that fields all entry points defined by the ODBC specification. It performs any common functions required by all databases, then passes function calls to specific ODBC drivers.

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 SQL data types.

prepared SQL statement — An SQL statement that has been checked before being passed to the server for execution.

programmatic SQL — 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.

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.

SAG — SQL access group. A consortium of database vendors exploring database interoperability.

SAG CLI — SQL access group call level interface. See **ODBC**.

scale — The maximum number of digits to the right of the decimal point.

single-valued column — A column that can contain only one value for each row in a table.

SQL Client Interface environment — Memory allocated and initialized with data necessary to describe all connections to data sources and all SQL statements being executed at those data sources. All connection environments and SQL statement environments are attached to an SQL Client Interface environment.

SQL statement environment	Memory allocated and initialized to let the software describe all context of an SQL statement executed at a data source. An SQL statement environment is attached to a connection environment.
UCI	UniCall Interface. A C application programming interface (API) that allows application programmers to write client application programs that use SQL function calls to access data in UniVerse databases.
UniRPC	Remote procedure call. A UniVerse middleware facility that receives requests from remote machines for services and that starts those services.
unirpc	The Windows service that handles calls to the server.
unirpcd	The UNIX daemon that handles calls to the server.
unirpcservices	The UniRPC services file, located on the server, which verifies client requests for services.
uvodbc.config	The client ODBC configuration file, which defines SQL client connections to a server in terms of DBMS, network, service, and host.
uvserver	The UniVerse server process to which the client application connects.