Adaptive Server® Anywhere
Programming Guide

ii

Contents

About This Manual

Subject

This book describes how to build and deploy database applications using the C, C++, and Java programming languages, as well as Visual Studio .NET. Users of tools such as Visual Basic and PowerBuilder can use the programming interfaces provided by those tools.

Audience

This manual is intended for application developers writing programs that work *directly* with one of the Adaptive Server Anywhere interfaces.

You do not need to read this manual if you are using a development tool such as PowerBuilder or Visual Basic, each of which has its own database interface on top of ODBC.

SQL Anywhere Studio documentation

This book is part of the SQL Anywhere documentation set. This section describes the books in the documentation set and how you can use them.

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

♦ **Introducing SQL Anywhere Studio**   This book provides an overview of the SQL Anywhere Studio database management and synchronization technologies. It includes tutorials to introduce you to each of the pieces that make up SQL Anywhere Studio.

♦ **What's New in SQL Anywhere Studio**   This book is for users of previous versions of the software. It lists new features in this and previous releases of the product and describes upgrade procedures.

♦ **Adaptive Server Anywhere Database Administration Guide**   This book covers material related to running, managing, and configuring databases and database servers.

♦ **Adaptive Server Anywhere SQL User's Guide**   This book describes how to design and create databases; how to import, export, and modify data; how to retrieve data; and how to build stored procedures and triggers.

♦ **Adaptive Server Anywhere SQL Reference**   This book provides a complete reference for the SQL language used by Adaptive Server Anywhere. It also describes the Adaptive Server Anywhere system tables and procedures.

♦ **Adaptive Server Anywhere Programming Guide**   This book describes how to build and deploy database applications using the C, C++, and Java programming languages. Users of tools such as Visual Basic and PowerBuilder can use the programming interfaces provided by those tools. It also describes the Adaptive Server Anywhere ADO.NET data provider.

♦ **Adaptive Server Anywhere Error Messages**   This book provides a complete listing of Adaptive Server Anywhere error messages together with diagnostic information.

♦ **SQL Anywhere Studio Security Guide**   This book provides information about security features in Adaptive Server Anywhere databases. Adaptive Server Anywhere 7.0 was awarded a TCSEC

(Trusted Computer System Evaluation Criteria) C2 security rating from the U.S. Government. This book may be of interest to those who wish to run the current version of Adaptive Server Anywhere in a manner equivalent to the C2-certified environment.

♦ **MobiLink Administration Guide**   This book describes how to use the MobiLink data synchronization system for mobile computing, which enables sharing of data between a single Oracle, Sybase, Microsoft or IBM database and many Adaptive Server Anywhere or UltraLite databases.

♦ **MobiLink Clients**   This book describes how to set up and synchronize Adaptive Server Anywhere and UltraLite remote databases.

♦ **MobiLink Server-Initiated Synchronization User's Guide**   This book describes MobiLink server-initiated synchronization, a feature of MobiLink that allows you to initiate synchronization from the consolidated database.

♦ **MobiLink Tutorials**   This book provides several tutorials that walk you through how to set up and run MobiLink applications.

♦ **QAnywhere User's Guide**   This manual describes MobiLink QAnywhere, a messaging platform that enables the development and deployment of messaging applications for mobile and wireless clients, as well as traditional desktop and laptop clients.

♦ **ODBC Drivers for MobiLink and Remote Data Access**   This manual describes how to use the iAnywhere Solutions ODBC drivers for Oracle, DB2, and Sybase Adaptive Server Enterprise. These ODBC drivers are included with SQL Anywhere Studio for use with MobiLink synchronization and Adaptive Server Anywhere remote data access.

♦ **SQL Remote User's Guide**   This book describes all aspects of the SQL Remote data replication system for mobile computing, which enables sharing of data between a single Adaptive Server Anywhere or Adaptive Server Enterprise database and many Adaptive Server Anywhere databases using an indirect link such as e-mail or file transfer.

♦ **Sybase Central Help**   This book includes the context-sensitive help for Sybase Central, Interactive SQL, and other graphical tools. It is not included in the printed documentation set.

♦ **Adaptive Server Anywhere SNMP Extension Agent User's Guide** This book covers material related to running and configuring the Adaptive Server Anywhere SNMP Extension Agent.

♦ **UltraLite Database User's Guide** This book is intended for all UltraLite developers. It introduces the UltraLite database system and provides information common to all UltraLite programming interfaces.

♦ **Native UltraLite for Java User's Guide** This manual describes Native UltraLite for Java. With Native UltraLite for Java you can develop and deploy database applications to Windows CE devices running the Jeode or CrEme VMs.

♦ **UltraLite Static Java User's Guide** This manual describes the UltraLite Static Java API. It is a complement for the *UltraLite Database User's Guide.*

♦ **UltraLite.NET User's Guide** This manual describes UltraLite.NET. With UltraLite.NET you can develop and deploy database applications to computers, or handheld, mobile, or embedded devices.

♦ **UltraLite for MobileVB User's Guide** This manual describes UltraLite for MobileVB. With UltraLite for MobileVB you can develop and deploy database applications to handheld, mobile, or embedded devices, running Palm OS or Windows CE.

♦ **UltraLite ActiveX User's Guide** This manual describes UltraLite ActiveX. With UltraLite ActiveX you can develop and deploy database applications to handheld, mobile, or embedded devices running Windows CE.

♦ **UltraLite C/C++ User's Guide** This manual describes UltraLite C and C++ programming interfaces. With UltraLite you can develop and deploy database applications to handheld, mobile, or embedded devices.

♦ **UltraLite for M-Business Anywhere User's Guide** This manual describes UltraLite for M-Business Anywhere. With UltraLite for M-Business Anywhere you can develop and deploy web-based database applications to handheld, mobile, or embedded devices, running Palm OS, Windows CE, or Windows XP.

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

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

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

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

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

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

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

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

Documentation conventions

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

Syntax conventions   The following conventions are used in the SQL syntax descriptions:

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

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

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

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

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

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

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

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

RELEASE SAVEPOINT [*savepoint-name*]

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

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

[**ASC** | **DESC**]

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

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

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

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

Graphic icons   The following icons are used in this documentation.

♦ A client application.
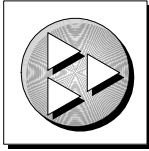
♦ A database server, such as Sybase Adaptive Server Anywhere.
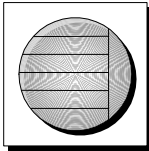
♦ A database. In some high-level diagrams, the icon may be used to
  represent both the database and the database server that manages it.
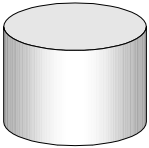
♦ Replication or synchronization middleware. These assist in sharing data
  among databases. Examples are the MobiLink Synchronization Server
  and the SQL Remote Message Agent.

♦ A programming interface.

API
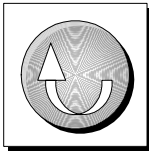
The Adaptive Server Anywhere sample database

Many of the examples throughout the documentation use the Adaptive Server Anywhere sample database.

The sample database is held in a file named *asademo.db*, and is located in your SQL Anywhere directory.

The sample database represents a small company. It contains internal information about the company (employees, departments, and finances) as well as product information and sales information (sales orders, customers, and contacts). All information in the database is fictional.

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

asademo.db

product

id	integer	<pk>
name	char(15)	
description	char(30)	
size	char(18)	
color	char(6)	
quantity	integer	
unit_price	numeric (15,2)	

sales_order_items

id	integer	<pk,fk>
line_id	smallint	<pk>
prod_id	integer	<fk>
quantity	integer	
ship_date	date	

id = prod_id

Employee

emp_id	integer	<pk>
manager_id	integer	
emp_fname	char(20)	
emp_lname	char(20)	
dept_id	integer	<fk>
street	char(40)	
city	char(20)	
state	char(4)	
zip_code	char(9)	
phone	char(10)	
status	char(1)	
ss_number	char(11)	
salary	numeric(20,3)	
start_date	date	
termination_date	date	
birth_date	date	
bene_health_ins	char(1)	
bene_life_ins	char(1)	
bene_day_care	char(1)	
sex	char(1)	

emp_id = sales_rep

id = id

sales_order

id	integer	<pk>
cust_id	integer	<fk>
order_date	date	
fin_code_id	char(2)	<fk>
region	char(7)	
sales_rep	integer	<fk>

customer

id	integer	<pk>
fname	char(15)	
lname	char(20)	
address	char(35)	
city	char(20)	
state	char(2)	
zip	char(10)	
phone	char(20)	
company_name	char(35)	

id = cust_id

code = fin_code_id

fin_code

code	char(2)	<pk>
type	char(10)	
description	char(50)	

dept_id = dept_id

emp_id = dept_head_id

contact

id	integer	<pk>
last_name	char(15)	
first_name	char(15)	
title	char(2)	
street	char(30)	
city	char(20)	
state	char(2)	
zip	char(5)	
phone	char(10)	
fax	char(10)	

code = code

fin_data

year	char(4)	<pk>
quarter	char(2)	<pk>
code	char(2)	<pk,fk>
amount	numeric(9)	

department

dept_id	integer	<pk>
dept_name	char(40)	
dept_head_id	integer	<fk>

Finding out more and providing feedback

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

You can provide feedback on this documentation and on the software through newsgroups set up to discuss SQL Anywhere technologies. These newsgroups can be found on the *forums.sybase.com* news server.

The newsgroups include the following:

♦ sybase.public.sqlanywhere.general

♦ sybase.public.sqlanywhere.linux

♦ sybase.public.sqlanywhere.mobilink

♦ sybase.public.sqlanywhere.product_futures_discussion

♦ sybase.public.sqlanywhere.replication

♦ sybase.public.sqlanywhere.ultralite

Newsgroup disclaimer

iAnywhere Solutions has no obligation to provide solutions, information or ideas on its newsgroups, nor is iAnywhere Solutions obliged to provide anything other than a systems operator to monitor the service and insure its operation and availability.

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

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

CHAPTER 1

Programming Interface Overview

About this chapter   This chapter introduces each of the programming interfaces for Adaptive
Server Anywhere. Any client application uses one of these interfaces to
communicate with the database.

Contents

The ODBC programming interface

ODBC (Open Database Connectivity) is a standard call level interface (CLI) developed by Microsoft. It is based on the SQL Access Group CLI specification. ODBC applications can run against any data source that provides an ODBC driver. ODBC is a good choice for a programming interface if you would like your application to be portable to other data sources that have ODBC drivers.

ODBC is a low-level interface. Almost all the Adaptive Server Anywhere functionality is available with this interface. ODBC is available as a DLL under Windows operating systems with the exception of Windows CE. It is provided as a library for UNIX.

The primary documentation for ODBC is the Microsoft ODBC Software Development Kit. The current book provides some additional notes specific to Adaptive Server Anywhere for ODBC developers.

☞ ODBC is described in .

The ADO.NET programming interface

ADO.NET is the latest data access API from Microsoft in the line of ODBC, OLE DB, and ADO. It is the preferred data access component for the Microsoft .NET Framework and allows you to access relational database systems.

The Adaptive Server Anywhere .NET data provider implements the iAnywhere.Data.AsaClient namespace and allows you to write programs in any of the .NET supported languages, such as C# and Visual Basic .NET, and access data from Adaptive Server Anywhere.

In addition to this book, you may wish to consult other materials on .NET data access to help in your development efforts. For example, *http://msdn.-microsoft.com/library/default.asp?url=/library/en-us/dnbda/html/daag.asp*.

☞ The ADO.NET programming interface is described in "Introduction to the Adaptive Server Anywhere .NET Data Provider" on page 343, "Using the Adaptive Server Anywhere .NET Data Provider Sample Applications" on page 347, "Developing Applications with the .NET Data Provider" on page 357, and "Adaptive Server Anywhere .NET Data Provider API Reference" on page 391.

The OLE DB and ADO programming interface

OLE DB is a set of Component Object Model (COM) interfaces developed by Microsoft, which provide applications with uniform access to data stored in diverse information sources and which also provide the ability to implement additional database services. These interfaces support the amount of DBMS functionality appropriate to the data store, enabling it to share its data.

ADO is an object model for programmatically accessing, editing, and updating a wide variety of data sources through OLE DB system interfaces. ADO is also developed by Microsoft. Most developers using the OLE DB programming interface do so by writing to the ADO API rather than directly to the OLE DB API.

Adaptive Server Anywhere includes an OLE DB provider for OLE DB and ADO programmers.

The primary documentation for OLE DB and ADO programming is the Microsoft Developer Network. The current book provides some additional notes specific to Adaptive Server Anywhere for OLE DB and ADO developers.

☞ The OLE DB provider is described in "The OLE DB and ADO Programming Interfaces" on page 327.

Do not confuse the ADO interface with ADO.NET. ADO.NET is a separate interface. For more information, see "The ADO.NET programming interface" on page 3.

The Embedded SQL programming interface

Embedded SQL is a system in which SQL commands are embedded right in a C or C++ source file. A preprocessor translates these statements into calls to a runtime library. Embedded SQL is an ISO/ANSI and IBM standard.

Embedded SQL is portable to other databases and other environments, and is functionally equivalent in all operating environments. It is a comprehensive, low-level interface that provides all of the functionality available in the product. Embedded SQL requires knowledge of C or C++ programming languages.

☞ Embedded SQL is described in "Embedded SQL Programming" on page 135.

The JDBC programming interface

JDBC is a call-level interface for Java applications. Developed by Sun Microsystems, JDBC provides Java programmers with a uniform interface to a wide range of relational databases, and provides a common base on which higher level tools and interfaces can be built. JDBC is now a standard part of Java and is included in the JDK.

SQL Anywhere Studio includes a pure Java JDBC driver, named Sybase jConnect. It also includes the iAnywhere JDBC driver, which is a type 2 driver. Both are described in "JDBC Programming" on page 103. For information on choosing a driver, see "Choosing a JDBC driver" on page 104.

In addition to using JDBC as a client side application programming interface, you can also use JDBC inside the database server to access data from Java in the database.

☞ The JDBC interface is described in "JDBC Programming" on page 103.

The Open Client programming interface

Sybase Open Client provides customer applications, third-party products, and other Sybase products with the interfaces needed to communicate with Adaptive Server Anywhere and other Open Servers.

When to use Open Client   You should consider using the Open Client interface if you are concerned with Adaptive Server Enterprise compatibility or if you are using other Sybase products that support the Open Client interface, such as Replication Server.

☞ For more information about the Open Client interface, see "Adaptive Server Anywhere as an Open Server" [*ASA Database Administration Guide, page 101*].

Open Client architecture

Open Client can be thought of as comprising two components: programming interfaces and network services.

Client Library and DB-Library   Open Client provides two core programming interfaces for writing client applications: DB-Library and Client-Library.

Open Client DB-Library provides support for older Open Client applications, and is a completely separate programming interface from Client-Library. DB-Library is documented in the *Open Client DB-Library/C Reference Manual*, provided with the Sybase Open Client product.

Client-Library programs also depend on CS-Library, which provides routines that are used in both Client-Library and Server-Library applications. Client-Library applications can also use routines from Bulk-Library to facilitate high-speed data transfer.

Both CS-Library and Bulk-Library are included in the Sybase Open Client, available separately.

Network services   Open Client network services include Sybase Net-Library, which provides support for specific network protocols such as TCP/IP and DECnet. The Net-Library interface is invisible to application programmers. However, on some platforms, an application may need a different Net-Library driver for different system network configurations. Depending on your host platform, the Net-Library driver is specified either by the system's Sybase configuration or when you compile and link your programs.

☞ Instructions for driver configuration can be found in the *Open Client/Server Configuration Guide.*

☞ Instructions for building Client-Library programs can be found in the

Open Client/Server Programmer's Supplement.

The Perl DBD::ASAny driver for the DBI module

The DBD::ASAny interface provides access to Adaptive Server Anywhere databases from scripts written in Perl. ASAny is a driver for the Database Independent Interface for Perl (DBI) module written by Tim Bunce. Once you have installed the DBI module and DBD::ASAny, your can access and change the information in Adaptive Server Anywhere databases from Perl.

☞ For instructions on installing these components and writing Perl scripts that use them, see "The DBD::ASAny Perl Interface" on page 469.

Code samples and other programming interfaces

Unsupported code that provides other interfaces to Adaptive Server Anywhere is available for download.

♦ **PHP module**   The Adaptive Server Anywhere PHP module can be used to retrieve data from Adaptive Server Anywhere databases. To make PHP connect to Adaptive Server Anywhere using the PHP module, you must add the Adaptive Server Anywhere module's files to PHP's source tree, then re-compile PHP.

The PHP module is available as a separate download. For more information, see *http://www.ianywhere.com/developer/code_-samples/sqlany_php_module.html*.

♦ **Perl DBI driver**   DBD::ASAny is the Adaptive Server Anywhere database driver for DBI, which is a database access Application Programming Interface (API) for the Perl Language. The DBI API Specification defines a set of functions, variables and conventions that provide a consistent database interface independent of the actual database being used. Using DBI and DBD::ASAny, your perl scripts will have direct access to Sybase Adaptive Server Anywhere database servers.

For more information, see *http://www.ianywhere.com/developer/code_samples/dbd_asa_perl.html*.

Code samples   Application code samples are one of the most useful tools for application developers. Code samples, utilities, and solution samples are available on the iAnywhere website at *http://www.ianywhere.com/downloads*.

CHAPTER 2

Using SQL in Applications

About this chapter   Many aspects of database application development depend on your
application development tool, database interface, and programming
language, but there are some common problems and principles that affect
multiple aspects of database application development.

This chapter describes some principles and techniques common to most or
all interfaces and provides pointers for more information. It does not provide
a detailed guide for programming using any one interface.

Contents

Executing SQL statements in applications

The way you include SQL statements in your application depends on the application development tool and programming interface you use.

♦ **ODBC**   If you are writing directly to the ODBC programming interface, your SQL statements appear in function calls. For example, the following C function call executes a DELETE statement:

```
SQLExecDirect( stmt,
    "DELETE FROM employee
     WHERE emp_id = 105",
    SQL_NTS );
```

♦ **ADO.NET**   You can execute SQL statements using a variety of ADO.NET objects. The AsaCommand object is one example:

```
AsaCommand cmd = new AsaCommand(
    "select emp_lname from employee", conn );
AsaDataReader reader = cmd.ExecuteReader();
```

♦ **JDBC**   If you are using the JDBC programming interface, you can execute SQL statements by invoking methods of the **statement** object. For example,

```
stmt.executeUpdate(
    "DELETE FROM employee
     WHERE emp_id = 105" );
```

♦ **Embedded SQL**   If you are using embedded SQL, you prefix your C language SQL statements with the keyword EXEC SQL. The code is then run through a preprocessor before compiling. For example,

```
EXEC SQL EXECUTE IMMEDIATE
 'DELETE FROM employee
  WHERE emp_id = 105';
```

♦ **Sybase Open Client**   If you use the Sybase Open Client interface, your SQL statements appear in function calls. For example, the following pair of calls executes a DELETE statement:

```
ret = ct_command( cmd, CS_LANG_CMD,
                  "DELETE FROM employee
                   WHERE emp_id=105"
                 CS_NULLTERM,
                 CS_UNUSED);
ret = ct_send(cmd);
```

♦ **Application Development Tools**   Application development tools such as the members of the Sybase Enterprise Application Studio family

provide their own SQL objects, which use either ODBC (PowerBuilder) or JDBC (Power J) under the covers.

For more detailed information on how to include SQL in your application, see your development tool documentation. If you are using ODBC or JDBC, consult the software development kit for those interfaces.

For a detailed description of embedded SQL programming, see "Embedded SQL Programming" on page 135.

Applications inside the server

In many ways, stored procedures and triggers act as applications or parts of applications running inside the server. You can use many of the techniques here in stored procedures also. Stored procedures use statements very similar to embedded SQL statements.

☞ For more information about stored procedures and triggers, see "Using Procedures, Triggers, and Batches" [*ASA SQL User's Guide,* page 655].

☞ Java classes in the database can use the JDBC interface in the same way as Java applications outside the server. This chapter discusses some aspects of JDBC. For other information on using JDBC, see "JDBC Programming" on page 103.

Preparing statements

Each time a statement is sent to a database, the server must first **prepare** the statement. Preparing the statement can include:

♦ Parsing the statement and transforming it into an internal form.

♦ Verifying the correctness of all references to database objects by checking, for example, that columns named in a query actually exist.

♦ Causing the query optimizer to generate an access plan if the statement involves joins or subqueries.

♦ Executing the statement after all these steps have been carried out.

Reusing prepared statements can improve performance

If you find yourself using the same statement repeatedly, for example, inserting many rows into a table, repeatedly preparing the statement causes a significant and unnecessary overhead. To remove this overhead, some database programming interfaces provide ways of using prepared statements. A **prepared statement** is a statement containing a series of placeholders. When you want to execute the statement, all you have to do is assign values to the placeholders, rather than prepare the entire statement over again.

Using prepared statements is particularly useful when carrying out many similar actions, such as inserting many rows.

Generally, using prepared statements requires the following steps:

1. **Prepare the statement** In this step you generally provide the statement with some placeholder character instead of the values.

2. **Repeatedly execute the prepared statement** In this step you supply values to be used each time the statement is executed. The statement does not have to be prepared each time.

3. **Drop the statement** In this step you free the resources associated with the prepared statement. Some programming interfaces handle this step automatically.

Do not prepare statements that are used only once

In general, you should not prepare statements if you'll only execute them once. There is a slight performance penalty for separate preparation and execution, and it introduces unnecessary complexity into your application.

In some interfaces, however, you do need to prepare a statement to associate it with a cursor.

☞ For information about cursors, see "Introduction to cursors" on page 17.

The calls for preparing and executing statements are not a part of SQL, and they differ from interface to interface. Each of the Adaptive Server

Anywhere programming interfaces provides a method for using prepared statements.

How to use prepared statements

This section provides a brief overview of how to use prepared statements. The general procedure is the same, but the details vary from interface to interface. Comparing how to use prepared statements in different interfaces illustrates this point.

❖ **To use a prepared statement (generic)**

1. Prepare the statement.

2. Set up **bound parameters**, which will hold values in the statement.

3. Assign values to the bound parameters in the statement.

4. Execute the statement.

5. Repeat steps 3 and 4 as needed.

6. Drop the statement when finished. This step is not required in JDBC, as Java's garbage collection mechanisms handle this for you.

❖ **To use a prepared statement (embedded SQL)**

1. Prepare the statement using the EXEC SQL PREPARE command.

2. Assign values to the parameters in the statement.

3. Execute the statement using the EXE SQL EXECUTE command.

4. Free the resources associated with the statement using the EXEC SQL DROP command.

❖ **To use a prepared statement (ODBC)**

1. Prepare the statement using **SQLPrepare**.

2. Bind the statement parameters using **SQLBindParameter**.

3. Execute the statement using **SQLExecute**.

4. Drop the statement using **SQLFreeStmt**.

   ☞ For more information, see "Executing prepared statements" on page 247 and the ODBC SDK documentation.

15

❖ **To use a prepared statement (ADO.NET)**

1. Create an AsaCommand object holding the statement.

   ```
   AsaCommand cmd = new AsaCommand(
        "select emp_lname from employee", conn );
   ```

2. Declare data types for the parameters in the statement.

   Use the AsaCommand.CreateParameter method.

3. Prepare the statement using the Prepare method.

   ```
   cmd.Prepare();
   ```

4. Execute the statement.

   ```
   AsaDataReader reader = cmd.ExecuteReader();
   ```

For more information, see

❖ **To use a prepared statement (JDBC)**

1. Prepare the statement using the **prepareStatement** method of the connection object. This returns a prepared statement object.

2. Set the statement parameters using the appropriate **set***Type* methods of the prepared statement object. Here, *Type* is the data type assigned.

3. Execute the statement using the appropriate method of the prepared statement object. For inserts, updates, and deletes this is the **executeUpdate** method.

   ☞ For more information on using prepared statements in JDBC, see "Using prepared statements for more efficient access" on page 129.

❖ **To use a prepared statement (Open Client)**

1. Prepare the statement using the **ct_dynamic** function, with a CS_PREPARE type parameter.

2. Set statement parameters using **ct_param**.

3. Execute the statement using **ct_dynamic** with a CS_EXECUTE type parameter.

4. Free the resources associated with the statement using **ct_dynamic** with a CS_DEALLOC type parameter.

   ☞ For more information on using prepared statements in Open Client, see "Using SQL in Open Client applications" on page 465.

Introduction to cursors

When you execute a query in an application, the result set consists of a number of rows. In general, you do not know how many rows the application is going to receive before you execute the query. Cursors provide a way of handling query result sets in applications.

The way you use cursors, and the kinds of cursors available to you, depend on the programming interface you use. For a list of cursor types available from each interface, see "Availability of cursors" on page 26.

With cursors, you can carry out the following tasks within any programming interface:

♦ Loop over the results of a query.

♦ Carry out inserts, updates, and deletes on the underlying data at any point within a result set.

In addition, some programming interfaces allow you to use special features to tune the way result sets return to your application, providing substantial performance benefits for your application.

☞ For more information on the kinds of cursors available through different programming interfaces, see "Availability of cursors" on page 26.

What are cursors?

A **cursor** is a name associated with a result set. The result set is obtained from a SELECT statement or stored procedure call.

A cursor is a handle on the result set. At any time, the cursor has a well-defined position within the result set. With a cursor you can examine and possibly manipulate the data one row at a time. Adaptive Server Anywhere cursors support forward and backward movement through the query results.

Cursor positions — Cursors can be positioned in the following places:

♦ Before the first row of the result set.

♦ On a row in the result set.

♦ After the last row of the result set.

Absolute row
from start

Absolute row
from end

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

Cursor position and result set are maintained in the database server. Rows are **fetched** by the client for display and processing either one at a time or a few at a time. The entire result set does not need to be delivered to the client.

Benefits of using cursors

You do not need to use cursors in database applications, but they do provide a number of benefits. These benefits follow from the fact that if you do not use a cursor, the entire result set must be transferred to the client for processing and display:

♦ **Client-side memory**   For large results, holding the entire result set on the client can lead to demanding memory requirements.

♦ **Response time**   Cursors can provide the first few rows before the whole result set is assembled. If you do not use cursors, the entire result set must be delivered before any rows are displayed by your application.

♦ **Concurrency control**   If you make updates to your data and do not use cursors in your application, you must send separate SQL statements to

the database server to apply the changes. This raises the possibility of concurrency problems if the result set has changed since it was queried by the client. In turn, this raises the possibility of lost updates.

Cursors act as pointers to the underlying data, and so impose proper concurrency constraints on any changes you make.

Steps in using cursors

Using a cursor in embedded SQL is different than using a cursor in other interfaces.

❖ **To use a cursor (embedded SQL)**

1. Prepare a statement.

   Cursors generally use a statement handle rather than a string. You need to prepare a statement to have a handle available.

   ☞ For information on preparing a statement, see "Preparing statements" on page 14.

2. Declare the cursor.

   Each cursor refers to a single SELECT or CALL statement. When you declare a cursor, you state the name of the cursor and the statement it refers to.

   ☞ For more information, see "DECLARE CURSOR statement [ESQL] [SP]" [*ASA SQL Reference,* page 436].

3. Open the cursor.

   ☞ For more information, see "OPEN statement [ESQL] [SP]" [*ASA SQL Reference,* page 553].

   In the case of a CALL statement, opening the cursor executes the query up to the point where the first row is about to be obtained.

4. Fetch results.

   Although simple fetch operations move the cursor to the next row in the result set, Adaptive Server Anywhere permits more complicated movement around the result set. How you declare the cursor determines which fetch operations are available to you.

   ☞ For more information, see "FETCH statement [ESQL] [SP]" [*ASA SQL Reference,* page 482], and "Fetching data" on page 166.

5. Close the cursor.

   When you have finished with the cursor, close it. This frees any locks held on the underlying data.

☞ For more information, see "CLOSE statement [ESQL] [SP]" [*ASA SQL Reference,* page 325].

6. Drop the statement.

   To free the memory associated with the cursor and its associated statement, you must free the statement.

   ☞ For more information, see "DROP STATEMENT statement [ESQL]" [*ASA SQL Reference,* page 463].

❖ **To use a cursor (ODBC, ADO.NET, JDBC, Open Client)**

1. Prepare and execute a statement.

   Execute a statement using the usual method for the interface. You can prepare and then execute the statement, or you can execute the statement directly.

   With ADO.NET, only the AsaCommand.ExecuteReader command returns a cursor. It provides a read-only, forward-only cursor.

2. Test to see if the statement returns a result set.

   A cursor is implicitly opened when a statement that creates a result set is executed. When the cursor is opened, it is positioned before the first row of the result set.

3. Fetch results.

   Although simple fetch operations move the cursor to the next row in the result set, Adaptive Server Anywhere permits more complicated movement around the result set.

4. Close the cursor.

   When you have finished with the cursor, close it to free associated resources.

5. Free the statement.

   If you used a prepared statement, free it to reclaim memory.

Prefetching rows   In some cases the interface library may carry out performance optimizations under the covers (such as prefetching results), so these steps in the client application may not correspond exactly to software operations.

Working with cursors

This section describes how to carry out different kinds of operations using cursors.

Cursor positioning

When a cursor is opened, it is positioned before the first row. You can move the cursor position to an absolute position from the start or the end of the query results, or to a position relative to the current cursor position. The specifics of how you change cursor position, and what operations are possible, is governed by the programming interface.

The number of row positions you can fetch in a cursor is governed by the size of an integer. You can fetch rows numbered up to number 2147483646, which is one less than the value that can be held in an integer. When using negative numbers (rows from the end) you can fetch down to one more than the largest negative value that can be held in an integer.

You can use special positioned update and delete operations to update or delete the row at the current position of the cursor. If the cursor is positioned before the first row or after the last row, a `No current row of cursor` error is returned.

> **Cursor positioning problems**
> Inserts and some updates to asensitive cursors can cause problems with cursor positioning. Adaptive Server Anywhere does not put inserted rows at a predictable position within a cursor unless there is an ORDER BY clause on the SELECT statement. In some cases, the inserted row does not appear at all until the cursor is closed and opened again.
>
> With Adaptive Server Anywhere, this occurs if a work table had to be created to open the cursor (see "Use of work tables in query processing" [*ASA SQL User's Guide,* page 190] for a description).
>
> The UPDATE statement may cause a row to move in the cursor. This happens if the cursor has an ORDER BY clause that uses an existing index (a work table is not created). Using STATIC SCROLL cursors alleviates these problems but requires more memory and processing.

Configuring cursors on opening

You can configure the following aspects of cursor behavior when you open the cursor:

♦ **Isolation level**   You can explicitly set the isolation level of operations on

a cursor to be different from the current isolation level of the transaction. To do this, set the ISOLATION_LEVEL option.

☞ For more information, see "ISOLATION_LEVEL option [compatibility]" [*ASA Database Administration Guide,* page 659].

♦ **Holding** By default, cursors in embedded SQL close at the end of a transaction. Opening a cursor **WITH HOLD** allows you to keep it open until the end of a connection, or until you explicitly close it. ODBC, JDBC and Open Client leave cursors open at the end of transactions by default.

Fetching rows through a cursor

The simplest way of processing the result set of a query using a cursor is to loop through all the rows of the result set until there are no more rows.

❖ **To loop through the rows of a result set**

1. Declare and open the cursor (embedded SQL), or execute a statement that returns a result set (ODBC, JDBC, Open Client) or AsaDataReader object (ADO.NET).

2. Continue to fetch the next row until you get a Row Not Found error.

3. Close the cursor.

How step 2 of this operation is carried out depends on the interface you use. For example,

♦ **ODBC** **SQLFetch, SQLExtendedFetch,** or **SQLFetchScroll** advances the cursor to the next row and returns the data.

☞ For more information on using cursors in ODBC, see "Working with result sets" on page 249.

♦ **ADO.NET** Use the AsaDataReader.NextResult method. See "NextResult method" on page 430.

♦ **Embedded SQL** The FETCH statement carries out the same operation.

☞ For more information on using cursors in embedded SQL, see "Using cursors in embedded SQL" on page 167.

♦ **JDBC** The **next** method of the **ResultSet** object advances the cursor and returns the data.

☞ For more information on using the **ResultSet** object in JDBC, see "Queries using JDBC" on page 128.

♦ **Open Client** The **ct_fetch** function advances the cursor to the next row and returns the data.

☞ For more information on using cursors in Open Client applications, see "Using cursors" on page 465.

Fetching multiple rows

This section discusses how fetching multiple rows at a time can improve performance.

Multiple-row fetching should not be confused with prefetching rows, which is described in the next section. Multiple row fetching is performed by the application, while prefetching is transparent to the application, and provides a similar performance gain.

Multiple-row fetches   Some interfaces provide methods for fetching more than one row at a time into the next several fields in an array. Generally, the fewer separate fetch operations you execute, the fewer individual requests the server must respond to, and the better the performance. A modified FETCH statement that retrieves multiple-rows is also sometimes called a **wide fetch**. Cursors that use multiple-row fetches are sometimes called **block cursors** or **fat cursors**.

Using multiple-row fetching
♦ In ODBC, you can set the number of rows that will be returned on each call to **SQLFetchScroll** or **SQLExtendedFetch** by setting the SQL_ROWSET_SIZE attribute.

♦ In embedded SQL, the FETCH statement uses an ARRAY clause to control the number of rows fetched at a time.

♦ Open Client and JDBC do not support multi-row fetches. They do use prefetching.

Fetching with scrollable cursors

ODBC and embedded SQL provide methods for using scrollable cursors and dynamic scrollable cursors. These methods allow you to move several rows forward at a time, or to move backwards through the result set.

The JDBC and Open Client interfaces do not support scrollable cursors.

Prefetching does not apply to scrollable operations. For example, fetching a row in the reverse direction does not prefetch several previous rows.

Modifying rows through a cursor

Cursors can do more than just read result sets from a query. You can also

modify data in the database while processing a cursor. These operations are commonly called **positioned** insert, update, and delete operations, or **PUT** operations if the action is an insert.

Not all query result sets allow positioned updates and deletes. If you carry out a query on a non-updatable view, then no changes occur to the underlying tables. Also, if the query involves a join, then you must specify which table you wish to delete from, or which columns you wish to update, when you carry out the operations.

Inserts through a cursor can only be executed if any non-inserted columns in the table allow NULL or have defaults.

If multiple rows are inserted into a value-sensitive (keyset driven) cursor, they appear at the end of the cursor result set. The rows appear at the end even if they do not match the WHERE clause of the query or if an ORDER BY clause would normally have placed them at another location in the result set. This behavior is independent of programming interface. For example, it applies when using the embedded SQL PUT statement or the ODBC SQLBulkOperations function. The value of an autoincrement column for the most recent row inserted can be found by selecting the last row in the cursor. For example, in embedded SQL the value could be obtained using FETCH ABSOLUTE -1 *cursor-name*. As a result of this behavior, the first multiple-row insert for a value-sensitive cursor may be expensive.

ODBC, embedded SQL, and Open Client permit data modification using cursors, but JDBC 1.1 does not. With Open Client, you can delete and update rows, but you can only insert rows on a single-table query.

Which table are rows deleted from?

If you attempt a positioned delete through a cursor, the table from which rows are deleted is determined as follows:

1. If no FROM clause is included in the DELETE statement, the cursor must be on a single table only.

2. If the cursor is for a joined query (including using a view containing a join), then the FROM clause must be used. Only the current row of the specified table is deleted. The other tables involved in the join are not affected.

3. If a FROM clause is included, and no table owner is specified, the table-spec value is first matched against any correlation names.

   ☞ For more information, see the "FROM clause" [*ASA SQL Reference*, page 491].

4. If a correlation name exists, the table-spec value is identified with the correlation name.

5. If a correlation name does not exist, the table-spec value must be unambiguously identifiable as a table name in the cursor.

6. If a FROM clause is included, and a table owner is specified, the table-spec value must be unambiguously identifiable as a table name in the cursor.

7. The positioned DELETE statement can be used on a cursor open on a view as long as the view is updatable.

Canceling cursor operations

You can cancel a request through an interface function. From Interactive SQL, you can cancel a request by pressing the Interrupt SQL Statement button on the toolbar (or by choosing Stop from the SQL menu).

If you cancel a request that is carrying out a cursor operation, the position of the cursor is indeterminate. After canceling the request, you must locate the cursor by its absolute position, or close it.

Choosing cursor types

This section describes mappings between Adaptive Server Anywhere cursors and the options available to you from the programming interfaces supported by Adaptive Server Anywhere.

☞ For information on Adaptive Server Anywhere cursors, see "Adaptive Server Anywhere cursors" on page 30.

Availability of cursors

Not all interfaces provide support for all types of cursors.

♦ ODBC and OLE DB (ADO) support all types of cursors.

☞ For more information, see "Working with result sets" on page 249.

♦ Embedded SQL supports all the types of cursors.

♦ ADO.NET provides only forward-only, read-only cursors.

♦ For JDBC:
  • jConnect 4.x provides only asensitive cursors.
  • jConnect 5.x supports all types of cursors, but there is a severe performance penalty for scrollable cursors.
  • The iAnywhere JDBC driver supports all types of cursors.

♦ Sybase Open Client supports only asensitive cursors. Also, a severe performance penalty results when using updatable, non-unique cursors.

Cursor properties

You request a cursor type, either explicitly or implicitly, from the programming interface. Different interface libraries offer different choices of cursor types. For example, JDBC and ODBC specify different cursor types.

Each cursor type is defined by a number of characteristics:

♦ **Uniqueness**   Declaring a cursor to be unique forces the query to return all the columns required to uniquely identify each row. Often this means returning all the columns in the primary key. Any columns required but not specified are added to the result set. The default cursor type is non-unique.

♦ **Updatability**   A cursor declared as read only may not be used in a positioned update or delete operation. The default cursor type us updatable.

♦ **Scrollability**   You can declare cursors to behave different ways as you move through the result set. Some cursors can fetch only the current row or the following row. Others can move backwards and forwards through the result set.

♦ **Sensitivity**   Changes to the database may or may not be visible through a cursor.

These characteristics may have significant side effects on performance and on database server memory usage.

Adaptive Server Anywhere makes available cursors with a variety of mixes of these characteristics. When you request a cursor of a given type, Adaptive Server Anywhere matches those characteristics as well as it can. The details of how Adaptive Server Anywhere cursors match the cursor types specified in the programming interfaces are the subject of the following sections.

There are some occasions when not all characteristics can be supplied. For example, insensitive cursors in Adaptive Server Anywhere must be read-only, for reasons described below. If your application requests an updatable insensitive cursor, a different cursor type (value-sensitive) is supplied instead.

Requesting Adaptive Server Anywhere cursors

When you request a cursor type from your client application, Adaptive Server Anywhere provides a cursor. Adaptive Server Anywhere cursors are defined, not by the type as specified in the programming interface, but by the sensitivity of the result set to changes in the underlying data. Depending on the cursor type you ask for, Adaptive Server Anywhere provides a cursor with behavior to match the type.

Adaptive Server Anywhere cursor sensitivity is set in response to the client cursor type request.

ODBC and OLE DB

The following table illustrates the cursor sensitivity that is set in response to different ODBC scrollable cursor types.

ODBC scrollable cursor type	Adaptive Server Anywhere cursor
STATIC	Insensitive
KEYSET	Value-sensitive
DYNAMIC	Sensitive
MIXED	Value-sensitive

☞ For information on Adaptive Server Anywhere cursors and their behavior, see "Adaptive Server Anywhere cursors" on page 30. For information on how to request a cursor type in ODBC, see "Choosing a cursor characteristics" on page 249.

Exceptions   If a STATIC cursor is requested as updatable, a value-sensitive cursor is supplied instead and a warning is issued.

If a DYNAMIC or MIXED cursor is requested and the query cannot be executed without using work tables, a warning is issued and an asensitive cursor is supplied instead.

ADO.NET

Forward-only, read-only cursors are available by using AsaCommand.ExecuteReader. The AsaDataAdapter object uses a client-side result set instead of cursors.

Embedded SQL

To request a cursor from an embedded SQL application, you specify the cursor type on the DECLARE statement. The following table illustrates the cursor sensitivity that is set in response to different requests:

Cursor type	Adaptive Server Anywhere cursor
NO SCROLL	Asensitive
DYNAMIC SCROLL	Asensitive
SCROLL	Value-sensitive
INSENSITIVE	Insensitive
SENSITIVE	Sensitive

Exceptions   If an DYNAMIC SCROLL or NO SCROLL cursor is requested as UPDATABLE, then a sensitive or value-sensitive cursor is supplied. It is not guaranteed which of the two is supplied. This uncertainty fits the definition of asensitive behavior.

If an INSENSITIVE cursor is requested as UPDATABLE, then a value-sensitive cursor is supplied.

If a DYNAMIC SCROLL cursor is requested, if the PREFETCH database option is set to OFF, and if the query execution plan involves no work tables, then a sensitive cursor may be supplied. Again, this uncertainty fits the definition of asensitive behavior.

JDBC

Only one kind of cursor is available to JDBC applications. This is an asensitive cursor. In JDBC you execute an **ExecuteQuery** statement to open a cursor.

Open Client

Only one kind of cursor is available to Open Client applications. This is an asensitive cursor.

Bookmarks and cursors

ODBC provides **bookmarks**, or values, used to identify rows in a cursor. Adaptive Server Anywhere supports bookmarks for all kinds of cursors except DYNAMIC cursors.

Block cursors

ODBC provides a cursor type called a block cursor. When you use a BLOCK cursor, you can use **SQLFetchScroll** or **SQLExtendedFetch** to fetch a block of rows, rather than a single row. Block cursors behave identically to embedded SQL ARRAY fetches.

Adaptive Server Anywhere cursors

Any cursor, once opened, has an associated result set. The cursor is kept open for a length of time. During that time, the result set associated with the cursor may be changed, either through the cursor itself or, subject to isolation level requirements, by other transactions. Some cursors permit changes to the underlying data to be visible, while others do not reflect these changes. The different behavior of cursors with respect to changes to the underlying data is the **sensitivity** of the cursor.

Adaptive Server Anywhere provides cursors with a variety of sensitivity characteristics. This section describes what sensitivity is, and describes the sensitivity characteristics of cursors.

This section assumes that you have read

Membership, order, and value changes

Changes to the underlying data can affect the result set of a cursor in the following ways:

♦ **Membership**   The set of rows in the result set, as identified by their primary key values.

♦ **Order**   The order of the rows in the result set.

♦ **Value**   The values of the rows in the result set.

For example, consider the following simple table with employee information (emp_id is the primary key column):

emp_id	emp_lname
1	Whitney
2	Cobb
3	Chin

A cursor on the following query returns all results from the table in primary key order:

```
SELECT emp_id, emp_lname
FROM employee
ORDER BY emp_id
```

The membership of the result set could be changed by adding a new row or deleting a row. The values could be changed by changing one of the names in the table. The order could be changed by changing the primary key value of one of the employees.

Visible and invisible changes

Subject to isolation level requirements, the membership, order, and values of the result set of a cursor can be changed after the cursor is opened.

Depending on the type of cursor in use, the result set as seen by the application may change to reflect these changes or may not.

Changes to the underlying data may be **visible** or **invisible** through the cursor. A visible change is a change that is reflected in the result set of the cursor. Changes to the underlying data that are not reflected in the result set seen by the cursor are invisible.

Cursor sensitivity overview

Adaptive Server Anywhere cursors are classified by their sensitivity with respect to changes of the underlying data. In particular, cursor sensitivity is defined in terms of which changes are visible.

♦ **Insensitive cursors**   The result set is fixed when the cursor is opened. No changes to the underlying data are visible.

☞ For more information, see "Insensitive cursors" on page 35.

♦ **Sensitive cursors**   The result set can change after the cursor is opened. All changes to the underlying data are visible.

☞ For more information, see "Sensitive cursors" on page 36.

♦ **Asensitive cursors**   Changes may be reflected in the membership, order, or values of the result set seen through the cursor, or may not be reflected at all.

☞ For more information, see "Asensitive cursors" on page 38.

♦ **Value-sensitive cursors**   Changes to the order or values of the underlying data. The membership of the result set is fixed when the cursor is opened.

☞ For more information, see "Value-sensitive cursors" on page 39.

The differing requirements on cursors place different constraints on execution, and so performance. For more information, see "Cursor sensitivity and performance" on page 41.

Cursor sensitivity example: a deleted row

This example uses a simple query to illustrate how different cursors respond to a row in the result set being deleted.

Consider the following sequence of events:

1. An application opens a cursor on the following query against the sample database.

```
SELECT emp_id, emp_lname
FROM employee
ORDER BY emp_id
```

emp_id	emp_lname
102	Whitney
105	Cobb
160	Breault
…	…

2. The application fetches the first row through the cursor (102).

3. The application fetches the next row through the cursor (105).

4. A separate transaction deletes employee 102 (Whitney) and commits the change.

The results of cursor actions in this situation depend on the cursor sensitivity:

♦ **Insensitive cursors**   The DELETE is not reflected in either the membership or values of the results as seen through the cursor:

Action	Result
Fetch previous row	Returns the original copy of the row (102).
Fetch the first row (absolute fetch)	Returns the original copy of the row (102).
Fetch the second row (absolute fetch)	Returns the unchanged row (105).

♦ **Sensitive cursors**   The membership of the result set has changed so that row 105 is now the first row in the result set:

Action	Result
Fetch previous row	Returns `Row Not Found` error. There is no previous row.
Fetch the first row (absolute fetch)	Returns row 105.
Fetch the second row (absolute fetch)	Returns row 160.

♦ **Value-sensitive cursors**   The membership of the result set is fixed, and so row 105 is still the second row of the result set. The DELETE is reflected in the values of the cursor, and creates an effective "hole" in the result set.

Action	Result
Fetch previous row	Returns No current row of cursor. There is a hole in the cursor where the first row used to be.
Fetch the first row (absolute fetch)	Returns No current row of cursor. There is a hole in the cursor where the first row used to be.
Fetch the second row (absolute fetch)	Returns row 105.

♦ **Asensitive cursors**   The membership and values of the result set are indeterminate with respect to the changes. The response to a fetch of the previous row, the first row, or the second row depends on the particular optimization method for the query, whether that method involved the formation of a work table, and whether the row being fetched was prefetched from the client.

The benefit of asensitive cursors is that for many applications, sensitivity is unimportant. In particular, if you are using a forward-only, read-only cursor, no underlying changes are seen. Also, if you are running at a high isolation level, underlying changes are disallowed.

Cursor sensitivity example: an updated row

This example uses a simple query to illustrate how different cursor types respond to a row in the result set being updated in such a way as to change the order of the result set.

Consider the following sequence of events:

1. An application opens a cursor on the following query against the sample database.

```
SELECT emp_id, emp_lname
FROM employee
```

emp_id	emp_lname
102	Whitney
105	Cobb
160	Breault
…	…

2. The application fetches the first row through the cursor (102).

3. The application fetches the next row through the cursor (105).

4. A separate transaction updates the employee ID of employee 102 (Whitney) to 165 and commits the change.

The results of the cursor actions in this situation depend on the cursor sensitivity:

♦ **Insensitive cursors**   The UPDATE is not reflected in either the membership or values of the results as seen through the cursor:

Action	Result
Fetch previous row	Returns the original copy of the row (102).
Fetch the first row (absolute fetch)	Returns the original copy of the row (102).
Fetch the second row (absolute fetch)	Returns the unchanged row (105).

♦ **Sensitive cursors**   The membership of the result set has changed so that row 105 is now the first row in the result set:

Action	Result
Fetch previous row	Returns Row Not Found. The membership of the result set has changed so that 105 is now the first row. The cursor is moved to the position before the first row.
Fetch the first row (absolute fetch)	Returns row 105.
Fetch the second row (absolute fetch)	Returns row 160.

In addition, a fetch on a sensitive cursor returns the warning SQLE_ROW_UPDATED_WARNING if the row has changed since the last

reading. The warning is given only once. Subsequent fetches of the same row do not produce the warning.

Similarly, a positioned update or delete through the cursor on a row since it was last fetched returns the SQLE_ROW_UPDATED_SINCE_READ error. An application must fetch the row again for an update or delete on a sensitive cursor to work.

An update to any column causes the warning/error, even if the column is not referenced by the cursor. For example, a cursor on a query returning emp_lname would report the update even if only the salary column was modified.

♦ **Value-sensitive cursors**   The membership of the result set is fixed, and so row 105 is still the second row of the result set. The UPDATE is reflected in the values of the cursor, and creates an effective "hole" in the result set.

Action	Result
Fetch previous row	Returns `Row Not Found`. The membership of the result set has changed so that 105 is now the first row: The cursor is positioned on the hole: it is before row 105.
Fetch the first row (absolute fetch)	Returns `Row Not Found`. The membership of the result set has changed so that 105 is now the first row: The cursor is positioned on the hole: it is before row 105.
Fetch the second row (absolute fetch)	Returns row 105.

♦ **Asensitive cursors**   The membership and values of the result set are indeterminate with respect to the changes. The response to a fetch of the previous row, the first row, or the second row depends on the particular optimization method for the query, whether that method involved the formation of a work table, and whether the row being fetched was prefetched from the client.

> **No warnings or errors in bulk operations mode**
> Update warning and error conditions do not occur in bulk operations mode (-b database server option).

Insensitive cursors

These cursors have insensitive membership, order, and values. No changes made after cursor open time are visible.

Insensitive cursors are used only for read-only cursor types.

Standards | Insensitive cursors correspond to the ISO/ANSI standard definition of insensitive cursors, and to ODBC static cursors.

Programming interfaces

Interface	Cursor type	Comment
ODBC, OLE DB, and ADO	Static	If an updatable static cursor is requested, a value-sensitive cursor is used instead.
Embedded SQL	INSENSITIVE or NO SCROLL	
JDBC	Unsupported	
Open Client	Unsupported	

Description | Insensitive cursors always return rows that match the query's selection criteria, in the order specified by any ORDER BY clause.

The result set of an insensitive cursor is fully materialized as a work table when the cursor is opened. This has the following consequences:

♦ If the result set is very large, the disk space and memory requirements for managing the result set may be significant.

♦ No row is returned to the application before the entire result set is assembled as a work table. For complex queries, this may lead to a delay before the first row is returned to the application.

♦ Subsequent rows can be fetched directly from the work table, and so are returned quickly. The client library may prefetch several rows at a time, further improving performance.

♦ Insensitive cursors are not affected by ROLLBACK or ROLLBACK TO SAVEPOINT.

Sensitive cursors

These cursors have sensitive membership, order, and values.

Sensitive cursors can be used for read-only or updatable cursor types.

Standards | Sensitive cursors correspond to the ISO/ANSI standard definition of sensitive cursors, and to ODBC dynamic cursors.

Programming interfaces

Interface	Cursor type	Comment
ODBC, OLE DB, and ADO	Dynamic	
Embedded SQL	SENSITIVE	Also supplied in response to a request for a DYNAMIC SCROLL cursor when no work table is required and PREFETCH is off.

Description

All changes are visible through the cursor, including changes through the cursor and from other transactions. Higher isolation levels may hide some changes made in other transactions because of locking.

Changes to cursor membership, order, and all column values are all visible. For example, if a sensitive cursor contains a join, and one of the values of one of the underlying tables is modified, then all result rows composed from that base row show the new value. Result set membership and order may change at each fetch.

Sensitive cursors always return rows that match the query's selection criteria, and are in the order specified by any ORDER BY clause. Updates may affect the membership, order, and values of the result set.

The requirements of sensitive cursors place restrictions on the implementation of sensitive cursors:

♦ Rows cannot be prefetched, as changes to the prefetched rows would not be visible through the cursor. This may impact performance.

♦ Sensitive cursors must be implemented without any work tables being constructed, as changes to those rows stored as work tables would not be visible through the cursor.

♦ The no work table limitation restricts the choice of join method by the optimizer and therefore may impact performance.

♦ For some queries, the optimizer is unable to construct a plan that does not include a work table that would make a cursor sensitive.

Work tables are commonly used for sorting and grouping intermediate results. A work table is not needed for sorting if the rows can be accessed through an index. It is not possible to state exactly which queries employ work tables, but the following queries do employ them:

• UNION queries, although UNION ALL do not necessarily use work tables.

• Statements with an ORDER BY clause, if there is no index on the ORDER BY column.

37

- Any query that is optimized using a hash join.

- Many queries involving DISTINCT or GROUP BY clauses.

In these cases, Adaptive Server Anywhere either returns an error to the application, or changes the cursor type to an asensitive cursor and returns a warning.

☞ For more information on query optimization and the use of work tables, see "Query Optimization and Execution" [*ASA SQL User's Guide, page 395*].

Asensitive cursors

These cursors do not have well-defined sensitivity in their membership, order, or values. The flexibility that is allowed in the sensitivity permits asensitive cursors to be optimized for performance.

Asensitive cursors are used only for read-only cursor types.

Standards   Asensitive cursors correspond to the ISO/ANSI standard definition of asensitive cursors, and to ODBC cursors with unspecific sensitivity.

Programming interfaces

Interface	Cursor type
ODBC, OLE DB, and ADO	Unspecified sensitivity
Embedded SQL	DYNAMIC SCROLL

Description   A request for an asensitive cursor places few restrictions on the methods Adaptive Server Anywhere can use to optimize the query and return rows to the application. For these reasons, asensitive cursors provide the best performance. In particular, the optimizer is free to employ any measure of materialization of intermediate results as work tables, and rows can be prefetched by the client.

Adaptive Server Anywhere makes no guarantees about the visibility of changes to base underlying rows. Some changes may be visible, others not. Membership and order may change at each fetch. In particular, updates to base rows may result in only some of the updated columns being reflected in the cursor's result.

Asensitive cursors do not guarantee to return rows that match the query's selection and order. The row membership is fixed at cursor open time, but subsequent changes to the underlying values are reflected in the results.

Asensitive cursors always return rows that matched the customer's WHERE and ORDER BY clauses at the time the cursor membership is established. If

column values change after the cursor is opened, rows may be returned that no longer match WHERE and ORDER BY clauses.

Value-sensitive cursors

These cursors are insensitive with respect to their membership, and sensitive with respect to the order and values of the result set.

Value-sensitive cursors can be used for read-only or updatable cursor types.

Standards   Value-sensitive cursors do not correspond to an ISO/ANSI standard definition. They correspond to ODBC keyset-driven cursors.

Programming interfaces

Interface	Cursor type
ODBC, OLE DB, and ADO	Keyset-driven
Embedded SQL	SCROLL
JDBC	Keyset-driven
Open Client	Keyset-driven

Description   If the application fetches a row composed of a base underlying row that has changed, then the application must be presented with the updated value, and the SQL_ROW_UPDATED status must be issued to the application. If the application attempts to fetch a row that was composed of a base underlying row that was deleted, a SQL_ROW_DELETED status must be issued to the application.

Changes to primary key values remove the row from the result set (treated as a delete, followed by an insert). A special case occurs when a row in the result set is deleted (either from cursor or outside) and a new row with the same key value is inserted. This will result in the new row replacing the old row where it appeared.

There is no guarantee that rows in the result set match the query's selection or order specification. Since row membership is fixed at open time, subsequent changes that make a row not match the WHERE clause or ORDER BY do not change a row's membership nor position.

All values are sensitive to changes made through the cursor. The sensitivity of membership to changes made through the cursor is controlled by the ODBC option SQL_STATIC_SENSITIVITY. If this option is on, then inserts through the cursor add the row to the cursor. Otherwise, they are not part of the result set. Deletes through the cursor remove the row from the result set, preventing a hole returning the SQL_ROW_DELETED status.

Value-sensitive cursors use a **key set table**. When the cursor is opened, Adaptive Server Anywhere populates a work table with identifying information for each row contributing to the result set. When scrolling through the result set, the key set table is used to identify the membership of the result set, but values are obtained, if necessary, from the underlying tables.

The fixed membership property of value-sensitive cursors allows your application to remember row positions within a cursor and be assured that these positions will not change. For more information, see "Cursor sensitivity example: a deleted row" on page 31.

♦ If a row was updated or may have been updated since the cursor was opened, Adaptive Server Anywhere returns a SQLE_ROW_UPDATED_WARNING when the row is fetched. The warning is generated only once: fetching the same row again does not produce the warning.

An update to any column of the row causes the warning, even if the updated column is not referenced by the cursor. For example, a cursor on emp_lname and emp_fname would report the update even if only the birthdate column was modified. These update warning and error conditions do not occur in bulk operations mode (-b database server option) when row locking is disabled. See "Performance aspects of bulk operations" [*ASA SQL User's Guide,* page 557].

☞ For more information, see "Row has been updated since last time read" [*ASA Error Messages,* page 322]

♦ An attempt to execute a positioned update or delete on a row that has been modified since it was last fetched returns a SQLE_ROW_UPDATED_SINCE_READ error and cancels the statement. An application must FETCH the row again before the UPDATE or DELETE is permitted.

An update to any column of the row causes the error, even if the updated column is not referenced by the cursor. The error does not occur in bulk operations mode.

☞ For more information, see "Row has changed since last read – operation canceled" [*ASA Error Messages,* page 322].

♦ If a row has been deleted after the cursor is opened, either through the cursor or from another transaction, a **hole** is created in the cursor. The membership of the cursor is fixed, so a row position is reserved, but the DELETE operation is reflected in the changed value of the row. If you fetch the row at this hole, you receive a `No Current Row of Cursor` error (SQL state 24503), indicating that there is no current row, and the

cursor is left positioned on the hole. You can avoid holes by using sensitive cursors, as their membership changes along with the values.

☞ For more information, see "No current row of cursor" [*ASA Error Messages,* page 286].

Rows cannot be prefetched for value-sensitive cursors. This requirement may impact performance in some cases.

Inserting multiple rows   When inserting multiple rows through a value-sensitive cursor, the new rows appear at the end of the result set. For more information, see "Modifying rows through a cursor" on page 23.

Cursor sensitivity and performance

There is a trade-off between performance and other cursor properties. In particular, making a cursor updatable places restrictions on the cursor query processing and delivery that constrain performance. Also, putting requirements on cursor sensitivity may constrain cursor performance.

To understand how the updatability and sensitivity of cursors affects performance, you need to understand how the results that are visible through a cursor are transmitted from the database to the client application.

In particular, results may be stored at two intermediate locations for performance reasons:

♦ **Work tables** Either intermediate or final results may be stored as work tables. Value-sensitive cursors employ a work table of primary key values. Query characteristics may also lead the optimizer to use work tables in its chosen execution plan.

♦ **Prefetching** The client side of the communication may retrieve rows into a buffer on the client side to avoid separate requests to the database server for each row.
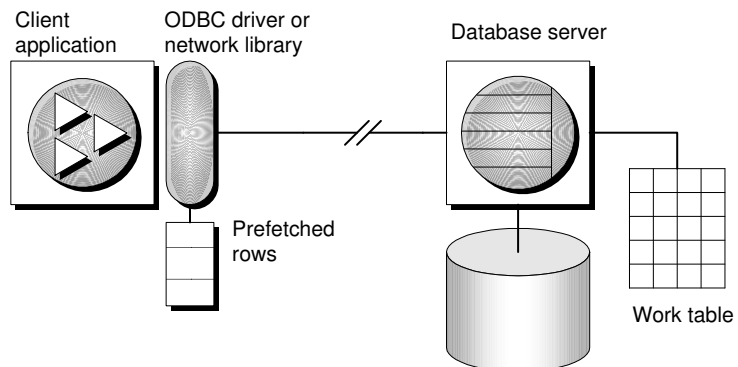
Sensitivity and updatability limit the use of intermediate locations.

Any updatable cursor is prevented from using work tables and from prefetching results. If either of these were used, the cursor would be vulnerable to lost updates. The following example illustrates this problem:

1. An application opens a cursor on the following query against the sample database.

```
SELECT id, quantity
FROM product
```

id	quantity
300	28
301	54
302	75
…	…

2. The application fetches the row with id = 300 through the cursor.

3. A separate transaction updates the row is updated using the following statement:

```
UPDATE product
SET quantity = quantity - 10
WHERE id = 300
```

4. The application updates the row through the cursor to a value of (quantity - 5).

5. The correct final value for the row would be 13. If the cursor had prefetched the row, the new value of the row would be 23. The update from the separate transaction is lost.

Similar restrictions govern sensitivity. For more information, see the descriptions of distinct cursor types.

Prefetching rows

Prefetches and multiple-row fetches are different. Prefetches can be carried out without explicit instructions from the client application. Prefetching retrieves rows from the server into a buffer on the client side, but does not make those rows available to the client application until the application fetches the appropriate row.

By default, the Adaptive Server Anywhere client library prefetches multiple rows whenever an application fetches a single row. The Adaptive Server Anywhere client library stores the additional rows in a buffer.

Prefetching assists performance by cutting down on client/server traffic, and increases throughput by making many rows available without a separate request to the server for each row or block of rows.

☞ For more information on controlling prefetches, see "PREFETCH option [database]" [*ASA Database Administration Guide,* page 682].

Controlling prefetching from an application

♦ The PREFETCH option controls whether or not prefetching occurs. You can set the PREFETCH option to ON or OFF for a single connection. By default, it is set to ON.

♦ In embedded SQL, you can control prefetching on a per-cursor basis when you open a cursor on an individual FETCH operation using the BLOCK clause.

The application can specify a maximum number of rows contained in a single fetch from the server by specifying the BLOCK clause. For example, if you are fetching and displaying 5 rows at a time, you could use BLOCK 5. Specifying BLOCK 0 fetches 1 record at a time and also causes a FETCH RELATIVE 0 to always fetch the row from the server again.

Although you can also turn off prefetch by setting a connection parameter on the application, it is more efficient to set BLOCK=0 than to set the PREFETCH option to OFF.

☞ For more information, see "PREFETCH option [database]" [*ASA Database Administration Guide,* page 682]

♦ In Open Client, you can control prefetching behavior using **ct_cursor** with CS_CURSOR_ROWS after the cursor is declared, but before it is opened.

Cursor sensitivity and isolation levels

Both cursor sensitivity and transaction isolation levels address the problem of concurrency, but in different ways.

By choosing an isolation level for a transaction (often at the connection level), you determine when locks are placed on rows in the database. Locks prevent other transactions from accessing or modifying values in the database.

By choosing a cursor sensitivity, you determine which changes are visible to the application using the cursor. By setting cursor sensitivity you are not

determining when locks are placed on rows in the database, and you do not limit the changes that can be made to the database itself.

Describing result sets

Some applications build SQL statements which cannot be completely specified in the application. In some cases, for example, statements depend on a response from the user before the application knows exactly what information to retrieve, such as when a reporting application allows a user to select which columns to display.

In such a case, the application needs a method for retrieving information about both the nature of the **result set** and the contents of the result set. The information about the nature of the result set, called a **descriptor**, identifies the data structure, including the number and type of columns expected to be returned. Once the application has determined the nature of the result set, retrieving the contents is straightforward.

This **result set metadata** (information about the nature and content of the data) is manipulated using descriptors. Obtaining and managing the result set metadata is called **describing**.

Since cursors generally produce result sets, descriptors and cursors are closely linked, although some interfaces hide the use of descriptors from the user. Typically, statements needing descriptors are either SELECT statements or stored procedures that return result sets.

A sequence for using a descriptor with a cursor-based operation is as follows:

1. Allocate the descriptor. This may be done implicitly, although some interfaces allow explicit allocation as well.

2. Prepare the statement.

3. Describe the statement. If the statement is a stored procedure call or batch, and the result set is not defined by a result clause in the procedure definition, then the describe should occur after opening the cursor.

4. Declare and open a cursor for the statement (embedded SQL) or execute the statement.

5. Get the descriptor and modify the allocated area if necessary. This is often done implicitly.

6. Fetch and process the statement results.

7. Deallocate the descriptor.

8. Close the cursor.

9. Drop the statement. Some interfaces do this automatically.

♦ In embedded SQL, a SQLDA (SQL Descriptor Area) structure holds the descriptor information.

☞ For more information, see "The SQL descriptor area (SQLDA)" on page 181.

♦ In ODBC, a descriptor handle allocated using **SQLAllocHandle** provides access to the fields of a descriptor. You can manipulate these fields using **SQLSetDescRec**, **SQLSetDescField**, **SQLGetDescRec**, and **SQLGetDescField**.

Alternatively, you can use **SQLDescribeCol** and **SQLColAttributes** to obtain column information.

♦ In Open Client, you can use **ct_dynamic** to prepare a statement and **ct_describe** to describe the result set of the statement. However, you can also use **ct_command** to send a SQL statement without preparing it first and use **ct_results** to handle the returned rows one by one. This is the more common way of operating in Open Client application development.

♦ In JDBC, the **java.SQL.ResultSetMetaData** class provides information about result sets.

♦ You can also use descriptors for sending data to the engine (for example, with the INSERT statement); however, this is a different kind of descriptor than for result sets.

☞ For more information about input and output parameters of the DESCRIBE statement, see the "DESCRIBE statement [ESQL]" [*ASA SQL Reference,* page 449].

Controlling transactions in applications

Transactions are sets of atomic SQL statements. Either all statements in the transaction are executed, or none. This section describes a few aspects of transactions in applications.

☞ For more information about transactions, see "Using Transactions and Isolation Levels" [*ASA SQL User's Guide,* page 101].

Setting autocommit or manual commit mode

Database programming interfaces can operate in either **manual commit** mode or **autocommit** mode.

♦ **Manual commit mode**   Operations are committed only when your application carries out an explicit commit operation or when the database server carries out an automatic commit, for example when executing an ALTER TABLE statement or other data definition statement. Manual commit mode is also sometimes called **chained mode**.

To use transactions in your application, including nested transactions and savepoints, you must operate in manual commit mode.

♦ **Autocommit mode**   Each statement is treated as a separate transaction. Autocommit mode is equivalent to appending a COMMIT statement to the end of each of your commands. Autocommit mode is also sometimes called **unchained mode**.

Autocommit mode can affect the performance and behavior of your application. Do not use autocommit if your application requires transactional integrity.

☞ For information on autocommit impact on performance, see "Turn off autocommit mode" [*ASA SQL User's Guide,* page 169].

Controlling autocommit behavior

The way to control the commit behavior of your application depends on the programming interface you are using. The implementation of autocommit may be client-side or server-side, depending on the interface.

☞ For more information, see "Autocommit implementation details" on page 48.

❖ **To control autocommit mode (ODBC)**

1. By default, ODBC operates in autocommit mode. The way you turn off autocommit depends on whether you are using ODBC directly, or using an application development tool. If you are programming directly to the ODBC interface, set the SQL_ATTR_AUTOCOMMIT connection attribute.

❖ **To control autocommit mode (ADO.NET)**

1. By default, the ADO.NET provider operates in autocommit mode. To use explicit transactions, use the AsaConnection.BeginTransaction method.

   ☞ For more information, see "Transaction processing" on page 386.

❖ **To control autocommit mode (JDBC)**

1. By default, JDBC operates in autocommit mode. To turn off autocommit, use the **setAutoCommit** method of the connection object:

   ```
   conn.setAutoCommit( false );
   ```

❖ **To control autocommit mode (Open Client)**

1. By default, a connection made through Open Client operates in autocommit mode. You can change this behavior by setting the CHAINED database option to ON in your application using a statement such as the following:

   ```
   SET OPTION CHAINED='ON'
   ```

❖ **To control autocommit mode (embedded SQL)**

1. By default, embedded SQL applications operate in manual commit mode. To turn on autocommit, set the CHAINED database option to OFF using a statement such as the following:

   ```
   SET OPTION CHAINED='OFF'
   ```

Autocommit implementation details

The previous section, "Controlling autocommit behavior" on page 47, describes how autocommit behavior can be controlled from each of the Adaptive Server Anywhere programming interfaces. Autocommit mode has slightly different behavior depending on the interface you are using and how you control the autocommit behavior.

Autocommit mode can be implemented in one of two ways:

♦ **Client-side autocommit**   When an application uses autocommit, the client-library sends a COMMIT statement after each SQL statement executed.

Adaptive Server Anywhere uses client-side autocommit for ODBC and OLE DB applications.

♦ **Server-side autocommit**   When an application uses autocommit, the database server issues a commit after each SQL statement. This behavior is controlled, implicitly in the case of JDBC, by the CHAINED database option.

Adaptive Server Anywhere uses server-side autocommit for embedded SQL, JDBC, and Open Client applications.

There is a difference between client-side and server-side autocommit in the case of compound statements such as stored procedures or triggers. From the client side, a stored procedure is a single statement, and so autocommit sends a single commit statement after the whole procedure is executed. From the database server perspective, the stored procedure may be composed of many SQL statements, and so server-side autocommit issues a COMMIT after each SQL statement within the procedure.

Do not mix client-side and server-side implementations
Do not combine use of the CHAINED option with autocommit in your ODBC or OLE DB application.

Controlling the isolation level

You can set the isolation level of a current connection using the ISOLATION_LEVEL database option.

Some interfaces, such as ODBC, allow you to set the isolation level for a connection at connection time. You can reset this level later using the ISOLATION_LEVEL database option.

Cursors and transactions

In general, a cursor closes when a COMMIT is performed. There are two exceptions to this behavior:

♦ The CLOSE_ON_ENDTRANS database option is set to OFF.

♦ A cursor is opened WITH HOLD, which is the default with Open Client and JDBC.

If either of these two cases is true, the cursor remains open on a COMMIT.

ROLLBACK and cursors	If a transaction rolls back, then cursors close except for those cursors opened WITH HOLD. However, don't rely on the contents of any cursor after a rollback.
	The draft ISO SQL3 standard states that on a rollback, all cursors (even those cursors opened WITH HOLD) should close. You can obtain this behavior by setting the ANSI_CLOSE_CURSORS_AT_ROLLBACK option to ON.
Savepoints	If a transaction rolls back to a savepoint, and if the ANSI_CLOSE_CURSORS_AT_ROLLBACK option is ON, then all cursors (even those cursors opened WITH HOLD) opened after the SAVEPOINT close.
Cursors and isolation levels	You can change the isolation level of a connection during a transaction using the SET OPTION statement to alter the ISOLATION_LEVEL option. However, this change affects only closed cursors.

Introduction to Java in the Database

This chapter provides motivation and concepts for using Java in the database.

Adaptive Server Anywhere is a runtime environment for Java. Java provides an alternative to the SQL stored procedure language.

Contents

Introduction

Adaptive Server Anywhere is a **runtime environment for Java**. This means that Java classes can be executed in the database server. Building a runtime environment for Java classes into the database server provides powerful ways of adding programming logic to a database.

Java in the database offers the following:

♦ You can reuse Java components in the different layers of your application—client, middle-tier, or server—and use them wherever makes the most sense to you. Adaptive Server Anywhere becomes a platform for distributed computing.

♦ Java is a more powerful language than stored procedures for building logic into the database.

♦ Java can be used in the database without jeopardizing the integrity, security, and robustness of the database.

Separately-licensable component
Java in the database is a separately licensable component and must be ordered before you can install it. To order this component, see the card in your SQL Anywhere Studio package or see *http://www.sybase.com/detail?id=1015780*.

The SQLJ standard
Java in the database is based on the SQLJ Part 1 proposed standard. SQLJ Part 1 provides specifications for calling Java static methods as SQL stored procedures and user-defined functions.

Learning about Java in the database

The following table outlines the documentation regarding the use of Java in the database.

Title	Purpose
"Introduction to Java in the Database" on page 51 (this chapter)	Java concepts and how to apply them in Adaptive Server Anywhere.
"Using Java in the Database" on page 81	Practical steps to using Java in the database.
"JDBC Programming" on page 103	Accessing data from Java classes, including distributed computing.
"Debugging Logic in the Database" [*ASA SQL User's Guide,* page 719]	Testing and debugging Java code running in the database.

Using the Java documentation

The following table is a guide to which parts of the Java documentation apply to you, depending on your interests and background.

If you . . .	Consider reading . . .
Are new to object-oriented programming.	"A Java seminar" on page 59
Want an explanation of terms such as instantiated, field, and class method.	"A Java seminar" on page 59
Are a Java developer who wants to just get started.	"The runtime environment for Java in the database" on page 69 "Tutorial: A Java in the database exercise" on page 75
Want to know the key features of Java in the database.	"Java in the database Q & A" on page 54
Want to find out how to access data from Java.	"JDBC Programming" on page 103
Want to prepare a database for Java.	"Java-enabling a database" on page 84

Java in the database Q & A

This section describes the key features of Java in the database.

What are the key features of Java in the database?

Detailed explanations of all the following points appear in later sections.

♦ **You can run Java in the database server**   An internal Java Virtual Machine (VM) runs Java code in the database server.

♦ **You can call Java from SQL**   You can call Java functions (methods) from SQL statements. Java methods provide a more powerful language than SQL stored procedures for adding logic to the database.

♦ **You can access data from Java**   An internal JDBC driver lets you access data from Java.

♦ **You can debug Java in the database**   You can use the Adaptive Server Anywhere debugger to test and debug your Java classes in the database.

♦ **SQL is preserved**   The use of Java does not alter the behavior of existing SQL statements or other aspects of non-Java relational database behavior.

How do I store Java instructions in the database?

Java is an object-oriented language, so its instructions (source code) come in the form of classes. To execute Java in a database, you write the Java instructions outside the database and compile them outside the database into compiled classes (**byte code**) which are binary files holding Java instructions.

You then install these compiled classes into a database. Once installed, you can execute these classes in the database server as a stored procedure. For example, the following statement creates a Java stored procedure:

```
CREATE PROCEDURE insertfix()
EXTERNAL NAME 'JDBCExample.InsertFixed ()V'
LANGUAGE JAVA;
```

Adaptive Server Anywhere is a runtime environment for Java classes, not a Java development environment. You need a Java development environment, such as the Sun Microsystems Java Development Kit, to write and compile Java.

☞ For more information, see "Installing Java classes into a database" on page 89.

How does Java get executed in a database?

Adaptive Server Anywhere includes a **Java Virtual Machine** (**VM**) which runs in the database environment. The Adaptive Server Anywhere Java VM interprets compiled Java instructions and runs them in the database server.

In addition to the VM, the SQL request processor in the database server has been extended so it can call into the VM to execute Java instructions. It can also process requests from the VM to enable data access from Java.

Differences from a standalone VM

There is a difference between executing Java code using a standard VM such as the Sun Java VM *java.exe* and executing Java code in the database. The Sun VM runs from a command line, while the Adaptive Server Anywhere Java VM is available at all times to perform a Java operation whenever it is required as part of the execution of a SQL statement.

You cannot access the Java VM externally. It is only used when the execution of a SQL statement requires a Java operation to take place. The database server starts the VM automatically when needed: you do not have to take any explicit action to start or stop the VM.

Why Java?

Java provides a number of features that make it ideal for use in the database:

♦ Thorough error checking at compile time.

♦ Built-in error handing with a well-defined error handling methodology.

♦ Built-in garbage collection (memory recovery).

♦ Elimination of many bug-prone programming techniques.

♦ Strong security features.

♦ Java code is interpreted, so no operations get executed without being acceptable to the VM.

On what platforms is Java in the database supported?

Java in the database is not supported on Windows CE. It is supported on other Windows operating systems, UNIX, and NetWare.

How do I use Java and SQL together?

Java methods are declared as stored procedures, and can then be called just like SQL stored procedures.

You can use many of the classes that are part of the Java API as included in the Sun Microsystems Java Development Kit. You can also use classes created and compiled by Java developers.

How do I access Java from SQL?

You can treat Java methods as stored procedures, which can be called from SQL.

You must create a stored procedure that runs your method. For example:

```
CREATE PROCEDURE javaproc()
EXTERNAL NAME 'JDBCExample.MyMethod ()V'
LANGUAGE JAVA;
```

☞ For more information, see "CREATE PROCEDURE statement" [*ASA SQL Reference,* page 373].

For example, the SQL function PI(*) returns the value for pi. The Java API class **java.lang.Math** has a parallel field named PI returning the same value. But **java.lang.Math** also has a field named E that returns the base of the natural logarithms, as well as a method that computes the remainder operation on two arguments as prescribed by the IEEE 754 standard.

Other members of the Java API offer even more specialized functionality. For example, **java.util.Stack** generates a last-in, first-out queue that can store ordered lists; **java.util.HashTable** maps values to keys; **java.util.StringTokenizer** breaks a string of characters into individual word units.

Which Java classes are supported?

The database does not support all Java API classes. Some classes, for example the *java.awt* package containing user interface components for applications, are inappropriate inside a database server. Other classes, including parts of *java.io*, deal with writing information to disk, and this also is unsupported in the database server environment.

How can I use my own Java classes in databases?

You can install your own Java classes into a database. For example, a developer could design, write in Java, and compile with a Java compiler a user-created Employee class or Package class.

User-created Java classes can contain both information about the subject and some computational logic. Once installed in a database, Adaptive Server Anywhere lets you use these classes in all parts and operations of the

database and execute their functionality (in the form of class or instance methods) as easily as calling a stored procedure.

> **Java classes and stored procedures are different**
> Java classes are different from stored procedures. Whereas stored procedures are written in SQL, Java classes provide a more powerful language, and can be called from client applications as easily and in the same way as stored procedures.

☞ For more information, see "Installing Java classes into a database" on page 89.

Can I access data using Java?

The JDBC interface is an industry standard, designed specifically to access database systems. The JDBC classes are designed to connect to a database, request data using SQL statements, and return result sets that can be processed in the client application.

Normally, client applications use JDBC classes, and the database system vendor supplies a JDBC driver that allows the JDBC classes to establish a connection.

You can connect from a client application to Adaptive Server Anywhere via JDBC, using jConnect or the iAnywhere JDBC driver. Adaptive Server Anywhere also provides an internal JDBC driver which permits Java classes installed in a database to use JDBC classes that execute SQL statements.

☞ For more information, see "JDBC Programming" on page 103.

Can I move classes from client to server?

You can create Java classes that can be moved between levels of an enterprise application. The same Java class can be integrated into either the client application, a middle tier, or the database—wherever is most appropriate.

You can move a class containing business logic to any level of the enterprise system, including the server, allowing you complete flexibility to make the most appropriate use of resources. It also enables enterprise customers to develop their applications using a single programming language in a multi-tier architecture with unparalleled flexibility.

What can I not do with Java in the database?

Adaptive Server Anywhere is a runtime environment for Java classes, not a

Java development environment.

You cannot carry out the following tasks in the database:

♦ Edit class source files (*.java* files).

♦ Compile Java class source files (*.java* files).

♦ Execute unsupported Java APIs, such as applet and visual classes.

♦ Execute Java methods that require the execution of native methods. All user classes installed into the database must be 100% Java.

The Java classes used in Adaptive Server Anywhere must be written and compiled using a Java application development tool, and then installed into a database for use, testing, and debugging.

A Java seminar

This section introduces key Java concepts. After reading this section you should be able to examine Java code, such as a simple class definition or the invocation of a method, and understand what is taking place.

> **Java samples directory**
> Some of the classes used as examples in this manual are located in the Java samples directory, which is the *Samples\ASA\Java* subdirectory of your SQL Anywhere directory.
>
> Two files represent each Java class example: the Java source and the compiled class. You can immediately install to a database (without modification) the compiled version of the Java class examples.

Understanding Java classes

A Java class combines data and functionality—the ability to hold information and perform computational operations. One way of understanding the concept of a class is to view it as an entity, an abstract representation of a thing.

You could design an Invoice class, for example, to mimic paper invoices, such as those used every day in business operations. Just as a paper invoice contains certain information (line-item details, who is being invoiced, the date, payment amount, payment due-date), so also does an instance of an Invoice class. Classes hold information in fields.

In addition to describing data, a class can make calculations and perform logical operations. For example, the Invoice class could calculate the tax on a list of line items for every Invoice object, and add it to the sub total to produce a final total, without any user intervention. Such a class could also ensure all essential pieces of information are present in the Invoice and even indicate when payment is over due or partially paid. Calculations and other logical operations are carried out by the *methods* of the class.

Example   The following Java code declares a class called Invoice. This class declaration would be stored in a file named *Invoice.java*, and then compiled into a Java class using a Java compiler.

```
public class Invoice {
  // So far, this class does nothing and knows nothing
}
```

The **class** keyword is used, followed by the name of the class. There is an opening and closing brace: everything declared between the braces, such as fields and methods, becomes part of the class.

In fact, no Java code exists outside class declarations. Even the Java procedure that a Java interpreter runs automatically to create and manage other objects—the **main** method that is often the start of your application—is itself located within a class declaration.

The Java runtime classes (*rt.jar* or *classes.zip*) in the database are built with 1.1 target as installed by JDKs. Any target other than 1.1 will result in ClassFormatErrors. When compiling classes for installation into the database with JDK version, the Java compiler option "target" should be set to 1.1. For example:

```
javac -target 1.1 Invoice.java
```

When using JDK 1.3, and earlier, the default target for the compiler is already 1.1.

Subclasses in Java

You can define classes as **subclasses** of other classes. A class that is a subclass of another class can use the fields and method of its parent: this is called **inheritance**. You can define additional methods and fields that apply only to the subclass, and redefine the meaning of inherited fields and methods.

Java is a single-hierarchy language, meaning that all classes you create or use eventually inherit from one class. This means the low-level classes (classes further up in the hierarchy) must be present before higher-level classes can be used. The base set of classes required to run Java applications is called the **runtime Java classes**, or the **Java API**.

Understanding Java objects

A **class** is a template that defines what an object is capable of doing, just as

an invoice form is a template that defines what information the invoice should contain.

Classes contain no specific information about objects. Rather, your application creates, or **instantiates**, objects based on the class (template), and the objects hold the data or perform calculations. The instantiated object is an **instance** of the class. For example, an Invoice object is an instance of the Invoice class. The class defines what the object is capable of but the object is the incarnation of the class that gives the class meaning and usefulness.

In the invoice example, the invoice form defines what all invoices based on that form can accomplish. There is one form and zero or many invoices based on the form. The form contains the definition but the invoice encapsulates the usefulness.

The Invoice object is created, stores information, is stored, retrieved, edited, updated, and so on.

Just as one invoice template can create many invoices, with each invoice separate and distinct from the other in its details, you can generate many objects from one class.

Methods and fields A **method** is a part of a class that does something—a function that performs a calculation or interacts with other objects—on behalf of the class. Methods can accept arguments, and return a value to the calling function. If no return value is necessary, a method can return **void**. Classes can have any number of methods.

A **field** is a part of a class that holds information. When you create an object of type *JavaClass*, the fields in *JavaClass* hold the state unique to that object.

Class constructors

You create an object by invoking a class constructor. A **constructor** is a method that has the following properties:

♦ A constructor method has the same name as the class, and has no declared data type. For example, a simple constructor for the Product class would be declared as follows:

```
Product () {
...constructor code here...
}
```

♦ If you include no constructor method in your class definition, a default method is used that is provided by the Java base object.

♦ You can supply more than one constructor for each class, with different

numbers and types of arguments. When a constructor is invoked, the one with the proper number and type of arguments is used.

Understanding fields

There are two categories of Java fields:

- ♦ **Instance fields**   Each object has its own set of instance fields, created when the object was created. They hold information specific to that instance. For example, a **lineItem1Description** field in the Invoice class holds the description for a line item on a particular invoice. You can access instance fields only through an object reference.

- ♦ **Class fields**   A class field holds information that is independent of any particular instance. A class field is created when the class is first loaded, and no further instances are created no matter how many objects are created. Class fields can be accessed either through the class name or the object reference.

To declare a field in a class, state its type, then its name, followed by a semicolon. To declare a class field, use the **static** Java keyword in the declaration. You declare fields in the body of the class and not within a method; declaring a variable within a method makes it a part of the method, not of the class.

Examples

The following declaration of the class Invoice has four fields, corresponding to information that might be contained on two line items on an invoice.

```java
public class Invoice {

    // Fields of an invoice contain the invoice data
    public String lineItem1Description;
    public int lineItem1Cost;

    public String lineItem2Description;
    public int lineItem2Cost;

}
```

Understanding methods

There are two categories of Java methods:

- ♦ **Instance methods**   A **totalSum** method in the Invoice class could calculate and add the tax, and return the sum of all costs, but would only be useful if it is called in conjunction with an **Invoice** object, one that had values for its line item costs. The calculation can only be performed for

an object, since the object (not the class) contains the line items of the invoice.

♦ **Class methods** Class methods (also called **static methods**) can be invoked without first creating an object. Only the name of the class and method is necessary to invoke a class method.

Similar to instance methods, class methods accept arguments and return values. Typically, class methods perform some sort of utility or information function related to the overall functionality of the class.

Class methods cannot access instance fields.

To declare a method, you state its return type, its name and any parameters it takes. Like a class declaration, the method uses an opening and closing brace to identify the body of the method where the code goes.

```
public class Invoice {

    // Fields
    public String lineItem1Description;
    public double lineItem1Cost;

    public String lineItem2Description;
    public double lineItem2Cost;

    // A method
    public double totalSum() {
        double runningsum;

        runningsum = lineItem1Cost + lineItem2Cost;
        runningsum = runningsum * 1.15;

        return runningsum;
    }
}
```

Within the body of the **totalSum** method, a variable named **runningsum** is declared. First, this holds the sub total of the first and second line item cost. This sub total is then multiplied by 15 per cent (the rate of taxation) to determine the total sum.

The local variable (as it is known within the method body) is then returned to the calling function. When you invoke the **totalSum** method, it returns the sum of the two line item cost fields plus the cost of tax on those two items.

Example   The **parseInt** method of the **java.lang.Integer** class, which is supplied with Adaptive Server Anywhere, is one example of a class method. When given a string argument, the **parseInt** method returns the integer version of the string.

For example given the string value "1", the **parseInt** method returns 1, the

integer value, without requiring an instance of the **java.lang.Integer** class to first be created, as illustrated by this Java code fragment:

```
String num = "1";
int i = java.lang.Integer.parseInt( num );
```

Example The following version of the Invoice class now includes both an instance method and a class method. The class method named **rateOfTaxation** returns the rate of taxation used by the class to calculate the total sum of the invoice.

The advantage of making the **rateOfTaxation** method a class method (as opposed to an instance method or field) is that other classes and procedures can use the value returned by this method without having to create an instance of the class first. Only the name of the class and method is required to return the rate of taxation used by this class.

Making **rateofTaxation** a method, as opposed to a field, allows the application developer to change how the rate is calculated without adversely affecting any objects, applications, or procedures that use its return value. Future versions of Invoice could make the return value of the **rateOfTaxation** class method based on a more complicated calculation without affecting other methods that use its return value.

```
public class Invoice {
    // Fields
    public String lineItem1Description;
    public double lineItem1Cost;
    public String lineItem2Description;
    public double lineItem2Cost;
    // An instance method
    public double totalSum() {
        double runningsum;
        double taxfactor = 1 + Invoice.rateOfTaxation();

        runningsum = lineItem1Cost + lineItem2Cost;
        runningsum = runningsum * taxfactor;

        return runningsum;
    }
    // A class method
    public static double rateOfTaxation() {
        double rate;
        rate = .15;

        return rate;
    }
}
```

Object oriented and procedural languages

If you are more familiar with procedural languages such as C, or the SQL stored procedure language, than object-oriented languages, this section explains some of the key similarities and differences between procedural and object-oriented languages.

Java is based on classes   The main structural unit of code in Java is a **class**.

A Java class could be looked at as just a collection of procedures and variables that have been grouped together because they all relate to a specific, identifiable category.

However the manner in which a class gets used sets object-oriented languages apart from procedural languages. When an application written in a procedural language is executed, it is typically loaded into memory once and takes the user down a pre-defined course of execution.

In object-oriented languages such as Java, a class is used like a template: a definition of potential program execution. Multiple copies of the class can be created and loaded dynamically, as needed, with each instance of the class capable of containing its own data, values, and course of execution. Each loaded class could be acted on or executed independently of any other class loaded into memory.

A class that is loaded into memory for execution is said to have been instantiated. An instantiated class is called an object: it is an application derived from the class that is prepared to hold unique values or have its methods executed in a manner independent of other class instances.

A Java glossary

The following items outline some of the details regarding Java classes. It is by no means an exhaustive source of knowledge about the Java language, but may aid in the use of Java classes in Adaptive Server Anywhere.

☞ For more information about the Java language, see the online book *Thinking in Java*, by Bruce Eckel, included with Adaptive Server Anywhere in the file *Samples\ASA\Java\Tjava.pdf*.

Packages   A **package** is a grouping of classes that share a common purpose or category. One member of a package has special privileges to access data and methods in other members of the package, hence the **protected** access modifier.

A package is the Java equivalent of a library. It is a collection of classes which can be made available using the **import** statement. The following Java statement imports the utility library from the Java API:

```
import java.util.*
```

Packages are typically held in JAR files, which have the extension *.jar* or *.zip*.

Public versus private

An access modifier determines the visibility (essentially the **public**, **private,** or **protected** keyword used in front of any declaration) of a field, method or class to other Java objects.

♦ A **public** class, method, or field is visible everywhere.

♦ A **private** class, method, or field is visible only in methods defined within that class.

♦ A **protected** method or field is visible to methods defined within that class, within sublclasses of the class, or within other classes in the same package.

♦ The default visibility, known as package, means that the method or field is visible within the class and to other classes in the same package.

Constructors

A constructor is a special method of a Java class that is called when an instance of the class is created.

Classes can define their own constructors, including multiple, overriding constructors. Which arguments were used in the attempt to create the object determine which constructor is used. When the type, number, and order of arguments used to create an instance of the class match one of the class's constructors, that constructor is used when creating the object.

Garbage collection

Garbage collection automatically removes any object with no references to it, with the exception of objects stored as values in a table.

There is no such thing as a destructor method in Java (as there is in C++). Java classes can define their own **finalize** method for clean up operations when an object is discarded during garbage collection.

Interfaces

Java classes can inherit only from one class. Java uses interfaces instead of multiple-inheritance. A class can implement multiple interfaces. Each **interface** defines a set of methods and method profiles that must be implemented by the class for the class to be compiled.

An interface defines what methods and static fields the class must declare. The implementation of the methods and fields declared in an interface is located within the class that uses the interface: the interface defines what the class must declare; it is up to the class to determine how it is implemented.

Java error handling

Java error handling code is separate from the code for normal processing.

Errors generate an exception object representing the error. This is called **throwing an exception**. A thrown exception terminates a Java program unless it is caught and handled properly at some level of the application.

Both Java API classes and custom-created classes can throw exceptions. In fact, users can create their own exception classes which throw their own custom-created classes.

If there is no exception handler in the body of the method where the exception occurred, then the search for an exception handler continues up the call stack. If the top of the call stack is reached and no exception handler has been found, the default exception handler of the Java interpreter running the application is called and the program terminates.

In Adaptive Server Anywhere, if a SQL statement calls a Java method, and an unhandled exception is thrown, a SQL error is generated.

Error types in Java   All errors in Java come from two types of error classes: **Exception** and **Error**. Usually, Exception-based errors are handled by error handling code in your method body. Error type errors are specifically for internal errors and resource exhaustion errors inside the Java run-time system.

Exception class errors are thrown and caught. Exception handling code is characterized by **try**, **catch**, and **finally** code blocks.

A **try** block executes code that may generate an error. A **catch** block is code that executes if the execution of a **try** block generates (or throws) an error.

A **finally** block defines a block of code that executes regardless of whether an error was generated and caught and is typically used for cleanup operations. It is used for code that, under no circumstances, can be omitted.

There are two types of exception class errors: those that are runtime exceptions and those that are not runtime exceptions.

Errors generated by the runtime system are known as implicit exceptions, in that they do not have to be explicitly handled as part of every class or method declaration.

For example, an array out of bounds exception can occur whenever an array is used, but the error does not have to be part of the declaration of the class or method that uses the array.

All other exceptions are explicit. If the method being invoked can throw an error, it must be explicitly caught by the class using the exception-throwing

method, or this class must explicitly throw the error itself by identifying the exception it may generate in its class declaration. Essentially, explicit exceptions must be dealt with explicitly. A method must declare all the explicit errors it throws, or catch all the explicit errors that may potentially be thrown.

Non-runtime exceptions are checked at compile time. Java catches many such errors during compilation, before running the code.

Every Java method is given an alternative path of execution so that all Java methods complete, even if they are unable to complete normally. If the type of error thrown is not caught, it's passed to the next code block or method in the stack.

The runtime environment for Java in the database

This section describes the Adaptive Server Anywhere runtime environment for Java, and how it differs from a standard Java runtime environment.

Supported versions of Java and JDBC

The Java VM provides you with the choice of using the JDK 1.1, JDK 1.2, or JDK 1.3 programming interfaces. The specific versions provided are JDK versions 1.1.8 and 1.3.

Between release 1.0 of the JDK and release 1.1, several new APIs were introduced. As well, a number were deprecated—the use of certain APIs became no longer recommended and support for them may be dropped in future releases.

A Java class file using deprecated APIs generates a warning when compiled, but does still execute on a Java virtual machine built to release 1.1 standards, such as the Adaptive Server Anywhere VM.

The internal JDBC driver supports JDBC version 2.

☞ For information on how to create a database that supports Java, see "Java-enabling a database" on page 84.

The runtime Java classes

The runtime Java classes are the low-level classes that are made available to a database when it is created or Java-enabled. These classes include a subset of the Java API. These classes are part of the Sun Java Development Kit.

The runtime classes provide basic functionality on which to build applications. The runtime classes are always available to classes in the database.

You can incorporate the runtime Java classes in your own user-created classes: either inheriting their functionality or using it within a calculation or operation in a method.

Examples
Some Java API classes included in the runtime Java classes include:

♦ **Primitive Java data types**   All primitive (native) data types in Java have a corresponding class. In addition to being able to create objects of these types, the classes have additional, often useful, functionality.

The Java **int** data type has a corresponding class in **java.lang.Integer**.

♦ **The utility package**   The package **java.util.*** contains a number of very helpful classes whose functionality has no parallel in the SQL functions

available in Adaptive Server Anywhere.

Some of the classes include:

- **Hashtable**   which maps keys to values.

- **StringTokenizer**   which breaks a String down into individual words.

- **Vector**   which holds an array of objects whose size can change dynamically

- **Stack**   which holds a last-in, first-out stack of objects.

♦ **JDBC for SQL operations**   The package **java.SQL.*** contains the classes needed by Java objects to extract data from the database using SQL statements.

Unlike user-defined classes, the runtime classes are not stored in the database. Instead, they are stored in files in the *java* subdirectory of the Adaptive Server Anywhere installation directory.

User-defined classes

User-defined classes are installed into a database using the INSTALL JAVA statement. Once installed, they become available to other classes in the database. If they are public classes, they are available from SQL as domains.

Java is case sensitive

Java syntax works as you would expect it to, and SQL syntax is unaltered by the presence of Java classes. This is true even if the same SQL statement contains both Java and SQL syntax. It's a simple statement, but with far-reaching implications.

Java is case sensitive. The Java class **FindOut** is a completely different class from the class **Findout**. SQL is case insensitive with respect to keywords and identifiers.

Java case sensitivity is preserved even when embedded in a SQL statement that is case insensitive. The Java parts of the statement must be case sensitive, even though the parts previous to and following the Java syntax can be in either upper or lower case.

For example, the following SQL statements successfully execute because the case of Java objects, classes, and operators is respected even though there is variation in the case of the remaining SQL parts of the statement.

```
SeLeCt java.lang.Math.random();
```

Strings in Java and SQL

A set of double quotes identifies string literals in Java, as in the following Java code fragment:

```
String str = "This is a string";
```

In SQL, however, single quotes mark strings, and double quotes indicate an identifier, as illustrated by the following SQL statement:

```
INSERT INTO TABLE DBA.t1
VALUES( 'Hello' )
```

You should always use the double quote in Java source code, and single quotes in SQL statements.

For example, the following SQL statements are valid.

```
CREATE VARIABLE str char(20);
SET str = NEW java.lang.String( 'Brand new object' )
```

The following Java code fragment is also valid, if used within a Java class.

```
String str = new java.lang.String(
             "Brand new object" );
```

Printing to the command line

Printing to the standard output is a quick way of checking variable values and execution results at various points of code execution. When the method in the second line of the following Java code fragment is encountered, the string argument it accepts prints out to standard output.

```
String str = "Hello world";
System.out.println( str );
```

In Adaptive Server Anywhere, standard output is the server window, so the string appears there. Executing the above Java code within the database is the equivalent of the following SQL statement.

```
MESSAGE 'Hello world'
```

Using the main method

When a class contains a **main** method matching the following declaration, most Java run time environments, such as the Sun Java interpreter, execute it automatically. Normally, this static method executes only if it is the class being invoked by the Java interpreter

```
public static void main( String args[ ] ) { }
```

Useful for testing the functionality of Java objects, you are always guaranteed this method will be called first, when the Sun Java runtime system starts.

In Adaptive Server Anywhere, the Java runtime system is always available. The functionality of objects and methods can be tested in an ad hoc, dynamic manner using SQL statements. In many ways this is far more flexible for testing Java class functionality.

Scope and persistence

SQL variables are persistent only for the duration of the connection. This is unchanged from previous versions of Adaptive Server Anywhere, and is unaffected by whether the variable is a Java class or a native SQL data type.

The persistence of Java classes is analogous to tables in a database: tables exist in the database until you drop them, regardless of whether they hold data or even whether they are ever used. Java classes installed to a database are similar: they are available for use until you explicitly remove them with a REMOVE JAVA statement.

☞ For more information on removing classes, see "REMOVE JAVA statement" [*ASA SQL Reference,* page 576].

A class method in an installed Java class can be called at any time from a SQL statement. You can execute the following statement anywhere you can execute SQL statements.

```
SELECT java.lang.Math.abs(-342)
```

A Java object is only available in two forms: as the value of a variable, or as a value in a table.

Java escape characters in SQL statements

In Java code, you can use escape characters to insert certain special characters into strings. Consider the following code, which inserts a new line and tab in front of a sentence containing an apostrophe.

```
String str = "\n\t\This is an object\'s string literal";
```

Adaptive Server Anywhere permits the use of Java escape characters only when being used by Java classes. From within SQL, however, you must follow the rules that apply to strings in SQL.

For example, to pass a string value to a field using a SQL statement, you

72

could use the following statement (which includes SQL escape characters), but the Java escape characters could not be used.

```
SET obj.str = '\nThis is the object''s string field';
```

☞ For more information on SQL string handling rules, see "Strings" [*ASA SQL Reference,* page 9].

Use of import statements

It is common in a Java class declaration to include an import statement to access classes in another package. You can reference imported classes using unqualified class names.

For example, you can reference the Stack class of the **java.util** package in two ways:

♦ explicitly using the name **java.util.Stack**, or

♦ using the name **Stack**, and including the following import statement:

```
import java.util.*;
```

Classes further up in the hierarchy must also be installed.

A class referenced by another class, either explicitly with a fully qualified name or implicitly using an import statement, must also be installed in the database.

The import statement works as intended within compiled classes. However, within the Adaptive Server Anywhere runtime environment, no equivalent to the import statement exists. All class names used in SQL statements or stored procedures must be fully qualified. For example, to create a variable of type String, you would reference the class using the fully qualified name: **java.lang.String**.

Using the CLASSPATH variable

Sun's Java runtime environment and the Sun JDK Java compiler use the CLASSPATH environment variable to locate classes referenced within Java code. A CLASSPATH variable provides the link between Java code and the actual file path or URL location of the classes being referenced. For example, import java.io.* allows all the classes in the **java.io** package to be referenced without a fully qualified name. Only the class name is required in the following Java code to use classes from the **java.io** package. The CLASSPATH environment variable on the system where the Java class declaration is to be compiled must include the location of the Java directory, the root of the **java.io** package.

CLASSPATH ignored at runtime

The CLASSPATH environment variable does not affect the Adaptive Server

Anywhere runtime environment for Java during the execution of Java operations because the classes are stored in the database, instead of in external files or archives.

CLASSPATH used to install classes

The CLASSPATH variable can, however, be used to locate a file during the installation of classes. For example, the following statement installs a user-created Java class to a database, but only specifies the name of the file, not its full path and name. (Note that this statement involves no Java operations.)

```
INSTALL JAVA NEW
FROM FILE 'Invoice.class'
```

If the file specified is in a directory or zip file specified by the CLASSPATH environmental variable, Adaptive Server Anywhere will successfully locate the file and install the class.

Public fields

It is a common practice in object-oriented programming to define class fields as private and make their values available only through public methods.

Many of the examples used in this documentation render fields public to make examples more compact and easier to read. Using public fields in Adaptive Server Anywhere also offers a performance advantage over accessing public methods.

The general convention followed in this documentation is that a user-created Java class designed for use in Adaptive Server Anywhere exposes its main values in its fields. Methods contain computational automation and logic that may act on these fields.

Tutorial: A Java in the database exercise

This tutorial is a primer for invoking Java operations on Java classes and objects using SQL statements. It describes how to install a Java class into the database. It also describes how to access the class and its members and methods from SQL statements. The tutorial uses the Invoice class created in "A Java seminar" on page 59.

Requirements   The tutorial assumes that you have installed Java in the database software. It also assumes that you have a Java Development Kit (JDK) installed, including the Java compiler (javac).

If you do not have Java installed into the database, the following SQL can be used:

```
ALTER DATABASE UPGRADE JAVA ON;
```

You must restart the database server for this operation to take effect.

If you do not have a database, you can initialize a database with Java enabled, using the Sybase Central Wizard or dbinit for example:

```
dbinit -ja mydb.db
```

Resources   Source code and batch files for this sample are provided in the folder *samples/ASA/Java* under your SQL Anywhere folder.

Create and compile the sample Java class

The first step is to write the Java code and compile it. This is done outside the database

❖ To create and compile the class

1. Create the sample Java class.

   The sample code is installed. For your convenience, the sample code is included here. Paste the following code into *Invoice.java*:

```
public class Invoice {
    public static String lineItem1Description;
    public static double lineItem1Cost;

    public static String lineItem2Description;
    public static double lineItem2Cost;

    public static double totalSum() {
        double runningsum;
        double taxfactor = 1 + Invoice.rateOfTaxation();

        runningsum = lineItem1Cost + lineItem2Cost;
        runningsum = runningsum * taxfactor;

        return runningsum;
    }

    public static double rateOfTaxation() {
        double rate;
        rate = .15;

        return rate;
    }

    public static void init( String item1desc, double
        item1cost,

        String item2desc, double item2cost ) {
        lineItem1Description = item1desc;
        lineItem1Cost = item1cost;
        lineItem2Description = item2desc;
        lineItem2Cost = item2cost;
    }

        public static String getLineItem1Description() {
        return lineItem1Description;
    }

        public static double getLineItem1Cost() {
        return lineItem1Cost;
    }

        public static String getLineItem2Description() {
        return lineItem2Description;
    }

        public static double getLineItem2Cost() {
        return lineItem2Cost;
    }
}
```

You can find source code for this class as the file
Samples\ASA\JavaInvoice\Invoice.java under your SQL Anywhere
directory.

2. Compile the file to create the file *Invoice.class*.

From a command prompt, change to the directory *samples\ASA\Java* and execute the following command.

```
javac -target 1.1 *.java
```

The Java runtime classes (*rt.jar* or *classes.zip*) in the database are built with 1.1 target as installed by JDKs. Any target other than 1.1 will result in ClassFormatErrors. When compiling classes for installation into the database with JDK version, the Java compiler option "target" should be set to 1.1.

When using JDK 1.3, and earlier, the default target for the compiler is already 1.1.

The class is now compiled and ready to be installed into the database.

Install the sample Java class

Java classes must be installed into a database before they can be used. You can install classes from Sybase Central or Interactive SQL. This section provides instructions for both. Choose whichever you prefer.

❖ **To install the class to the sample database (Sybase Central)**

1. Start Sybase Central and connect to the sample database.

2. Open the Java Objects folder and double-click Add Java Class.

   The Java Class Creation wizard appears.

3. Use the Browse button to locate *Invoice.class* in the *Samples/ASA/Java* subfolder of your SQL Anywhere installation folder.

4. Click Finish to exit the wizard.

❖ **To install the class to the sample database (Interactive SQL)**

1. Start Interactive SQL and connect to the sample database.

2. In the SQL Statements pane of Interactive SQL, type the following command:

```
INSTALL JAVA NEW
FROM FILE
'path/samples/ASA/Java/Invoice.class'
```

where path is your SQL Anywhere folder. The class is now installed into the sample database.

Notes

♦ At this point no Java in the database operations have taken place. The class has been installed into the database and is ready for use.

♦ Changes made to the class file from now on are *not* automatically reflected in the copy of the class in the database. You must re-install the classes if you want the changes reflected.

☞ For more information on installing classes, and for information on updating an installed class, see "Installing Java classes into a database" on page 89.

Access fields and methods of the Java object

To access the Java methods in the class, you must creating stored procedures or user-defined functions that act as wrappers for the methods in the class.

❖ **To create stored procedures/functions for the methods in the class**

1. To pass arguments to a Java method, create the following SQL wrapper:

```
CREATE PROCEDURE init( IN arg1 CHAR(10),
                       IN arg2 DOUBLE,
                        IN arg3 CHAR(10),
                         IN arg4 DOUBLE)
  EXTERNAL NAME 'Invoice.init
         (Ljava/lang/String;DLjava/lang/String;D)V'
  LANGUAGE JAVA;
```

☞ For more information about the syntax of this command, see "CREATE PROCEDURE statement" [*ASA SQL Reference,* page 373].

2. To retrieve return codes from a Java method, create the following SQL wrappers:

```
CREATE FUNCTION taxationrate()
 RETURNS DOUBLE
 EXTERNAL NAME      'Invoice.rateOfTaxation ()D'
 LANGUAGE JAVA;
CREATE FUNCTION totalSum()
 RETURNS DOUBLE
 EXTERNAL NAME 'Invoice.totalSum ()D'
 LANGUAGE JAVA;
CREATE FUNCTION linedesc1()
 RETURNS CHAR(10)
 EXTERNAL NAME
      'Invoice.getLineItem1Description ()Ljava/lang/String;'
 LANGUAGE JAVA;
```

☞ For more information about the syntax of these commands, see "CREATE FUNCTION statement" [*ASA SQL Reference,* page 362].

3. To retrieve return codes from a Java attribute, create the following SQL wrapper:

```
CREATE FUNCTION linecost2()
RETURNS DOUBLE
EXTERNAL NAME
      'Invoice.getLineItem2Cost ()D'
LANGUAGE JAVA;
```

Note that there is a space between the signature and the name of the method/attribute. Also note the case of the signature of Java method is case sensitive.

We can now invoke the methods and access attributes from SQL with stored procedures and functions.

```
CALL init('prod1',10.00,'prod2',25.00);
SELECT taxationrate(),totalSum(),linedesc1(),linecost2();
```

The query returns four columns with values as follows:

Taxationrate(*)	totalSum(*)	linedesc1(*)	linecost2(*)
.15	40.25	prod1	25

CHAPTER 4

Using Java in the Database

About this chapter   This chapter describes how to add Java classes to your database, and how to
                      use these classes in a relational database.

Contents

Topic:	page

Introduction

This chapter describes how to accomplish tasks using Java in the database, including the following:

♦ **How to Java-enable a database**   You need to follow certain steps to enable your database to use Java.

♦ **Installing Java classes**   You need to install Java classes in a database to make them available for use in Adaptive Server Anywhere.

Setting up the Java sample

Some of the examples in this chapter require you to add the JDBCExamples class to the sample database.

Setting up the Java examples involves two steps:

1. Java-enable the sample database. Adaptive Server Anywhere databases are not Java-enabled by default.

2. Add the JDBCExamples class to the database.

❖ **To Java-enable the sample database**

1. Start Sybase Central and connect to the sample database (ASA 9.0 Sample ODBC data source). An asademo9 database server appears with an asademo database.

2. In the left pane of Sybase Central, right click the asademo database and choose Upgrade Database from the popup menu. The Upgrade a Database wizard appears.

3. Follow the instructions in the Upgrade a Database wizard. Choose the option to Install Java Support with a JDK version of 1.3.

4. Restart the sample database.

   When the Upgrade a Database wizard has completed, disconnect and ensure that the sample database is shut down. The database must be shut down and restarted before Java support can be used.

5. Confirm that Java support has been added:
   ♦ From Sybase Central, connect to the sample database.
   ♦ In the left pane of Sybase Central, right click the asademo database and choose Properties from the popup menu.
   ♦ Confirm that Java JDK version is set to 1.3.

❖ **To add the JDBCExamples class to the sample database**

1. Start Sybase Central and connect to the sample database (ASA 9.0 Sample ODBC data source). An asademo9 database server appears with an asademo database.

2. In the left pane of Sybase Central, open the Java Objects folder.

3. Right-click the right pane and choose New ➤ Java Class from the popup menu. The Create a New Java Class wizard appears.

4. Click Browse and locate JDBCExamples.class in the *Samples\ASA\Java* subdirectory of your SQL Anywhere installation.

5. Click OK and click Finish to complete the installation.

Managing the runtime environment for Java

The runtime environment for Java consists of:

♦ **The Adaptive Server Anywhere Java Virtual Machine**   Running within the database server, the Adaptive Server Anywhere Java Virtual Machine interprets and executes the compiled Java class files.

♦ **The runtime Java classes**   When you create a database, a set of Java classes becomes available to the database. Java applications in the database require these runtime classes to work properly.

Management tasks for Java

To provide a runtime environment for Java, you need to carry out the following tasks:

♦ **Java-enable your database**   This task involves ensuring the availability of built-in classes and the upgrading of the database to Version 9.

☞ For more information, see "Java-enabling a database" on page 84.

♦ **Install other classes your users need**   This task involves ensuring that classes other than the runtime classes are installed and up to date.

☞ For more information, see "Installing Java classes into a database" on page 89.

♦ **Configuring your server**   You must configure your server to make the necessary memory available to run Java tasks.

☞ For more information, see "Configuring memory for Java" on page 99.

Tools for managing Java

You can carry out all these tasks from Sybase Central or from Interactive SQL.

Java-enabling a database

The Adaptive Server Anywhere Runtime environment for Java requires a Java VM and the **Adaptive Server Anywhere runtime Java classes**. You need to Java-enable a database for it to be able to use the runtime Java classes.

Java in the database is a separately-licensed component of SQL Anywhere Studio.

> **New databases are not Java-enabled by default**
> By default, databases created with Adaptive Server Anywhere are not Java-enabled.

Java is a single-hierarchy language, meaning that all classes you create or use eventually inherit from one class. This means the low-level classes (classes further up in the hierarchy) must be present before you can use higher-level classes. The base set of classes required to run Java applications are the runtime Java classes, or the Java API.

When not to Java-enable a database

Java-enabling a database adds many entries into the system tables. This adds to the size of the database and, more significantly, adds about 200K to the memory requirements for running the database, even if you do not use any Java functionality.

If you are not going to use Java, and if you are running in a limited-memory environment, you may wish to not Java-enable your database.

The Adaptive Server Anywhere runtime Java classes

The Adaptive Server Anywhere runtime Java classes are held on disk rather than stored in a database like other classes. The following files contain the Adaptive Server Anywhere runtime Java classes. The files are in the *Java* subdirectory of your SQL Anywhere directory:

♦ **1.1\classes.zip** This file, licensed from Sun Microsystems, contains a subset of the Sun Microsystems Java runtime classes for JDK 1.1.8.

♦ **1.3\rt.jar** This file, licensed from Sun Microsystems, contains a subset of the Sun Microsystems Java runtime classes for JDK 1.3.

♦ **asajdbc.zip** This file contains Adaptive Server Anywhere internal JDBC driver classes for JDK 1.1.

♦ **asajrt12.zip** This file contains Adaptive Server Anywhere internal JDBC driver classes for JDK 1.2 and JDK 1.3.

When you Java-enable a database, you also update the system tables with a list of available classes from the system JAR files. You can then browse the class hierarchy from Sybase Central, but the classes themselves are not present in the database.

JAR files

The database stores runtime class names the under the following JAR files:

♦ **ASACIS**   Classes required for remote data access are stored here.

♦ **ASAJDBCDRV**   Class names from *jdbcdrv.zip* are held here. (com.sybase.jdbc package).

♦ **ASAJIO**

♦ **ASAJRT**   Class names from *asajdbc.zip* are held here.

♦ **ASASystem**   Class names from *classes.zip* are held here.

♦ **ASASystemUNIX**   Class names from *classes.zip* are held here.

Installed packages

These runtime classes include the following packages:

♦ **java**   Packages stored here include the supported Java runtime classes from Sun Microsystems.

♦ **com.sybase**   Packages stored here provide server-side JDBC support.

♦ **sun**   Sun Microsystems provides the packages stored here.

♦ **sybase.sql**   Packages stored here are part of the server-side JDBC support.

Caution: do not install classes from another version of Sun's JDK
Classes in Sun's JDK share names with the Adaptive Server Anywhere runtime Java classes that must be installed in any database intended to execute Java operations.

You must not replace the classes.zip file included with Adaptive Server Anywhere. Using another version of these classes could cause compatibility problems with the Adaptive Server Anywhere Java Virtual Machine.

You must only Java-enable a database using the methods outlined in this section.

Ways of Java-enabling a database

You can Java-enable databases when you create them, when you upgrade them, or in a separate operation at a later time.

You can create a Java-enabled database using:

♦ the CREATE DATABASE statement.

☞ For details of the syntax, see "CREATE DATABASE statement" [*ASA SQL Reference,* page 338].

♦ the *dbinit* utility.

☞ For details, see "Creating a database using the dbinit command-line utility" [*ASA Database Administration Guide,* page 531].

♦ Sybase Central.

☞ For details, see "Creating a database" [*ASA SQL User's Guide,* page 31].

You can upgrade a database to a Java-enabled Version 9 database using:

♦ the ALTER DATABASE statement.

☞ For details of the syntax, see "ALTER DATABASE statement" [*ASA SQL Reference,* page 266].

♦ the *dbupgrad.exe* upgrade utility.

☞ For details, see "Upgrading a database using the dbupgrad command-line utility" [*ASA Database Administration Guide,* page 600].

♦ Sybase Central.

☞ For details, see "Java-enabling a database" on page 87.

If you choose not to install Java in the database, all database operations not involving Java operations remain fully functional and work as expected.

New databases and Java

By default, Adaptive Server Anywhere does not install Adaptive Server Anywhere runtime Java classes each time you create a database. The installation of this separately-licensable component is optional, and controlled by the method you use to create the database.

The CREATE DATABASE SQL statement has an option called JAVA. To Java-enable a database, you can set the option to ON. To disable Java, set the option to OFF. This option is set to OFF by default.

For example, the following statement creates a Java-enabled database file named *temp.db*:

```
CREATE DATABASE 'c:\\sybase\\asa9\\temp' JAVA ON
```

The following statement creates a database file named *temp2.db*, which does not support Java.

```
CREATE DATABASE 'c:\\sybase\\asa9\\temp2'
```

Database initialization utility
You can create databases using the *dbinit.exe* database initialization utility. This utility has options that control whether or not to install the runtime Java classes in the newly-created database. By default, the classes are not installed.

The same options are available when creating databases using Sybase Central.

Upgrading databases and Java

You can upgrade existing databases created with earlier versions of the software using the Upgrade utility or the ALTER DATABASE statement.

Database upgrade utility
You can upgrade databases to Adaptive Server Anywhere Version 9 standards using the *dbupgrad.exe* utility. Using the -jr Upgrade utility option prevents the installation of Adaptive Server Anywhere runtime Java classes.

☞ For information on the conditions under which Java in the database is included in the upgraded database, see "Upgrading a database using the dbupgrad command-line utility" [*ASA Database Administration Guide, page 600*].

Java-enabling a database

If you have created a database, or upgraded a database to standards, but have chosen not to Java-enable the database, you can add the necessary Java classes at a later date, using either Sybase Central or Interactive SQL.

❖ **To add the Java runtime classes to a database (Sybase Central)**

1. Connect to the database from Sybase Central as a user with DBA authority.

2. Right-click the database and choose Upgrade Database.

3. Click Next on the introductory page of the wizard.

4. Select the database you want to upgrade from the list.

5. You can choose to create a backup of the database if you wish. Click Next.

6. You can choose to install jConnect meta-information support if you wish. Click Next.

7. Select the Install Java Support option. You must also choose which version of the JDK you want to install. The default classes are the JDK 1.3 classes.

8. Follow the remaining instructions in the wizard.

❖ **To add the Java runtime classes to a database (SQL)**

1. Connect to the database from Interactive SQL as a user with DBA authority.

2. Execute the following statement:

   ```
   ALTER DATABASE UPGRADE JAVA ON
   ```

   ☞ For more information, see "ALTER DATABASE statement" [*ASA SQL Reference,* page 266].

3. Restart the database for the Java support to take effect.

Using Sybase Central to Java-enable a database

You can use Sybase Central to create databases using wizards. During the creation or upgrade of a database, the wizard prompts you to choose whether or not you have the Adaptive Server Anywhere runtime Java classes installed. By default, this option Java-enables the database.

Using Sybase Central, you can create or upgrade a database by choosing:

♦ Choosing File ➤ Create Database to create a new database.

♦ Clicking the database server in the left pane, clicking the Utilities tab in the right pane, and double-clicking Upgrade Database to upgrade a database from a previous version of the software to a database with Java capabilities.

Installing Java classes into a database

Before you install a Java class into a database, you must compile it. You can install Java classes into a database as:

♦ **A single class**   You can install a single class into a database from a compiled class file. Class files typically have extension *.class*.

♦ **A JAR**   You can install a set of classes all at once if they are in either a compressed or uncompressed JAR file. JAR files typically have the extension *.jar* or *.zip*. Adaptive Server Anywhere supports all compressed JAR files created with the Sun JAR utility, and some other JAR compression schemes as well.

This section describes how to install Java classes once you have compiled them. You must have DBA authority to install a class or JAR.

Creating a class

Although the details of each step may differ depending on whether you are using a Java development tool, the steps involved in creating your own class generally include the following:

❖ **To create a class**

1. **Define your class**   Write the Java code that defines your class. If you are using the Sun Java SDK then you can use a text editor. If you are using a development tool, the development tool provides instructions.

   | **Use only supported classes** |
   | User classes must be 100% Java. Native methods are not allowed. |

2. **Name and save your class**   Save your class declaration (Java code) in a file with the extension *.java*. Make certain the name of the file is the same as the name of the class and that the case of both names is identical.

   For example, a class called Utility should be saved in a file called *Utility.java*.

3. **Compile your class**   This step turns your class declaration containing Java code into a new, separate file containing byte code. The name of the new file is the same as the Java code file but has an extension of *.class*. You can run a compiled Java class in a Java runtime environment, regardless of the platform you compiled it on or the operating system of the runtime environment.

   The Sun JDK contains a Java compiler, *Javac.exe*.

> **Java-enabled databases only**
>
> You can install any compiled Java class file in a database. However, Java operations using an installed class can only take place if the database has been Java-enabled as described in "Java-enabling a database" on page 84.

Installing a class

To make your Java class available within the database, you **install** the class into the database either from Sybase Central, or using the INSTALL JAVA statement from Interactive SQL or other application. You must know the path and file name of the class you wish to install.

You require DBA authority to install a class.

❖ **To install a class (Sybase Central)**

1. Connect to a database with DBA authority.

2. Open the Java Objects folder for the database.

3. Right-click in the right pane and choose New ➤ Java Class from the popup menu.

4. Follow the instructions in the wizard.

❖ **To install a class (SQL)**

1. Connect to a database with DBA authority.

2. Execute the following statement:

   ```
   INSTALL JAVA NEW
   FROM FILE 'path\\ClassName.class'
   ```

   where *path* is the directory where the class file is, and *ClassName.class* is the name of the class file.

   The double backslash ensures that the backslash is not treated as an escape character.

   For example, to install a class in a file named *Utility.class*, held in the directory *c:\source*, you would enter the following statement:

   ```
   INSTALL JAVA NEW
   FROM FILE 'c:\\source\\Utility.class'
   ```

   If you use a relative path, it must be relative to the current working directory of the database server.

   ☞ For more information, see "INSTALL JAVA statement" [*ASA SQL Reference,* page 532].

Installing a JAR

It is useful and common practice to collect sets of related classes together in packages, and to store one or more packages in a **JAR file**.

You install a JAR file the same way as you install a class file. A JAR file can have the extension JAR or ZIP. Each JAR file must have a name in the database. Usually, you use the same name as the JAR file, without the extension. For example, if you install a JAR file named *myjar.zip*, you would generally give it a JAR name of *myjar*.

☞ For more information, see "INSTALL JAVA statement" [*ASA SQL Reference,* page 532].

❖ To install a JAR (Sybase Central)

1. Connect to a database with DBA authority.

2. Open the Java Objects folder for the database.

3. Right click in the right pane and choose New ➤ Jar File from the popup menu.

4. Follow the instructions in the wizard.

❖ To install a JAR (SQL)

1. Connect to a database with DBA authority.

2. Enter the following statement:

```
INSTALL JAVA NEW
JAR 'jarname'
FROM FILE 'path\\JarName.jar'
```

Updating classes and Jars

You can update classes and JAR files using Sybase Central or by entering an INSTALL JAVA statement in Interactive SQL or some other client application.

To update a class or JAR, you must have DBA authority and a newer version of the compiled class file or JAR file available in a file on disk.

When updated classes take effect

Only new connections established after installing the class, or which use the class for the first time after installing the class, use the new definition. Once the Virtual Machine loads a class definition, it stays in memory until the connection closes.

If you have been using a Java class or objects based on a class in the current connection, you need to disconnect and reconnect to use the new class definition.

☞ To understand why the updated classes take effect in this manner, you need to know a little about how the VM works. For information, see "Configuring memory for Java" on page 99.

❖ **To update a class or JAR (Sybase Central)**

1. Connect to a database with DBA authority.

2. Open the Java Objects folder.

3. Locate the class or JAR file you wish to update.

4. Right-click the class or JAR file and choose Update from the popup menu.

5. In the resulting dialog, specify the name and location of the class or JAR file to be updated. You can click Browse to search for it.

Tip
You can also update a Java class or JAR file by clicking Update Now on the General tab of its property sheet.

❖ **To update a class or JAR (SQL)**

1. Connect to a database with DBA authority.

2. Execute the following statement:

```
INSTALL JAVA UPDATE
[ JAR 'jarname' ]
FROM FILE 'filename'
```

If you are updating a JAR, you must enter the name by which the JAR is known in the database.

☞ For more information, see "INSTALL JAVA statement" [*ASA SQL Reference,* page 532].

Special features of Java classes in the database

This section describes features of Java classes when used in the database.

Supported classes

You cannot use all classes from the JDK. The runtime Java classes available for use in the database server belong to a subset of the Java API.

☞ For more information about supported packages, see "Supported Java packages" on page 101.

Calling the main method

You typically start Java applications (outside the database) by running the Java VM on a class that has a **main** method.

For example, the **JDBCExamples** class in the file *Samples\ASA\Java\JDBCExamples.java* under your SQL Anywhere directory has a main method. When you execute the class from the command line using a command such as the following, it is the main method that executes:

```
java JDBCExamples
```

☞ For more information about how to run the **JDBCExamples** class, see "Establishing JDBC connections" on page 117.

❖ **To call the main method of a class from SQL**

1. Declare the method with an array of strings as an argument:

```
public static void main( java.lang.String[] args ){
...
}
```

2. Create a stored procedure that wraps this method.

☞ For more information, see "CREATE PROCEDURE statement" [*ASA SQL Reference,* page 373].

3. Invoke the main method using the CALL statement.

Each member of the array of strings must be of CHAR or VARCHAR data type, or a literal string.

Using threads in Java applications

With features of the **java.lang.Thread** package, you can use multiple threads

in a Java application. Each Java thread is an engine thread, and comes from the number of threads permitted by the -gn database server option.

You can synchronize, suspend, resume, interrupt, or stop threads in Java applications.

☞ For more information about database server threads, see "-gn server option" [*ASA Database Administration Guide,* page 145].

Serialization of JDBC calls

All calls to the server-side JDBC driver are serialized, such that only one thread is actively executing JDBC at any one time.

Procedure Not Found error

If you supply an incorrect number of arguments when calling a Java method, or if you use an incorrect data type, the server responds with a `Procedure Not Found` error. You should check the number and type of arguments.

Returning result sets from Java methods

This section describes how to make result sets available from Java methods. You must write a Java method that returns a result set to the calling environment, and wrap this method in a SQL stored procedure declared to be EXTERNAL NAME of LANGUAGE JAVA.

❖ **To return result sets from a Java method**

1. Ensure that the Java method is declared as public and static in a public class.

2. For each result set you expect the method to return, ensure that the method has a parameter of type **java.sql.ResultSet[]**. These result set parameters must all occur at the end of the parameter list.

3. In the method, first create an instance of **java.sql.ResultSet** and then assign it to one of the **ResultSet[]** parameters.

4. Create a SQL stored procedure of type EXTERNAL NAME LANGUAGE JAVA. This type of procedure is a wrapper around a Java method. You can use a cursor on the SQL procedure result set in the same way as any other procedure that returns result sets.

☞ For more information about the syntax for stored procedures that are wrappers for Java methods, see "CREATE PROCEDURE statement" [*ASA SQL Reference,* page 373].

Example

The following simple class has a single method which executes a query and passes the result set back to the calling environment.

```
import java.sql.*;

public class MyResultSet {
  public static void return_rset( ResultSet[] rset1 )
      throws SQLException {
    Connection conn = DriverManager.getConnection(
                      "jdbc:default:connection" );
    Statement stmt = conn.createStatement();
    ResultSet rset =
      stmt.executeQuery (
                      "SELECT CAST( JName.lastName " +
                      "AS CHAR( 50 ) )" +
                      "FROM jdba.contact " );
    rset1[0] = rset;
  }
}
```

You can expose the result set using a CREATE PROCEDURE statement that indicates the number of result sets returned from the procedure and the **signature** of the Java method.

A CREATE PROCEDURE statement indicating a result set could be defined as follows:

```
CREATE PROCEDURE result_set()
  DYNAMIC RESULT SETS 1
  EXTERNAL NAME
    'MyResultSet.return_rset ([Ljava/sql/ResultSet;)V'
  LANGUAGE JAVA
```

You can open a cursor on this procedure, just as you can with any ASA procedure returning result sets.

The string **(Ljava/sql/ResultSet;)V** is a Java method signature which is a compact character representation of the number and type of the parameters and return value.

☞ For more information about Java method signatures, see "CREATE PROCEDURE statement" [*ASA SQL Reference,* page 373].

Returning values from Java via stored procedures

You can use stored procedures created using the EXTERNAL NAME LANGUAGE JAVA as wrappers around Java methods. This section describes how to write your Java method to exploit OUT or INOUT parameters in the stored procedure.

Java does not have explicit support for INOUT or OUT parameters. Instead, you can use an array of the parameter. For example, to use an integer OUT parameter, create an array of exactly one integer:

```
public class TestClass {
  public static boolean testOut( int[] param ){
    param[0] = 123;
    return true;
  }
}
```

The following procedure uses the **testOut** method:

```
CREATE PROCEDURE sp_testOut ( OUT p INTEGER )
EXTERNAL NAME 'TestClass/testOut ([I)Z'
LANGUAGE JAVA
```

The string **([I)Z** is a Java method signature, indicating that the method has a single parameter, which is an array of integers, and returns a Boolean value. You must define the method so that the method parameter you wish to use as an OUT or INOUT parameter is an array of a Java data type that corresponds to the SQL data type of the OUT or INOUT parameter.

☞ For more information about the syntax, including the method signature, see "CREATE PROCEDURE statement" [*ASA SQL Reference,* page 373].

Security management for Java

Java provides security managers than you can use to control user access to security-sensitive features of your applications, such as file access and network access. Adaptive Server Anywhere provides the following support for Java security managers in the database:

♦ Adaptive Server Anywhere provides a default security manager.

♦ You can provide your own security manager.

   ☞ For information, see "Implementing your own security manager" on page 97.

The default security manager   The default security manager is the class **com.sybase.asa.jrt.SAGenericSecurityManager**. It carries out the following tasks:

1. It checks the value of the database option JAVA_INPUT_OUTPUT.

2. It checks whether the database server was started in C2 security mode using the -sc database server option.

3. If the connection property is OFF, it disallows access to Java file I/O features.

4. If the database server is running in C2 security mode, it disallows access to **java.net** packages.

5. When the security manager prevents a user from accessing a feature, it returns a **java.lang.SecurityException**.

☞ For more information, see "JAVA_INPUT_OUTPUT option [database]" [*ASA Database Administration Guide,* page 664], and "-sc server option" [*ASA Database Administration Guide,* page 157].

Controlling Java file I/O using the default security manager

Java file I/O is controlled through the JAVA_INPUT_OUTPUT database option. By default this option is set to OFF, disallowing file I/O.

❖ **To permit file access using the default security manager**

1. Set the JAVA_INPUT_OUTPUT option to ON:

```
SET OPTION JAVA_INPUT_OUTPUT='ON'
```

Implementing your own security manager

There are several steps to implementing your own security manager.

❖ **To provide your own security manager**

1. Implement a class that extends **java.lang.SecurityManager**.

   The SecurityManager class has a number of methods to check whether a particular action is allowed. If the action is permitted, the method returns silently. If the method returns a value a **SecurityException** is thrown.

   You must override methods that govern actions you wish to permit with methods that return silently. You can do this by implementing a `public void` method.

2. Assign appropriate users to your security manager.

   You use the add_user_security_manager, update_user_security_manager, and delete_user_security_manager system stored procedures to assign security managers to a user. For example, to assign the MySecurityManager class as the security manager for a user, you would execute the following command:

```
call dbo.add_user_security_manager(
   user_name, 'MySecurityManager', NULL )
```

Example

The following class allows reading from files but disallows writing:

```
public class MySecurityManager extends SecurityManager
{ public void checkRead(FileDescriptor) {}
  public void checkRead(String) {}
  public void checkRead(String, Object) {}
}
```

The **SecurityManager.checkWrite** methods are not overridden, and prevent write operations on files. The **checkRead** methods return silently, permitting the action.

Configuring memory for Java

This section describes the memory requirements for running Java in the database and how to set up your server to meet those requirements.

The Java VM requires a significant amount of cache memory.

☞ For information on tuning the cache, see "Using the cache to improve performance" [*ASA SQL User's Guide,* page 180].

Database and connection-level requirements

The Java VM uses memory on both a per-database and on a per-connection basis.

♦ The per-database requirements are not **relocatable**: they cannot be paged out to disk. They must fit into the server cache. This type of memory is not for the server; it is for each database. When estimating cache requirements, you must sum the requirements for each database you run on the server.

♦ The per-connection requirements are relocatable, but only as a unit. The requirements for one connection are either all in cache, or all in the temporary file.

How memory is used

Java in the database requires memory for several purposes:

♦ When Java is first used when a server is running, the VM is loaded into memory, requiring over 1.5 Mb of memory. This is part of the database-wide requirements. An additional VM is loaded for each database that uses Java.

♦ For each connection that uses Java, a new instance of the VM loads for that connection. The new instance requires about 200K per connection.

♦ Each class definition that is used in a Java application is loaded into memory. This is held in database-wide memory: separate copies are not required for individual connections.

♦ Each connection requires a working set of Java variables and application stack space (used for method arguments and so on).

Managing memory

You can control memory use in the following ways:

♦ **Set the overall cache size**   You must use a cache size sufficient to meet all the requirements for non-relocatable memory.

The cache size is set when the server is started using the -c option.

In many cases, a cache size of 8 Mb is sufficient.

◆ **Set the namespace size**   The Java namespace size is the maximum size, in bytes, of the per database memory allocation.

You can set this using the JAVA_NAMESPACE_SIZE option. The option is global, and can only be set by a user with DBA authority.

◆ **Set the heap size**   This JAVA_HEAP_SIZE option sets the maximum size, in bytes, of per-connection memory.

This option can be set for individual connections, but as it affects the memory available for other users it can be set only by a user with DBA authority.

Starting and stopping the VM

In addition to setting memory parameters for Java, you can unload the VM when Java is not in use using the STOP JAVA statement. Only a user with DBA authority can execute this statement. The syntax is simply:

```
STOP JAVA
```

The VM loads whenever a Java operation is carried out. If you wish to explicitly load it in readiness for carrying out Java operations, you can do so by executing the following statement:

```
START JAVA
```

Java classes reference

This section provides reference material on JDK classes and packages supported within Adaptive Server Anywhere. User-defined classes and packages are must be installed into the database by a user with DBA authority before they can be used.

Supported Java packages

This section lists the packages of built-in classes available for use in a Java-enabled database. For information about any classes within the package that may be unsupported or partially supported, see "Unsupported Java packages and classes" on page 102, and "Partially supported packages and classes" on page 102.

Packages not listed here must be installed into your database before you can use them.

♦ java.beans

♦ java.io. The classes that govern file access are supported only on certain Windows operating systems, and only if the JAVA_INPUT_OUTPUT option is set to ON. See "JAVA_INPUT_OUTPUT option [database]" [*ASA Database Administration Guide,* page 664].

♦ java.lang

♦ java.lang.reflect

♦ java.lang.Thread

♦ java.math

♦ java.net

♦ java.net.PlainDatagramSocketImpl

♦ java.rmi

♦ java.rmi.dgc

♦ java.rmi.registry

♦ java.rmi.server

♦ java.security

♦ java.security.acl

♦ java.security.interfaces

- java.SQL. For details on support for JDBC 2.0 features, see "JDBC in the database features" on page 106.

- java.text

- java.util

- java.util.zip

Unsupported Java packages and classes

Classes in the following packages are not supported in Adaptive Server Anywhere:

- java.applet

- java.awt

- java.awt.datatransfer

- java.awt.event

- java.awt.image

- All packages prefixed by sun. For example, sun.audio.

Partially supported packages and classes

The following classes are *partially supported* . They have some unsupported native methods:

- java.lang.ClassLoader

- java.lang.Compiler

- java.lang.Runtime (exec/load/loadlibrary)

- java.io.File

- java.io.FileDescriptor

- java.io.FileInputStream

- java.io.FileOutputStream

- java.io.RandomAccessFile

- java.util.zip.Deflater

- java.util.zip.Inflater

JDBC Programming

About this chapter

This chapter describes how to use JDBC to access data.

JDBC can be used both from client applications and inside the database. Java classes using JDBC provide a more powerful alternative to SQL stored procedures for incorporating programming logic in the database.

Contents

JDBC overview

JDBC provides a SQL interface for Java applications: if you want to access relational data from Java, you do so using JDBC calls.

Rather than a thorough guide to the JDBC database interface, this chapter provides some simple examples to introduce JDBC and illustrates how you can use it on the client and in the database.

☞ The examples illustrate the distinctive features of using JDBC in Adaptive Server Anywhere. For more information about JDBC programming, see any JDBC programming book.

JDBC and Adaptive Server Anywhere

You can use JDBC with Adaptive Server Anywhere in the following ways:

♦ **JDBC on the client**   Java client applications can make JDBC calls to Adaptive Server Anywhere. The connection takes place through a JDBC driver. SQL Anywhere Studio includes two JDBC drivers: the jConnect driver for pure Java applications and the iAnywhere JDBC driver, which is a type 2 JDBC driver.

In this chapter, the phrase **client application** applies both to applications running on a user's machine and to logic running on a middle-tier application server.

♦ **JDBC in the database**   Java classes installed into a database can make JDBC calls to access and modify data in the database using an internal JDBC driver.

JDBC resources

♦ **Required software**   You need TCP/IP to use the Sybase jConnect driver.

The Sybase jConnect driver may already be available, depending on your installation of Adaptive Server Anywhere.

For more information about the jConnect driver and its location, see "The jConnect driver files" on page 110.

♦ **Example source code**   You can find source code for the examples in this chapter in the file *Samples\ASA\Java\JDBCExamples.java* in your SQL Anywhere directory.

☞ For more information about how to set up the Java examples, including the **JDBCExamples** class, see "Setting up the Java sample" on page 82.

Choosing a JDBC driver

Two JDBC drivers are provided for Adaptive Server Anywhere:

♦ **jConnect**   This driver is a 100% pure Java driver. It communicates with Adaptive Server Anywhere using the TDS client/server protocol.

☞ For jConnect documentation, see *http://sybooks.sybase.com/jc.html*.

♦ **iAnywhere JDBC driver**   This driver communicates with Adaptive Server Anywhere using the Command Sequence client/server protocol. Its behavior is consistent with ODBC, embedded SQL, and OLE DB applications.

When choosing which driver to use, you may want to consider the following factors:

♦ **Features**   Both drivers are JDK 2 compliant. The iAnywhere JDBC driver provides fully-scrollable cursors, which are not available in jConnect.

♦ **Pure Java**   The jConnect driver is a pure Java solution. The iAnywhere JDBC driver requires the Adaptive Server Anywhere ODBC driver and is not a pure Java solution.

♦ **Performance**   The iAnywhere JDBC driver provides better performance for most purposes than the jConnect driver.

♦ **Compatibility**   The TDS protocol used by the jConnect driver is shared with Adaptive Server Enterprise. Some aspects of the driver's behavior are governed by this protocol, and are configured to be compatible with Adaptive Server Enterprise.

Both drivers are available on Windows 95/98/Me and Windows NT/2000/XP, as well as supported UNIX and Linux operating systems. They are not available on NetWare or Windows CE.

JDBC program structure

The following sequence of events typically occur in JDBC applications:

1. **Create a Connection object**   Calling a **getConnection** class method of the **DriverManager** class creates a **Connection** object, and establishes a connection with a database.

2. **Generate a Statement object**   The **Connection** object generates a **Statement** object.

3. **Pass a SQL statement**   A SQL statement that executed within the database environment passes to the **Statement** object. If the statement is a query, this action returns a **ResultSet** object.

   The **ResultSet** object contains the data returned from the SQL statement, but exposes it one row at a time (similar to the way a cursor works).

4. **Loop over the rows of the result set**   The **next** method of the **ResultSet** object performs two actions:

   ♦ The current row (the row in the result set exposed through the **ResultSet** object) advances one row.

   ♦ A Boolean value (true/false) returns to indicate whether there is, in fact, a row to advance to.

5. **For each row, retrieve the values**   Values are retrieved for each column in the **ResultSet** object by identifying either the name or position of the column. You can use the **getDate** method to get the value from a column on the current row.

Java objects can use JDBC objects to interact with a database and get data for their own use, for example to manipulate or for use in other queries.

JDBC in the database features

The version of JDBC that you can use from Java in the database is determined by the JDK version that the database is set up to use.

♦ If your database is initialized with JDK 1.2 or JDK 1.3, you can use the JDBC 2.0 API.

☞ For information on upgrading databases to JDK 1.2 or JDK 1.3, see "ALTER DATABASE statement" [*ASA SQL Reference,* page 266] or "Upgrading a database using the dbupgrad command-line utility" [*ASA Database Administration Guide,* page 600].

♦ If your database is initialized with JDK 1.1, you can use JDBC 1.2 features. The internal JDBC driver for JDK 1.1 (asajdbc) makes some features of JDBC 2.0 available from server-side Java applications, but does not provide full JDBC 2.0 support.

☞ For more information, see "Using JDBC 2.0 features from JDK 1.1 databases" on page 106.

Using JDBC 2.0 features from JDK 1.1 databases

This section describes how to access JDBC 2.0 features from databases initialized with JDK 1.1 support. For many purposes, a better solution is to upgrade your version of Java in the database to 1.3.

For databases initialized with JDK 1.1 support, the **sybase.sql.ASA** package contains features that are part of JDBC 2.0. To use these JDBC 2.0 features you must cast your JDBC objects into the corresponding classes in the **sybase.sql.ASA** package, rather than the **java.sql** package. Classes that are declared as **java.sql** are restricted to JDBC 1.2 functionality only.

The classes in **sybase.sql.ASA** are as follows:

JDBC class	Sybase internal driver class
java.sql.Connection	sybase.sql.ASA.SAConnection
java.sql.Statement	sybase.sql.ASA.SAStatement
java.sql.PreparedStatement	sybase.sql.ASA.SAPreparedStatement
java.sql.CallableStatement	sybase.sql.ASA.SACallableStatement
java.sql.ResultSetMetaData	sybase.sql.ASA.SAResultSetMetaData
java.sql.ResultSet	sybase.sql.SAResultSet
java.sql.DatabaseMetaData	sybase.sql.SADatabaseMetaData

The following function provides a **ResultSetMetaData** object for a prepared statement without requiring a **ResultSet** or executing the statement. This function is not part of the JDBC 1.2 standard.

```
ResultSetMetaData sybase.sql.ASA.SAPreparedStatement.describe()
```

The following code fetches the previous row in a result set, a feature not supported in JDBC 1.2:

```
import java.sql.*;
import sybase.sql.asa.*;
ResultSet rs;
// more code here
( ( sybase.sql.asa.SAResultSet)rs ).previous();
```

JDBC 2.0 restrictions

The following classes are part of the JDBC 2.0 core interface, but are not available in the **sybase.sql.ASA** package:

♦ java.sql.Blob

♦ java.sql.Clob

♦ java.sql.Ref

♦ java.sql.Struct

♦ java.sql.Array

♦ java.sql.Map

The following JDBC 2.0 core functions are not available in the **sybase.sql.ASA** package:

Class in sybase.sql.-ASA	Missing functions
SAConnection	java.util.Map getTypeMap()
	void setTypeMap(java.util.Map map)
SAPreparedStatement	void setRef(int pidx, java.sql.Ref r)
	void setBlob(int pidx, java.sql.Blob b)
	void setClob(int pidx, java.sql.Clob c)
	void setArray(int pidx, java.sql.Array a)
SACallableStatement	Object getObject(pidx, java.util.Map map)
	java.sql.Ref getRef(int pidx)
	java.sql.Blob getBlob(int pidx)
	java.sql.Clob getClob(int pidx)
	java.sql.Array getArray(int pidx)
SAResultSet	Object getObject(int cidx, java.util.Map map)
	java.sql.Ref getRef(int cidx)
	java.sql.Blob getBlob(int cidx)
	java.sql.Clob getClob(int cidx)
	java.sql.Array getArray(int cidx)
	Object getObject(String cName, java.util.Map map)
	java.sql.Ref getRef(String cName)
	java.sql.Blob getBlob(String cName)
	java.sql.Clob getClob(String cName)
	java.sql.Array getArray(String cName)

Differences between client- and server-side JDBC connections

A difference between JDBC on the client and in the database server lies in establishing a connection with the database environment.

♦ **Client side**   In client-side JDBC, establishing a connection requires the Sybase jConnect JDBC driver or the iAnywhere JDBC driver. Passing arguments to the **DriverManager.getConnection** establishes the connection. The database environment is an external application from the perspective of the client application.

♦ **Server-side**   When using JDBC within the database server, a connection already exists. A value of **jdbc:default:connection** passes to

DriverManager.getConnection, which provides the JDBC application with the ability to work within the current user connection. This is a quick, efficient, and safe operation because the client application has already passed the database security to establish the connection. The user ID and password, having been provided once, do not need to be provided again. The internal JDBC driver can only connect to the database of the current connection.

You can write JDBC classes in such a way that they can run both at the client and at the server by employing a single conditional statement for constructing the URL. An external connection requires the machine name and port number, while the internal connection requires **jdbc:default:connection**.

Using the jConnect JDBC driver

If you wish to use JDBC from a client application or applet, you must have the jConnect JDBC driver to connect to Adaptive Server Anywhere databases.

jConnect is included with SQL Anywhere Studio. If you received Adaptive Server Anywhere as part of another package, jConnect may or may not be included. You must have jConnect in order to use JDBC from client applications. You can use JDBC in the database without jConnect.

☞ For jConnect documentation, see *http://sybooks.sybase.com/jc.html*.

The jConnect driver files

The jConnect JDBC driver is installed into a set of directories under the *Sybase\Shared* directory. Two versions of jConnect are supplied:

♦ **jConnect 4.5**   This version of jConnect is for use when developing JDK 1.1 applications. jConnect 4.5 is installed into the *Sybase\Shared\jConnect-4_5* directory.

jConnect 4.5 is supplied as a set of classes.

♦ **jConnect 5.5**   This version of jConnect is for use when developing JDK 1.2 or later applications. jConnect 5.5 is installed into the *Sybase\Shared\jConnect-5_5* directory.

jConnect 5.5 is supplied as a jar file named *jconn2.jar*.

Examples in this chapter use jConnect 5.5. Users of jConnect 4.5 must make appropriate substitutions.

Setting the CLASSPATH for jConnect
For your application to use jConnect, the jConnect classes must be in your classpath at compile time and run time, so the Java compiler and Java runtime can locate the necessary files.

The following command adds the jConnect 5.5 driver to an existing CLASSPATH environment variable where *path* is your *Sybase\Shared* directory.

```
set classpath=%classpath%;path\jConnect-5_5\classes\jconn2.jar
```

The following command adds the jConnect 4.5 driver to an existing CLASSPATH environment variable:

```
set classpath=%classpath%;path\jConnect-4_5\classes
```

Importing the jConnect classes
The classes in jConnect are all in the **com.sybase** package.

If you are using jConnect 5.5, your application must access classes in
com.sybase.jdbc2.jdbc. You must import these classes at the beginning of
each source file:

```
import com.sybase.jdbc2.jdbc.*
```

If you are using jConnect 4.5, the classes are in **com.sybase.jdbc**. You must
import these classes at the beginning of each source file:

```
import com.sybase. jdbc.*
```

Installing jConnect system objects into a database

If you wish to use jConnect to access system table information (database
metadata), you must add the jConnect system objects to your database.

By default, the jConnect system objects are added to any new database. You
can choose to add the jConnect objects to the database when creating, when
upgrading, or at a later time.

You can install the jConnect system objects from Sybase Central or from
Interactive SQL.

❖ **To add jConnect system objects to a database (Sybase Central)**

1. Connect to the database from Sybase Central as a user with DBA
   authority.

2. In the left pane, right-click the database and choose Upgrade database
   from the popup menu.

   The Upgrade a Database Wizard appears.

3. Follow the instructions in the wizard to add jConnect support to the
   database.

❖ **To add jConnect system objects to a database (Interactive SQL)**

1. Connect to the database from Interactive SQL as a user with DBA
   authority, and enter the following command in the SQL Statements pane:

   ```
   read path\scripts\jcatalog.sql
   ```

where *path* is your SQL Anywhere directory.

> **Tip**
>
> You can also use a command prompt to add the jConnect system objects to a database. At the command prompt, type:
>
> ```
> dbisql -c "uid=user;pwd=pwd" path\scripts\jcatalog.sql
> ```
>
> where *user* and *pwd* identify a user with DBA authority, and *path* is your SQL Anywhere directory.

Loading the jConnect driver

Before you can use jConnect in your application, load the driver by entering the following statement:

```
Class.forName("com.sybase.jdbc2.jdbc.SybDriver").newInstance();
```

Using the **newInstance** method works around issues in some browsers.

Supplying a URL for the server

To connect to a database via jConnect, you need to supply a Uniform Resource Locator (URL) for the database. An example given in the section "Connecting from a JDBC client application using jConnect" on page 117 is as follows:

```
StringBuffer temp = new StringBuffer();
// Use the jConnect driver...
temp.append("jdbc:sybase:Tds:");
// to connect to the supplied machine name...
temp.append(_coninfo);
// on the default port number for ASA...
temp.append(":2638");
// and connect.
System.out.println(temp.toString());
conn = DriverManager.getConnection(temp.toString() , _props );
```

The URL is put together in the following way:

```
jdbc:sybase:Tds:machine-name:port-number
```

The individual components are:

- ♦ **jdbc:sybase:Tds** The Sybase jConnect JDBC driver, using the TDS application protocol.

- ♦ **machine-name** The IP address or name of the machine on which the server is running. If you are establishing a same-machine connection, you can use **localhost**, which means the current machine

- ♦ **port number** The port number on which the database server listens.

The port number assigned to Adaptive Server Anywhere is 2638. Use that number unless there are specific reasons not to do so.

The connection string must be less than 253 characters in length.

Specifying a database on a server

Each Adaptive Server Anywhere server may have one or more databases loaded at a time. The URL in the previous section specifies a server, but does not specify a database. The connection attempt is made to the default database on the server.

You can specify a particular database by providing an extended form of the URL in one of the following ways.

Using the ServiceName parameter

```
jdbc:sybase:Tds:machine-name:port-number?ServiceName=DBN
```

The question mark followed by a series of assignments is a standard way of providing arguments to a URL. The case of **servicename** is not significant, and there must be no spaces around the = sign. The *DBN* parameter is the database name.

Using the RemotePWD parameter

A more general method allows you to provide additional connection parameters such as the database name, or a database file, using the **RemotePWD** field. You set **RemotePWD** as a Properties field using the **setRemotePassword()** method.

Here is sample code that illustrates how to use the field.

```
sybDrvr = (SybDriver)Class.forName(
    "com.sybase.jdbc2.jdbc.SybDriver" ).newInstance();
props = new Properties();
props.put( "User", "DBA" );
props.put( "Password", "SQL" );
sybDrvr.setRemotePassword(
    null, "dbf=asademo.db", props );
Connection con = DriverManager.getConnection(
    "jdbc:sybase:Tds:localhost", props );
```

Using the database file parameter **DBF**, you can start a database on a server using jConnect. By default, the database is started with autostop=YES. If you specify a DBF or DBN of **utility_db**, then the utility database will automatically be started.

☞ For more information on the utility database, see "Using the utility database" [*ASA Database Administration Guide,* page 296].

☞ For complete jConnect documentation, see *http://sybooks.sybase.com/jc.html*.

Database options set for jConnect connections

When an application connects to the database using the jConnect driver, two stored procedures are called:

1. sp_tsql_environment sets some database options for compatibility with Adaptive Server Enterprise behavior.

2. The sp_mda procedure is then called, and sets some other options. In particular, the sp_mda procedure determines the QUOTED_IDENTIFIER setting. To change the default behavior, you should modify the sp_mda procedure.

Using the iAnywhere JDBC driver

The iAnywhere JDBC driver provides a JDBC driver that has some performance benefits and feature benefits compared to the pure Java jConnect JDBC driver, but which is not a pure-Java solution.

☞ For information on choosing which JDBC driver to use, see "Choosing a JDBC driver" on page 104.

Required files

☞ The Java component of the iAnywhere JDBC driver is included in the *jodbc.jar* file installed into the *Java* subdirectory of your SQL Anywhere installation. For Windows, the native component is *dbjodbc9.dll* in the *win32* subdirectory of your SQL Anywhere installation; for UNIX and Linux, the native component is *dbjodbc9.so.* This component must be in the system path. When deploying applications using this driver, you must also deploy the ODBC driver files.

Establishing a connection

The following code illustrates how to establish a connection using the iAnywhere JDBC driver:

```
String      driver, url;
Connection  conn;
driver="ianywhere.ml.jdbcodbc.IDriver";
url = "jdbc:odbc:dsn=ASA 9.0 Sample";
Class.forName( driver );
conn = DriverManager.getConnection( url );
```

There are several things to note about this code:

♦ As the classes are loaded using `Class.forName`, the package containing the iAnywhere JDBC driver does not have to be imported using `import` statements.

♦ *jodbc.jar* must be in your classpath when you run the application.

♦ The URL contains **jdbc:odbc:** followed by a standard ODBC connection string. The connection string is commonly an ODBC data source, but you can also use explicit semicolon separated individual connection parameters in addition to or instead of the data source. For more information on the parameters that you can use in a connection string, see "Connection parameters" [*ASA Database Administration Guide,* page 176].

If you do not use a data source, you should specify the ODBC driver to use by including the driver parameter in your connection string:

```
url = "jdbc:odbc:";
url += "driver=Adaptive Server Anywhere 9.0;...";
```

Character sets

On UNIX the iAnywhere JDBC driver does *not* use ODBC Unicode bindings or calls and does not carry out character translations. Sending

non-ASCII data through the iAnywhere JDBC driver leads to data corruption.

On Windows the iAnywhere JDBC driver *does* use ODBC Unicode bindings and calls to translate among character sets.

Establishing JDBC connections

This section presents classes that establish a JDBC database connection from a Java application. The examples in this section use jConnect (client side) or Java in the database (server side). For information on establishing connections using the iAnywhere JDBC driver, see "Using the iAnywhere JDBC driver" on page 115.

Connecting from a JDBC client application using jConnect

If you wish to access database system tables (database metadata) from a JDBC application, you must add a set of jConnect system objects to your database. The internal JDBC driver classes and jConnect share stored procedures for database metadata support. These procedures are installed to all databases by default. The *dbinit* `-i` option prevents this installation.

☞ For more information about adding the jConnect system objects to a database, see "Using the jConnect JDBC driver" on page 110.

The following complete Java application is a command-line application that connects to a running database, prints a set of information to your command-line, and terminates.

Establishing a connection is the first step any JDBC application must take when working with database data.

☞ This example illustrates an external connection, which is a regular client/server connection. For information on how to create an internal connection from Java classes running inside the database server, see "Establishing a connection from a server-side JDBC class" on page 120.

External connection example code

The following is the source code for the methods used to make a connection. The source code can be found in the **main** method and the **ASAConnect** method of the file *JDBCExamples.java* in the *Samples\ASA\Java* directory under your SQL Anywhere directory:

```
import java.sql.*;              // JDBC
import com.sybase.jdbc2.jdbc.*;     // Sybase jConnect
import java.util.Properties;   // Properties
import sybase.sql.*;             // Sybase utilities
import asademo.*;                // Example classes
public class JDBCExamples{
  private static Connection conn;
```

```java
public static void main( String args[] ){
    // Establish a connection
    conn = null;
    String machineName =
        ( args.length == 1 ? args[0] : "localhost" );
    ASAConnect( "DBA", "SQL", machineName );
    if( conn!=null ) {
        System.out.println( "Connection successful" );
    }else{
        System.out.println( "Connection failed" );
    }

    try{
        getObjectColumn();
        getObjectColumnCastClass();
        insertObject();
    }
    catch( Exception e ){
        System.out.println( "Error: " + e.getMessage() );
        e.printStackTrace();
    }
  }
private static void ASAConnect( String userID,
                String password,
                String machineName ) {
    // Connect to an Adaptive Server Anywhere
    String coninfo = new String( machineName );

    Properties props = new Properties();
    props.put( "user", userID );
    props.put( "password", password );
    props.put("DYNAMIC_PREPARE", "true");

// Load jConnect
    try {
        Class.forName(
            "com.sybase.jdbc2.jdbc.SybDriver" ).newInstance();
        String dbURL = "jdbc:sybase:Tds:" + machineName   +
                        ":2638/?JCONNECT_VERSION=5";
        System.out.println( dbURL );
        conn = DriverManager.getConnection( dbURL , props );
    }
    catch ( Exception e ) {
        System.out.println( "Error: " + e.getMessage() );
        e.printStackTrace();
    }
  }
}
```

How the external connection example works

The external connection example is a Java command-line application.

Importing packages

The application requires several libraries, which are imported in the first lines of *JDBCExamples.java*:

♦ The **java.sql** package contains the Sun Microsystems JDBC classes, which are required for all JDBC applications. You'll find it in the *classes.zip* file in your Java subdirectory.

♦ Imported from **com.sybase.jdbc2.jdbc**, the Sybase jConnect JDBC driver is required for all applications that connect using jConnect.

♦ The application uses a **property list**. The **java.util.Properties** class is required to handle property lists. You'll find it in the *classes.zip* file in your *Java* subdirectory.

♦ The **asademo** package contains classes used in some samples. You'll find it in the *Samples\ASA\Java\asademo.jar* file.

The main method

Each Java application requires a class with a method named **main**, which is the method invoked when the program starts. In this simple example, **JDBCExamples.main** is the only public method in the application.

The **JDBCExamples.main** method carries out the following tasks:

1. Processes the command-line argument, using the machine name if supplied. By default, the machine name is *localhost*, which is appropriate for the personal database server.

2. Calls the **ASAConnect** method to establish a connection.

3. Executes several methods that scroll data to your command-line.

The ASAConnect method

The **JDBCExamples.ASAConnect** method carries out the following tasks:

1. Connects to the default running database using Sybase jConnect.
   ♦ **Class.forName** loads jConnect. Using the **newInstance** method, it works around issues in some browsers.
   ♦ The **StringBuffer** statements build up a connection string from the literal strings and the supplied machine name provided on the command-line.
   ♦ **DriverManager.getConnection** establishes a connection using the connection string.

2. Returns control to the calling method.

Running the external connection example

This section describes how to run the external connection example.

❖ **To create and execute the external connection example applica-tion**

1. Open the command prompt.

2. Change to your SQL Anywhere directory.

3. Change to the *Samples\ASA\Java* subdirectory.

4. Ensure the database is loaded onto a database server running TCP/IP. You can start such a server on your local machine using the following command (from the *Samples\ASA\Java* subdirectory):

   ```
   start dbeng9 ..\..\..\asademo
   ```

5. Enter the following at the command prompt to run the example:

   ```
   java JDBCExamples
   ```

   If you wish to try this against a server running on another machine, you must enter the correct name of that machine. The default is **localhost,** which is an alias for the current machine name.

6. Confirm that a list of people and products appears at the command prompt.

   If the attempt to connect fails, an error message appears instead. Confirm that you have executed all the steps as required. Check that your CLASSPATH is correct. An incorrect CLASSPATH results in a failure to locate a class.

☞ For more information about using jConnect, see , and see the online documentation for jConnect.

Establishing a connection from a server-side JDBC class

SQL statements in JDBC are built using the **createStatement** method of a **Connection** object. Even classes running inside the server need to establish a connection to create a **Connection** object.

Establishing a connection from a server-side JDBC class is more straightforward than establishing an external connection. Because a user already connected executes the server-side class, the class simply uses the current connection.

Server-side connection example code

The following is the source code for the example. You can find the source code in the **InternalConnect** method of

Samples\ASA\Java\JDBCExamples.java under your SQL Anywhere
directory:

```
public static void InternalConnect() {
  try {
      conn =
        DriverManager.getConnection("jdbc:default:connection");
      System.out.println("Hello World");
    }
    catch ( Exception e ) {
      System.out.println("Error: " + e.getMessage());
      e.printStackTrace();
    }
  }
}
```

How the server-side connection example works

In this simple example, **InternalConnect()** is the only method used in the
application.

The application requires only one of the libraries (JDBC) imported in the
first line of the *JDBCExamples.java* class. The others are for external
connections. The package named **java.sql** contains the JDBC classes.

The **InternalConnect()** method carries out the following tasks:

1. Connects to the default running database using the current connection:
   ◆ **DriverManager.getConnection** establishes a connection using a
     connection string of **jdbc:default:connection**.

2. Prints **Hello World** to the current standard output, which is the server
   window. **System.out.println** carries out the printing.

3. If there is an error in the attempt to connect, an error message appears in
   the server window, together with the place where the error occurred.

   The **try** and **catch** instructions provide the framework for the error
   handling.

4. Terminates the class.

Running the server-side connection example

This section describes how to run the server-side connection example.

❖ **To create and execute the internal connection example application**

1. If you have not already done so, compile the *JDBCExamples.java* file. If you are using the JDK, you can do the following in the *Samples\ASA\Java* directory from a command prompt:

   ```
   javac JDBCExamples.java
   ```

2. Start a database server using the sample database. You can start such a server on your local machine using the following command (from the *Samples\ASA\Java* subdirectory):

   ```
   start dbeng9 ..\..\..\asademo
   ```

   The TCP/IP network protocol is not necessary in this case since you are not using jConnect.

3. Install the class into the sample database. Once connected to the sample database, you can do this from Interactive SQL using the following command:

   ```
   INSTALL JAVA NEW
   FROM FILE 'path\Samples\ASA\Java\JDBCExamples.class'
   ```

   where *path* is the path to your installation directory.

   You can also install the class using Sybase Central. While connected to the sample database, open the Java Objects folder and choose File ➤ New ➤ Java Class. Then follow the instructions in the wizard.

4. You can now call the **InternalConnect** method of this class just as you would a stored procedure:

   ```
   CALL JDBCExamples>>InternalConnect()
   ```

   The first time a Java class is called in a session, the internal Java virtual machine must be loaded. This can take a few seconds.

5. Confirm that the message **Hello World** prints on the server screen.

Notes on JDBC connections

♦ **Autocommit behavior**   The JDBC specification requires that, by default, a COMMIT is performed after each data modification statement. Currently, the server-side JDBC behavior is to commit. You can control this behavior using a statement such as the following:

```
conn.setAutoCommit( false ) ;
```

where **conn** is the current connection object.

♦ **Connection defaults** From server-side JDBC, only the first call to **getConnection("jdbc:default:connection"**) creates a new connection with the default values. Subsequent calls return a wrapper of the current connection with all connection properties unchanged. If you set AutoCommit to OFF in your initial connection, any subsequent **getConnection** calls within the same Java code return a connection with AutoCommit set to OFF.

You may wish to ensure that closing a connection resets the connection properties to their default values, so that subsequent connections are obtained with standard JDBC values. The following type of code achieves this:

```
Connection conn = DriverManager.getConnection("");
boolean oldAutoCommit = conn.getAutoCommit();
try {
     // do code here
}
finally {
    conn.setAutoCommit( oldAutoCommit );
}
```

This discussion applies not only to AutoCommit, but also to other connection properties such as TransactionIsolation and isReadOnly.

Using JDBC to access data

Java applications that hold some or all classes in the database have significant advantages over traditional SQL stored procedures. At an introductory level, however, it may be helpful to use the parallels with SQL stored procedures to demonstrate the capabilities of JDBC. In the following examples, we write Java classes that insert a row into the Department table.

As with other interfaces, SQL statements in JDBC can be either **static** or **dynamic**. Static SQL statements are constructed in the Java application and sent to the database. The database server parses the statement, selects an execution plan, and executes the statement. Together, parsing and selecting an execution plan are referred to as **preparing** the statement.

If a similar statement has to be executed many times (many inserts into one table, for example), there can be significant overhead in static SQL because the preparation step has to be executed each time.

In contrast, a dynamic SQL statement contains placeholders. The statement, prepared once using these placeholders, can be executed many times without the additional expense of preparing.

In this section, we use static SQL. Dynamic SQL is discussed in a later section.

Preparing for the examples

This section describes how to prepare for the examples in the remainder of the chapter.

Sample code The code fragments in this section are taken from the complete class *Samples\ASA\Java\JDBCExamples.java*.

❖ To install the JDBCExamples class

1. If you have not already done so, install the *JDBCExamples.class* file into the sample database. Once connected to the sample database from Interactive SQL, enter the following command in the SQL Statements pane:

    ```
    INSTALL JAVA NEW
    FROM FILE 'path\Samples\ASA\Java\JDBCExamples.class'
    ```

    where *path* is the path to your installation directory.

    You can also install the class using Sybase Central. While connected to the sample database, open the Java Objects folder and choose File ➤ New ➤ Java Class. Then follow the instructions in the wizard.

Inserts, updates, and deletes using JDBC

The **Statement** object executes static SQL statements. You execute SQL statements such as INSERT, UPDATE, and DELETE, which do not return result sets, using the **executeUpdate** method of the **Statement** object. Statements, such as CREATE TABLE and other data definition statements, can also be executed using **executeUpdate**.

The following code fragment illustrates how JDBC carries out INSERT statements. It uses an internal connection held in the Connection object named **conn**. The code for inserting values from an external application using JDBC would need to use a different connection, but otherwise would be unchanged.

```
public static void InsertFixed()  {
    // returns current connection
    conn = DriverManager.getConnection(
        "jdbc:default:connection");
    // Disable autocommit
    conn.setAutoCommit( false );

    Statement stmt = conn.createStatement();

    Integer IRows = new Integer( stmt.executeUpdate
      ("INSERT INTO Department (dept_id, dept_name )"
      + "VALUES (201, 'Eastern Sales')"
        ) );
    // Print the number of rows updated
    System.out.println(IRows.toString() + " row inserted" );
  }
```

> **Source code available**
> This code fragment is part of the **InsertFixed** method of the **JDBCExamples** class included in the *Samples\ASA\Java* subdirectory of your installation directory.

Notes

♦ The **setAutoCommit** method turns off the AutoCommit behavior so changes are only committed if you execute an explicit COMMIT instruction.

♦ The **executeUpdate** method returns an integer which reflects the number of rows affected by the operation. In this case, a successful INSERT would return a value of one (1).

♦ The integer return type converts to an **Integer** object. The Integer class is a wrapper around the basic **int** data type, providing some useful methods such as **toString()**.

♦ The Integer **IRows** converts to a string to be printed. The output goes to the server window.

❖ **To run the JDBC Insert example**

1. Using Interactive SQL, connect to the sample database as user ID **DBA**.

2. Ensure the JDBCExamples class has been installed. It is installed together with the other Java examples classes.

   ☞ For more information about installing the Java examples classes, see "Setting up the Java sample" on page 82.

3. Call the method as follows:

   ```
   CALL JDBCExamples>>InsertFixed()
   ```

4. Confirm that a row has been added to the department table.

   ```
   SELECT *
   FROM department
   ```

   The row with ID 201 is not committed. You can execute a ROLLBACK statement to remove the row.

In this example, you have seen how to create a very simple JDBC class. Subsequent examples expand on this.

Passing arguments to Java methods

We can expand the **InsertFixed** method to illustrate how arguments are passed to Java methods.

The following method uses arguments passed in the call to the method as the values to insert:

```
public static void InsertArguments(
              String id, String name)  {
try {
    conn = DriverManager.getConnection(
               "jdbc:default:connection" );

String sqlStr =  "INSERT INTO Department "
   + " ( dept_id, dept_name )"
   + " VALUES (" +  id +  ", '"  + name + "')";

    // Execute the statement
    Statement stmt = conn.createStatement();
    Integer IRows = new Integer(
       stmt.executeUpdate( sqlStr.toString() ) );

    // Print the number of rows updated
    System.out.println(IRows.toString() + " row inserted" );
    }
    catch ( Exception e ) {
      System.out.println("Error: " + e.getMessage());
      e.printStackTrace();
    }
  }
```

Notes

♦ The two arguments are the department ID (an integer) and the department name (a string). Here, both arguments pass to the method as strings because they are part of the SQL statement string.

♦ The INSERT is a static statement and takes no parameters other than the SQL itself.

♦ If you supply the wrong number or type of arguments, you receive the `Procedure Not Found` error.

❖ **To use a Java method with arguments**

1. If you have not already installed the *JDBCExamples.class* file into the sample database, do so.

2. Connect to the sample database from Interactive SQL and enter the following command:

   ```
   call JDBCExamples>>InsertArguments(
       '203', 'Northern Sales' )
   ```

3. Verify that an additional row has been added to the Department table:

   ```
   SELECT *
   FROM Department
   ```

4. Roll back the changes to leave the database unchanged:

```
ROLLBACK
```

Queries using JDBC

The **Statement** object executes static queries, as well as statements that do not return result sets. For queries, you use the **executeQuery** method of the **Statement** object. This returns the result set in a **ResultSet** object.

The following code fragment illustrates how queries can be handled within JDBC. The code fragment places the total inventory value for a product into a variable named **inventory**. The product name is held in the **String** variable **prodname**. This example is available as the **Query** method of the **JDBCExamples** class.

The example assumes an internal or external connection has been obtained and is held in the Connection object named **conn**.

```
public static int Query () {
int max_price = 0;
    try{
      conn = DriverManager.getConnection(
                  "jdbc:default:connection" );

      // Build the query
      String sqlStr =  "SELECT id, unit_price "
          + "FROM product" ;

      // Execute the statement
      Statement stmt = conn.createStatement();
      ResultSet result = stmt.executeQuery( sqlStr );

      while( result.next() ) {
          int price = result.getInt(2);
          System.out.println( "Price is "  + price );
          if( price > max_price ) {
             max_price = price ;
          }
      }
    }
    catch( Exception e ) {
      System.out.println("Error: " + e.getMessage());
      e.printStackTrace();
    }
      return max_price;
}
```

Running the example
Once you have installed the **JDBCExamples** class into the sample database, you can execute this method using the following statement in Interactive SQL:

```
CALL JDBCExamples>>Query()
```

Notes

♦ The query selects the quantity and unit price for all products named **prodname**. These results are returned into the **ResultSet** object named **result**.

♦ There is a loop over each of the rows of the result set. The loop uses the **next** method.

♦ For each row, the value of each column is retrieved into an integer variable using the **getInt** method. **ResultSet** also has methods for other data types, such as **getString**, **getDate**, and **getBinaryString**.

The argument for the **getInt** method is an index number for the column, starting from 1.

♦ Adaptive Server Anywhere supports bidirectional scrolling cursors. However, JDBC provides only the **next** method, which corresponds to scrolling forward through the result set.

♦ The method returns the value of **max_price** to the calling environment, and Interactive SQL displays it on the Results tab in the Results pane.

Using prepared statements for more efficient access

If you use the **Statement** interface, you parse each statement you send to the database, generate an access plan, and execute the statement. The steps prior to actual execution are called **preparing** the statement.

You can achieve performance benefits if you use the **PreparedStatement** interface. This allows you to prepare a statement using placeholders, and then assign values to the placeholders when executing the statement.

Using prepared statements is particularly useful when carrying out many similar actions, such as inserting many rows.

☞ For more information about prepared statements, see "Preparing statements" on page 14.

Example

The following example illustrates how to use the **PreparedStatement** interface, although inserting a single row is not a good use of prepared statements.

The following method of the **JDBCExamples** class carries out a prepared statement:

```
public static void JInsertPrepared(int id, String name)
try {
    conn = DriverManager.getConnection(
                "jdbc:default:connection");

    // Build the INSERT statement
    // ? is a placeholder character
    String sqlStr = "INSERT INTO Department "
            + "( dept_id, dept_name ) "
            + "VALUES ( ? , ? )" ;

    // Prepare the statement
    PreparedStatement stmt =
        conn.prepareStatement( sqlStr );

    stmt.setInt(1, id);
    stmt.setString(2, name );
    Integer IRows = new Integer(
                        stmt.executeUpdate() );

    // Print the number of rows updated
    System.out.println(
        IRows.toString() + " row inserted" );
}
catch ( Exception e ) {
    System.out.println("Error: " + e.getMessage());
    e.printStackTrace();
}
}
```

Running the example

Once you have installed the **JDBCExamples** class into the sample database, you can execute this example by entering the following statement:

```
call JDBCExamples>>InsertPrepared(
                        202, 'Eastern Sales' )
```

The string argument is enclosed in single quotes, which is appropriate for SQL. If you invoke this method from a Java application, use double quotes to delimit the string.

Miscellaneous JDBC notes

♦ **Access permissions**   Like all Java classes in the database, classes containing JDBC statements can be accessed by any user. There is no equivalent to the GRANT EXECUTE statement that grants permission to execute procedures, and there is no need to qualify the name of a class with the name of its owner.

♦ **Execution permissions**   Java classes are executed with the permissions of the connection executing them. This behavior is different to that of stored procedures, which execute with the permissions of the owner.

Using JDBC escape syntax

You can use JDBC escape syntax from any JDBC application, including Interactive SQL. This escape syntax allows you to call stored procedures regardless of the database management system you are using. The general form for the escape syntax is

{{ **keyword** *parameters* }}

The braces *must* be doubled. This doubling is specific to Interactive SQL. There must not be a space between successive braces: "{{" is acceptable, but "{ {" is not. As well, you cannot use newline characters in the statement. The escape syntax cannot be used in stored procedures because they are not executed by Interactive SQL.

You can use the escape syntax to access a library of functions implemented by the JDBC driver that includes number, string, time, date, and system functions.

For example, to obtain the name of the current user in a database management system-neutral way, you would type the following:

```
select {{ fn user() }}
```

The functions that are available depend on the JDBC driver that you are using. The following tables list the functions that are supported by jConnect, and by the iAnywhere JDBC driver.

jConnect supported functions

Numeric functions	String functions	System functions	Time/Date functions
ABS	ASCII	DATABASE	CURDATE
ACOS	CHAR	IFNULL	CURTIME
ASIN	CONCAT	USER	DAYNAME
ATAN	DIFFERENCE	CONVERT	DAYOFMONTH
ATAN2	LCASE		DAYOFWEEK
CEILING	LENGTH		HOUR
COS	REPEAT		MINUTE
COT	RIGHT		MONTH
DEGREES	SOUNDEX		MONTHNAME
EXP	SPACE		NOW

Numeric functions	String functions	System functions	Time/Date functions
FLOOR	SUBSTRING		QUARTER
LOG	UCASE		SECOND
LOG10			TIMESTAMPADD
PI			TIMESTAMPDIFF
POWER			YEAR
RADIANS			
RAND			
ROUND			
SIGN			
SIN			
SQRT			
TAN			

iAnywhere JDBC driver supported functions

Numeric functions	String functions	System functions	Time/Date functions
ABS	ASCII	IFNULL	CURDATE
ACOS	CHAR	USERNAME	CURTIME
ASIN	CONCAT		DAYNAME
ATAN	DIFFERENCE		DAYOFMONTH
ATAN2	INSERT		DAYOFWEEK
CEILING	LCASE		DAYOFYEAR
COS	LEFT		HOUR
COT	LENGTH		MINUTE
DEGREES	LOCATE		MONTH
EXP	LOCATE_2		MONTHNAME
FLOOR	LTRIM		NOW
LOG	REPEAT		QUARTER

Numeric functions	String functions	System functions	Time/Date functions
LOG10	RIGHT		SECOND
MOD	RTRIM		WEEK
PI	SOUNDEX		YEAR
POWER	SPACE		
RADIANS	SUBSTRING		
RAND	UCASE		
ROUND			
SIGN			
SIN			
SQRT			
TAN			
TRUNCATE			

A statement using the escape syntax should work in Adaptive Server Anywhere, Adaptive Server Enterprise, Oracle, SQL Server, or another database management system to which you are connected.

For example, to obtain database properties with the sa_db_info procedure using SQL escape syntax, you would type the following in the SQL Statements pane in Interactive SQL:

```
{{CALL sa_db_info( 1 ) }}
```

Embedded SQL Programming

About this chapter   This chapter describes how to use the embedded SQL programming
interface to Adaptive Server Anywhere.

Contents

Introduction

Embedded SQL is a database-programming interface for the C and C++ programming languages. It consists of SQL statements intermixed with (embedded in) C or C++ source code. These SQL statements are translated by a **SQL preprocessor** into C or C++ source code, which you then compile.

At runtime, embedded SQL applications use an Adaptive Server Anywhere **interface library** to communicate **with** database server. The interface library is a dynamic link library (**DLL**) or shared library on most platforms.

♦ On Windows operating systems, the interface library is *dblib9.dll.*

♦ On UNIX operating systems, the interface library is *libdblib9.so*, *libdblib9.sl,* or *libdblib9.a*, depending on the operating system.

Adaptive Server Anywhere provides two flavors of embedded SQL. Static embedded SQL is simpler to use but less flexible than dynamic embedded SQL. Both flavors are discussed in this chapter.

Development process overview
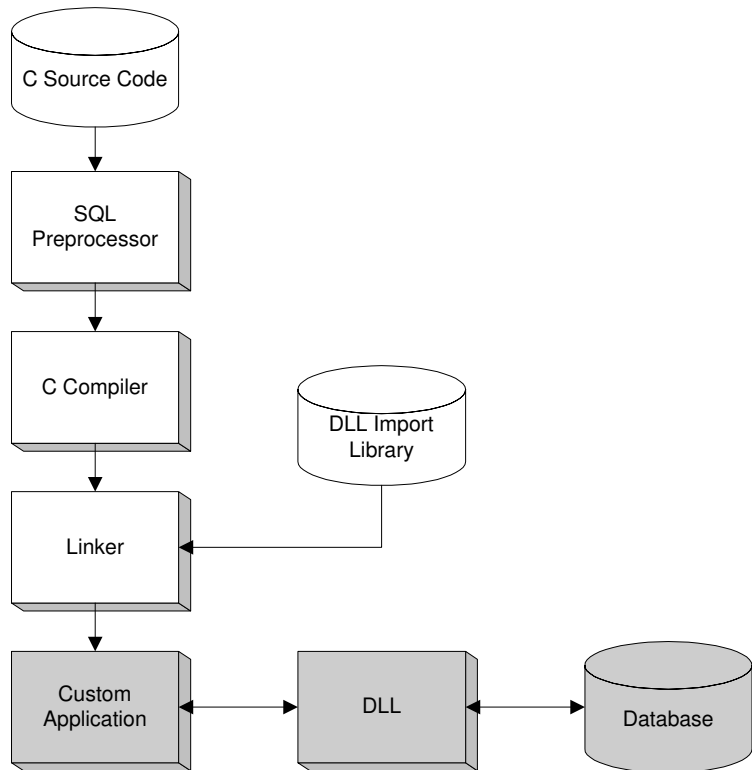
Once the program has been successfully preprocessed and compiled, it is linked with the **import library** for the Adaptive Server Anywhere interface library to form an executable file. When the database is running, this executable file uses the Adaptive Server Anywhere DLL to interact with the database. The database does not have to be running when the program is preprocessed.

For Windows, there are separate import libraries for Watcom C/C++, for Microsoft Visual C++, and for Borland C++.

☞ Using import libraries is the standard development method for applications that call functions in DLLs. Adaptive Server Anywhere also provides an alternative, and recommended method which avoids the use of import libraries. For more information, see "Loading the interface library dynamically" on page 141.

137

Running the SQL preprocessor

The SQL preprocessor is an executable named *sqlpp.exe*.

Command line   The SQLPP command line is as follows:

sqlpp [*options*] *sql-filename* [*output-filename*]

The SQL preprocessor processes a C program with embedded SQL before the C or C++ compiler is run. The preprocessor translates the SQL statements into C/C++ language source that is put into the output file. The normal extension for source programs with embedded SQL is *.sqc*. The default output filename is the *sql-filename* with an extension of *.c*. If the *sql-filename* already has a *.c* extension, then the output filename extension is *.cc* by default.

☞ For a full listing of the command-line options, see "The SQL preprocessor" on page 203.

Supported compilers

The C language SQL preprocessor has been used in conjunction with the following compilers:

Operating system	Compiler	Version
Windows	Watcom C/C++	9.5 and above
Windows	Microsoft Visual C/C++	1.0 and above
Windows	Borland C++	4.5
Windows CE	Microsoft Visual C/C++	5.0
UNIX	GNU or native compiler	
NetWare	Watcom C/C++	10.6, 11

☞ For instructions on building NetWare NLMs, see "Building NetWare Loadable Modules" on page 142.

Embedded SQL header files

All header files are installed in the *h* subdirectory of your SQL Anywhere installation directory.

Filename	Description
sqlca.h	Main header file included in all embedded SQL programs. This file includes the structure definition for the SQL Communication Area (SQLCA) and prototypes for all embedded SQL database interface functions.
sqlda.h	SQL Descriptor Area structure definition included in embedded SQL programs that use dynamic SQL.
sqldef.h	Definition of embedded SQL interface data types. This file also contains structure definitions and return codes needed for starting the database server from a C program.
sqlerr.h	Definitions for error codes returned in the **sqlcode** field of the SQLCA.
sqlstate.h	Definitions for ANSI/ISO SQL standard error states returned in the **sqlstate** field of the SQLCA.
pshpk1.h, *pshpk2.h*, *poppk.h*	These headers ensure that structure packing is handled correctly. They support Watcom C/C++, Microsoft Visual C++, IBM Visual Age, and Borland C/C++ compilers.

Import libraries

All import libraries are installed in the *lib* subdirectory, under the operating system subdirectory of the SQL Anywhere installation directory. For example, Windows import libraries are stored in the *win32\lib* subdirectory.

Operating system	Compiler	Import library
Windows	Watcom C/C++	*dblibtw.lib*
Windows	Microsoft Visual C++	*dblibtm.lib*
Windows CE	Microsoft Visual C++	*dblib9.lib*
NetWare	Watcom C/C++	*dblib9.lib*
Solaris (unthreaded applications)	All compilers	*libdblib9.so*, *libdbtasks9.so*
Solaris (threaded applications)	All compilers	*libdblib9_r.so*, *libdbtasks9_r.so*

The *libdbtasks9* libraries are called by the *libdblib9* library. Some compilers locate *libdbtasks9* automatically, while for others you need to specify it explicitly.

A simple example

The following is a very simple example of an embedded SQL program.

```
#include <stdio.h>
EXEC SQL INCLUDE SQLCA;
main()
{
   db_init( &sqlca );
   EXEC SQL WHENEVER SQLERROR GOTO error;
   EXEC SQL CONNECT "DBA" IDENTIFIED BY "SQL";
   EXEC SQL UPDATE employee
      SET emp_lname =    'Plankton'
      WHERE emp_id = 195;
   EXEC SQL COMMIT WORK;
   EXEC SQL DISCONNECT;
   db_fini( &sqlca );
   return( 0 );

   error:
   printf( "update unsuccessful -- sqlcode = %ld\n",
      sqlca.sqlcode );
   db_fini( &sqlca );
   return( -1 );
}
```

This example connects to the database, updates the last name of employee number 195, commits the change, and exits. There is virtually no interaction between the SQL and C code. The only thing the C code is used for in this example is control flow. The WHENEVER statement is used for error checking. The error action (GOTO in this example) is executed after any SQL statement that causes an error.

☞ For a description of fetching data, see

Structure of embedded SQL programs

SQL statements are placed (embedded) within regular C or C++ code. All embedded SQL statements start with the words EXEC SQL and end with a semicolon (;). Normal C language comments are allowed in the middle of embedded SQL statements.

Every C program using embedded SQL must contain the following statement before any other embedded SQL statements in the source file.

```
EXEC SQL INCLUDE SQLCA;
```

The first embedded SQL statement executed by the C program must be a CONNECT statement. The CONNECT statement is used to establish a connection with the database server and to specify the user ID that is used

for authorizing all statements executed during the connection.

The CONNECT statement must be the first embedded SQL statement executed. Some embedded SQL commands do not generate any C code, or do not involve communication with the database. These commands are thus allowed before the CONNECT statement. Most notable are the INCLUDE statement and the WHENEVER statement for specifying error processing.

Loading the interface library dynamically

The usual practice for developing applications that use functions from DLLs is to link the application against an **import library**, which contains the required function definitions.

This section describes an alternative to using an import library for developing Adaptive Server Anywhere applications. The Adaptive Server Anywhere interface library can be loaded dynamically, without having to link against the import library, using the *esqldll.c* module in the *src* subdirectory of your installation directory. Using *esqldll.c* is recommended because it is easier to use and more robust in its ability to locate the interface DLL.

❖ **To load the interface DLL dynamically**

1. Your program must call **db_init_dll** to load the DLL, and must call **db_fini_dll** to free the DLL. The **db_init_dll** call must be before any function in the database interface, and no function in the interface can be called after **db_fini_dll**.

   You must still call the **db_init** and **db_fini** library functions.

2. You must **#include** the *esqldll.h* header file before the EXEC SQL INCLUDE SQLCA statement or **#include** <*sqlca.h*> line in your embedded SQL program.

3. A SQL OS macro must be defined. The header file *sqlca.h*, which is included by *esqdll.c*, attempts to determine the appropriate macro and define it. However, certain combinations of platforms and compilers may cause this to fail. In this case, you must add a **#define** to the top of this file, or make the definition using a compiler option.

Macro	Platforms
_SQL_OS_WINNT	All Windows operating systems
_SQL_OS_UNIX	UNIX
_SQL_OS_NETWARE	NetWare

4. Compile *esqldll.c*.

5. Instead of linking against the imports library, link the object module *esqldll.obj* with your embedded SQL application objects.

Sample   You can find a sample program illustrating how to load the interface library dynamically in the *Samples\ASA\ESQLDynamicLoad* subdirectory of your SQL Anywhere directory. The source code is in *Samples\ASA\ESQLDynamicLoad\sample.sqc*.

Building NetWare Loadable Modules

You must use the Watcom C/C++ compiler, version 10.6 or 11.0, to compile embedded SQL programs as NetWare Loadable Modules (NLM).

❖ **To create an embedded SQL NLM**

1. On Windows, preprocess the embedded SQL file using the following command:

    ```
    sqlpp -o NETWARE srcfile.sqc
    ```

    This instruction creates a file with *.c* extension.

2. Compile *file.c* using the Watcom compiler (10.6 or 11.0), using the `/bt=netware` option.

3. Link the resulting object file using the Watcom linker with the following options:

    ```
    FORMAT NOVELL
    MODULE dblib9
    OPTION CASEEXACT
    IMPORT @dblib9.imp
    LIBRARY dblib9.lib
    ```

    The files *dblib9.imp* and *dblib9.lib* are shipped with Adaptive Server Anywhere, in the *nlm\lib* directory. The IMPORT and LIBRARY lines may require a full path.

142

Sample embedded SQL programs

Sample embedded SQL programs are included with the Adaptive Server Anywhere installation. They are placed in the *Samples\ASA\C* subdirectory of your SQL Anywhere directory.

♦ The static cursor embedded SQL example, *Samples\ASA\C\cur.sqc*, demonstrates the use of static SQL statements.

♦ The dynamic cursor embedded SQL example, *Samples\ASA\C\dcur.sqc*, demonstrates the use of dynamic SQL statements.

To reduce the amount of code that is duplicated by the sample programs, the mainlines and the data printing functions have been placed into a separate file. This is *mainch.c* for character mode systems and *mainwin.c* for windowing environments.

The sample programs each supply the following three routines, which are called from the mainlines.

♦ **WSQLEX_Init**   Connects to the database and opens the cursor.

♦ **WSQLEX_Process_Command**   Processes commands from the user, manipulating the cursor as necessary.

♦ **WSQLEX_Finish**   Closes the cursor and disconnect from the database.

The function of the mainline is to:

1. Call the **WSQLEX_Init** routine

2. Loop, getting commands from the user and calling **WSQL_Process_Command** until the user quits

3. Call the **WSQLEX_Finish** routine

Connecting to the database is accomplished with the embedded SQL CONNECT command supplying the appropriate user ID and password.

In addition to these samples, you may find other programs and source files as part of SQL Anywhere Studio which demonstrate features available for particular platforms.

Building the sample programs

Files to build the sample programs are supplied with the sample code.

♦ For Windows and NetWare operating systems, hosted on Windows operating systems, use *makeall.bat* to compile the sample programs.

- For UNIX, use the shell script *makeall.*

- For Windows CE, use the *dcur.dsp* project file for Microsoft Visual C++.

The format of the command is as follows:

```
makeall {Example} {Platform} {Compiler}
```

The first parameter is the name of the example program that you want to compile. It is one of the following:

- **CUR**   static cursor example

- **DCUR**   dynamic cursor example

- **ODBC**   ODBC example

The second parameter is the target platform. It is one of the following:

- **WINNT**   compile for Windows.

- **NETWARE**   compile for NetWare NLM

The third parameter is the compiler to use to compile the program. The compiler can be one of:

- **WC**   use Watcom C/C++

- **MC**   use Microsoft C

- **BC**   use Borland C

Running the sample programs

The executable files are held in the *Samples\ASA\C* directory, together with the source code.

❖ **To run the static cursor sample program**

1. Start the program:
   - Start the Adaptive Server Anywhere Personal Server Sample database.
   - Run the file *Samples\ASA\C\curwnt.exe.*

2. Follow the on-screen instructions.

   The various commands manipulate a database cursor and print the query results on the screen. Type the letter of the command you wish to perform. Some systems may require you to press ENTER after the letter.

❖ **To run the dynamic cursor sample program**

1. Start the program:

   ♦ Run the file *Samples\ASA\C\dcurwnt.exe.*

2. Connect to a database:

   ♦ Each sample program presents a console-type user interface and
     prompts you for a command. Enter the following connection string to
     connect to the sample database:

     ```
     DSN=ASA 9.0 Sample
     ```

3. Choose a table:

   ♦ Each sample program prompts you for a table. Choose one of the
     tables in the sample database. For example, you may enter **Customer**
     or **Employee**.

4. Follow the on-screen instructions.

   The various commands manipulate a database cursor and print the query
   results on the screen. Type the letter of the command you wish to
   perform. Some systems may require you to press ENTER after the letter.

Windows samples   The Windows versions of the example programs are real Windows
programs. However, to keep the user interface code relatively simple, some
simplifications have been made. In particular, these applications do not
repaint their Windows on WM_PAINT messages except to reprint the
prompt.

Static cursor sample

This example demonstrates the use of cursors. The particular cursor used
here retrieves certain information from the **employee** table in the sample
database. The cursor is declared statically, meaning that the actual SQL
statement to retrieve the information is "hard coded" into the source
program. This is a good starting point for learning how cursors work. The
next example ("Dynamic cursor sample" on page 146) takes this first
example and converts it to use dynamic SQL statements.

☞ For information on where the source code can be found and how to build
this example program, see "Sample embedded SQL programs" on page 143.

The **open_cursor** routine both declares a cursor for the specific SQL
command and also opens the cursor.

Printing a page of information is accomplished by the **print** routine. It loops
pagesize times, fetching a single row from the cursor and printing it out.
Note that the fetch routine checks for warning conditions (such as Row not

found) and prints appropriate messages when they arise. In addition, the cursor is repositioned by this program to the row before the one that appears at the top of the current page of data.

The **move**, **top,** and **bottom** routines use the appropriate form of the FETCH statement to position the cursor. Note that this form of the FETCH statement doesn't actually get the data—it only positions the cursor. Also, a general relative positioning routine, **move,** has been implemented to move in either direction depending on the sign of the parameter.

When the user quits, the cursor is closed and the database connection is also released. The cursor is closed by a ROLLBACK WORK statement, and the connection is release by a DISCONNECT.

Dynamic cursor sample

This sample demonstrates the use of cursors for a dynamic SQL SELECT statement. It is a slight modification of the static cursor example. If you have not yet looked at , it would be helpful to do so before looking at this sample.

☞ For information on where the source code can be found and how to build this sample program, see .

The **dcur** program allows the user to select a table to look at with the **n** command. The program then presents as much information from that table as fits on the screen.

When this program is run, it prompts for a connection string of the form:

```
uid=DBA;pwd=SQL;dbf=c:\asa\asademo.db
```

The C program with the embedded SQL is held in the *Samples\ASA\C* subdirectory of your SQL Anywhere directory. The program looks much like the static cursor sample with the exception of the **connect**, **open_cursor**, and **print** functions.

The **connect** function uses the embedded SQL interface function **db_string_connect** to connect to the database. This function provides the extra functionality to support the connection string that is used to connect to the database.

The **open_cursor** routine first builds the SELECT statement

```
SELECT * FROM tablename
```

where *tablename* is a parameter passed to the routine. It then prepares a dynamic SQL statement using this string.

The embedded SQL DESCRIBE command is used to fill in the SQLDA structure the results of the SELECT statement.

Size of the SQLDA
An initial guess is taken for the size of the SQLDA (3). If this is not big enough, the actual size of the select list returned by the database server is used to allocate a SQLDA of the correct size.

The SQLDA structure is then filled with buffers to hold strings that represent the results of the query. The **fill_s_sqlda** routine converts all data types in the SQLDA to DT_STRING and allocates buffers of the appropriate size.

A cursor is then declared and opened for this statement. The rest of the routines for moving and closing the cursor remain the same.

The **fetch** routine is slightly different: it puts the results into the SQLDA structure instead of into a list of host variables. The **print** routine has changed significantly to print results from the SQLDA structure up to the width of the screen. The **print** routine also uses the name fields of the SQLDA to print headings for each column.

Service examples

The example programs *cur.sqc* and *dcur.sqc*, when compiled for a version of Windows that supports services, run optionally as services.

The two files containing the example code for Windows services are the source file *ntsvc.c* and the header file *ntsvc.h*. The code allows a linked executable to be run either as a regular executable or as a Windows service.

❖ **To run one of the compiled examples as a Windows service**

1. Start Sybase Central.

2. In the left pane, select Adaptive Server Anywhere 9.

3. In the right pane, select the Services tab.

4. From the File menu, choose New ➤ Service.
   The Service Creation wizard appears.

5. On the first page, enter a name for the service.

6. On the second page, select Sample program.

7. On the third page, browse to the sample program (*curwnt.exe* or *dcurwnt.exe*) from the *Samples\ASA\C* subdirectory of your SQL Anywhere directory.

8. Complete the wizard to install the service.

9. Click Start on the main window to start the service.

When run as a service, the programs display the normal user interface if possible. They also write the output to the Application Event Log. If it is not possible to start the user interface, the programs print one page of data to the Application Event Log and stop.

These examples have been tested with the Watcom C/C++ 10.5 compiler and the Microsoft Visual C++ compiler.

Embedded SQL data types

To transfer information between a program and the database server, every piece of data must have a data type. The embedded SQL data type constants are prefixed with DT_, and can be found in the *sqldef.h* header file. You can create a host variable of any one of the supported types. You can also use these types in a SQLDA structure for passing data to and from the database.

You can define variables of these data types using the DECL_ macros listed in *sqlca.h*. For example, a variable holding a BIGINT value could be declared with DECL_BIGINT.

The following data types are supported by the embedded SQL programming interface:

♦ **DT_BIT**   8-bit signed integer

♦ **DT_SMALLINT**   16-bit signed integer.

♦ **DT_UNSSMALLINT**   16-bit unsigned integer

♦ **DT_TINYINT**   8-bit signed integer

♦ **DT_BIGINT**   64-bit signed integer

♦ **DT_INT**   32-bit signed integer.

♦ **DT_UNSINT**   16-bit unsigned integer

♦ **DT_FLOAT**   4-byte floating point number.

♦ **DT_DOUBLE**   8-byte floating point number.

♦ **DT_DECIMAL**   Packed decimal number.

```
typedef struct DECIMAL {
  char array[1];
} DECIMAL;
```

♦ **DT_STRING**   NULL-terminated character string. The string is blank-padded if the database is initialized with blank-padded strings.

♦ **DT_DATE**   NULL-terminated character string that is a valid date.

♦ **DT_TIME**   NULL-terminated character string that is a valid time.

♦ **DT_TIMESTAMP**   NULL-terminated character string that is a valid timestamp.

♦ **DT_FIXCHAR**   Fixed-length blank padded character string.

♦ **DT_VARCHAR**   Varying length character string with a two-byte length field. When supplying information to the database server, you must set the length field. When fetching information from the database server, the server sets the length field (not padded).

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

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

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

The DECL_LONGVARCHAR struct may be used with more than 32K of data. Large data may be fetched all at once, or in pieces using the GET DATA statement. Large data may be supplied to the server all at once, or in pieces by appending to a database variable using the SET statement. The data is not null terminated.

☞ For more information, see

♦ **DT_BINARY**   Varying length binary data with a two-byte length field. When supplying information to the database server, you must set the length field. When fetching information from the database server, the server sets the length field.

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

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

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

The DECL_LONGBINARY struct may be used with more than 32K of data. Large data may be fetched all at once, or in pieces using the GET

DATA statement. Large data may be supplied to the server all at once, or in pieces by appending to a database variable using the SET statement.

☞ For more information, see "Sending and retrieving long values" on page 190.

♦ **DT_TIMESTAMP_STRUCT**   SQLDATETIME structure with fields for each part of a timestamp.

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

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

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

☞ For more information, see the DATE_FORMAT, TIME_FORMAT, TIMESTAMP_FORMAT, and DATE_ORDER database options in "Database Options" [*ASA Database Administration Guide,* page 613].

♦ **DT_VARIABLE**   NULL-terminated character string. The character string must be the name of a SQL variable whose value is used by the database server. This data type is used only for supplying data to the database server. It cannot be used when fetching data from the database server.

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

DATE and TIME database types

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

☞ For more information see "GET DATA statement [ESQL]" [*ASA SQL Reference,* page 497] and "SET statement" [*ASA SQL Reference,* page 604].

Using host variables

Host variables are C variables that are identified to the SQL preprocessor. Host variables can be used to send values to the database server or receive values from the database server.

Host variables are quite easy to use, but they have some restrictions. Dynamic SQL is a more general way of passing information to and from the database server using a structure known as the SQL Descriptor Area (SQLDA). The SQL preprocessor automatically generates a SQLDA for each statement in which host variables are used.

☞ For information on dynamic SQL, see "Static and dynamic SQL" on page 176.

Declaring host variables

Host variables are defined by putting them into a **declaration section**. According to the IBM SAA and ANSI embedded SQL standards, host variables are defined by surrounding the normal C variable declarations with the following:

```
EXEC SQL BEGIN DECLARE SECTION;
/* C variable declarations */
EXEC SQL END DECLARE SECTION;
```

These host variables can then be used in place of value constants in any SQL statement. When the database server executes the command, the value of the host variable is used. Note that host variables cannot be used in place of table or column names: dynamic SQL is required for this. The variable name is prefixed with a colon (:) in a SQL statement to distinguish it from other identifiers allowed in the statement.

A standard SQL preprocessor does not scan C language code except inside a DECLARE SECTION. Thus, TYPEDEF types and structures are not allowed. Initializers on the variables are allowed inside a DECLARE SECTION.

Example
♦ The following sample code illustrates the use of host variables on an INSERT command. The variables are filled in by the program and then inserted into the database:

```
EXEC SQL BEGIN DECLARE SECTION;
long employee_number;
char employee_name[50];
char employee_initials[8];
char employee_phone[15];
EXEC SQL END DECLARE SECTION;
/* program fills in variables with appropriate values
*/
EXEC SQL INSERT INTO Employee
VALUES (:employee_number, :employee_name,
:employee_initials, :employee_phone );
```

☞ For a more extensive example, see "Static cursor sample" on page 145.

C host variable types

Only a limited number of C data types are supported as host variables. Also, certain host variable types do not have a corresponding C type.

Macros defined in the *sqlca.h* header file can be used to declare host variables of the following types: VARCHAR, FIXCHAR, BINARY, PACKED DECIMAL, LONG VARCHAR, LONG BINARY, or SQLDATETIME structure. They are used as follows:

```
EXEC SQL BEGIN DECLARE SECTION;
DECL_VARCHAR( 10 ) v_varchar;
DECL_FIXCHAR( 10 ) v_fixchar;
DECL_LONGVARCHAR( 32678 ) v_longvarchar;
DECL_BINARY( 4000 ) v_binary;
DECL_LONGBINARY( 128000 ) v_longbinary;
DECL_DECIMAL( 10, 2 ) v_packed_decimal;
DECL_DATETIME v_datetime;
EXEC SQL END DECLARE SECTION;
```

The preprocessor recognizes these macros within a declaration section and treats the variable as the appropriate type.

The following table lists the C variable types that are allowed for host variables and their corresponding embedded SQL interface data types.

C Data Type	Embedded SQL Interface Type	Description

`short i;`
`short int i;`
`unsigned short int i;`
	DT_SMALLINT	16-bit signed integer

C Data Type	Embedded SQL Interface Type	Description

`long l;`
`long int l;`
`unsigned long int l;`
	DT_INT	32-bit signed integer

`float f;`	DT_FLOAT	4-byte floating point
`double d;`	DT_DOUBLE	8-byte floating point
`DECL_DECIMAL(p,s)`	DT_DECIMAL(p,s)	Packed decimal

`char a; /*n=1*/`
`DECL_FIXCHAR(n) a;`
`DECL_FIXCHAR a[n];`
	DT_FIXCHAR(n)	Fixed length character string blank padded.

`char a[n]; /*n>=1*/`	DT_STRING(n)	NULL-terminated string. The string is blank-padded if the database is initialized with blank-padded strings.
`char *a;`	DT_STRING(32767)	NULL-terminated string
`DECL_VARCHAR(n) a;`	DT_VARCHAR(n)	Varying length character string with 2-byte length field. Not blank padded
`DECL_BINARY(n) a;`	DT_BINARY(n)	Varying length binary data with 2-byte length field
`DECL_DATETIME a;`	DT_TIMESTAMP_-STRUCT	SQLDATETIME structure
`DECL_LONGVARCHAR( n ) a;`	DT_LONGVARCHAR	Varying length long character string with three 4-byte length fields. Not blank padded or NULL terminated.

C Data Type	Embedded SQL Interface Type	Description
`DECL_LONGBINARY( n ) a;`	DT_LONGBINARY	Varying length long binary data with three 4-byte length fields. Not blank padded.

Pointers to char

A host variable declared as a **pointer to char** (*char *a*) is considered by the database interface to be 32 767 bytes long. Any host variable of type **pointer to char** used to retrieve information from the database must point to a buffer large enough to hold any value that could possibly come back from the database.

This is potentially quite dangerous because somebody could change the definition of the column in the database to be larger than it was when the program was written. This could cause random memory corruption problems. If you are using a 16-bit compiler, requiring 32 767 bytes could make the program stack overflow. It is better to use a declared array, even as a parameter to a function, where it is passed as a **pointer to char**. This lets the PREPARE statements know the size of the array.

Scope of host variables

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

As far as the SQL preprocessor is concerned, host variables are global; two host variables cannot have the same name.

Host variable usage

Host variables can be used in the following circumstances:

♦ SELECT, INSERT, UPDATE and DELETE statements in any place where a number or string constant is allowed.

♦ The INTO clause of SELECT and FETCH statements.

♦ Host variables can also be used in place of a statement name, a cursor name, or an option name in commands specific to embedded SQL.

♦ For CONNECT, DISCONNECT, and SET CONNECT, a host variable can be used in place of a user ID, password, connection name, connection string, or database environment name.

♦ For SET OPTION and GET OPTION, a host variable can be used in place of a user ID, option name, or option value.

♦ Host variables cannot be used in place of a table name or a column name in any statement.

Examples   ♦ The following is valid embedded SQL:

```
INCLUDE SQLCA;
long SQLCODE;
sub1() {
    char SQLSTATE[6];
    exec SQL CREATE TABLE ...
}
```

♦ The following is not valid embedded SQL:

```
INCLUDE SQLCA;
sub1() {
    char SQLSTATE[6];
    exec SQL CREATE TABLE...
}
sub2() {
    exec SQL DROP TABLE...
    // No SQLSTATE in scope of this statement
}
```

♦ The case of SQLSTATE and SQLCODE is important and the ISO/ANSI standard requires that their definitions be exactly as follows:

```
long SQLCODE;
char SQLSTATE[6];
```

Indicator variables

Indicator variables are C variables that hold supplementary information when you are fetching or putting data. There are several distinct uses for indicator variables:

♦ **NULL values**   To enable applications to handle NULL values.

♦ **String truncation**   To enable applications to handle cases when fetched values must be truncated to fit into host variables.

♦ **Conversion errors**   To hold error information.

An indicator variable is a host variable of type **short int** that is placed immediately following a regular host variable in a SQL statement. For example, in the following INSERT statement, **:ind_phone** is an indicator variable:

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

Using indicator variables to handle NULL

In SQL data, NULL represents either an unknown attribute or inapplicable information. The SQL concept of NULL is not to be confused with the C language constant by the same name (**NULL**). The C constant is used to represent a non-initialized or invalid pointer.

When NULL is used in the Adaptive Server Anywhere documentation, it refers to the SQL database meaning given above. The C language constant is referred to as the **null** pointer (lower case).

NULL is not the same as any value of the column's defined type. Thus, in order to pass NULL values to the database or receive NULL results back, something extra is required beyond regular host variables. **Indicator variables** are used for this purpose.

Using indicator variables when inserting NULL

An INSERT statement could include an indicator variable as follows:

```
EXEC SQL BEGIN DECLARE SECTION;
short int employee_number;
char employee_name[50];
char employee_initials[6];
char employee_phone[15];
short int ind_phone;
EXEC SQL END DECLARE SECTION;
/*
program fills in empnum, empname,
initials and homephone
*/
if( /* phone number is unknown */ ) {
   ind_phone = -1;
} else {
   ind_phone = 0;
}
EXEC SQL INSERT INTO Employee
   VALUES (:employee_number, :employee_name,
   :employee_initials, :employee_phone:ind_phone );
```

If the indicator variable has a value of –1, a NULL is written. If it has a value of 0, the actual value of **employee_phone** is written.

Using indicator variables when fetching NULL

Indicator variables are also used when receiving data from the database. They are used to indicate that a NULL value was fetched (indicator is negative). If a NULL value is fetched from the database and an indicator variable is not supplied, an error is generated (SQLE_NO_INDICATOR).

Errors are explained in the next section.

Using indicator variables for truncated values

Indicator variables indicate whether any fetched values were truncated to fit into a host variable. This enables applications to handle truncation appropriately.

If a value is truncated on fetching, the indicator variable is set to a positive value, containing the actual length of the database value before truncation. If the length of the value is greater than 32 767, then the indicator variable contains 32 767.

Using indicator values for conversion errors

By default, the CONVERSION_ERROR database option is set to ON, and any data type conversion failure leads to an error, with no row returned.

You can use indicator variables to tell which column produced a data type conversion failure. If you set the database option CONVERSION_ERROR to OFF, any data type conversion failure gives a CANNOT_CONVERT warning, rather than an error. If the column that suffered the conversion error has an indicator variable, that variable is set to a value of –2.

If you set the CONVERSION_ERROR option to OFF when inserting data into the database, a value of NULL is inserted when a conversion failure occurs.

Summary of indicator variable values

The following table provides a summary of indicator variable usage.

Indicator Value	Supplying Value to database	Receiving value from database
> 0	Host variable value	Retrieved value was truncated — actual length in indicator variable
0	Host variable value	Fetch successful, or CONVERSION_-ERROR set to ON
–1	NULL value	NULL result
–2	NULL value	Conversion error (when CONVERSION_ERROR is set to OFF only). SQLCODE indicates a CANNOT_CONVERT warning
< –2	NULL value	NULL result

☞ For more information on retrieving long values, see "GET DATA statement [ESQL]" [*ASA SQL Reference,* page 497].

The SQL Communication Area (SQLCA)

The **SQL Communication Area** (**SQLCA**) is an area of memory that is used on every database request for communicating statistics and errors from the application to the database server and back to the application. The SQLCA is used as a handle for the application-to-database communication link. It is passed in to all database library functions that need to communicate with the database server. It is implicitly passed on all embedded SQL statements.

A global SQLCA variable is defined in the interface library. The preprocessor generates an external reference for the global SQLCA variable and an external reference for a pointer to it. The external reference is named **sqlca** and is of type SQLCA. The pointer is named **sqlcaptr**. The actual global variable is declared in the imports library.

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

SQLCA provides error codes

You reference the SQLCA to test for a particular error code. The **sqlcode** and **sqlstate** fields contain error codes when a database request has an error (see below). Some C macros are defined for referencing the **sqlcode** field, the **sqlstate** field, and some other fields.

SQLCA fields

The fields in the SQLCA have the following meanings:

♦ **sqlcaid**   An 8-byte character field that contains the string **SQLCA** as an identification of the SQLCA structure. This field helps in debugging when you are looking at memory contents.

♦ **sqlcabc**   A long integer that contains the length of the SQLCA structure (136 bytes).

♦ **sqlcode**   A long integer that specifies the error code when the database detects an error on a request. Definitions for the error codes can be found in the header file *sqlerr.h*. The error code is 0 (zero) for a successful operation, positive for a warning and negative for an error.

☞ For a full listing of error codes, see *ASA Error Messages*.

♦ **sqlerrml**   The length of the information in the **sqlerrmc** field.

♦ **sqlerrmc**   Zero or more character strings to be inserted into an error message. Some error messages contain one or more placeholder strings (*%1*, *%2*, . . .) which are replaced with the strings in this field.

For example, if a `Table Not Found` error is generated, **sqlerrmc** contains the table name, which is inserted into the error message at the appropriate place.

☞ For a full listing of error messages, see *ASA Error Messages*.

♦ **sqlerrp**   Reserved.

♦ **sqlerrd**   A utility array of long integers.

♦ **sqlwarn**   Reserved.

♦ **sqlstate**   The SQLSTATE status value. The ANSI SQL standard (SQL-92) defines a new type of return value from a SQL statement in addition to the SQLCODE value in previous standards. The SQLSTATE value is always a five-character null-terminated string, divided into a two-character class (the first two characters) and a three-character subclass. Each character can be a digit from 0 through 9 or an upper case alphabetic character A through Z.

Any class or subclass that begins with 0 through 4 or A through H is defined by the SQL standard; other classes and subclasses are implementation defined. The SQLSTATE value '00000' means that there has been no error or warning.

☞ For more SQLSTATE values, see *ASA Error Messages*.

sqlerror array   The **sqlerror** field array has the following elements.

♦ **sqlerrd[1] (SQLIOCOUNT)**   The actual number of input/output operations that were required to complete a command.

The database does not start this number at zero for each command. Your program can set this variable to zero before executing a sequence of commands. After the last command, this number is the total number of input/output operations for the entire command sequence.

♦ **sqlerrd[2] (SQLCOUNT)**   The value of this field depends on which statement is being executed.

• **INSERT, UPDATE, PUT, and DELETE statements**   The number of rows that were affected by the statement.

On a cursor OPEN, this field is filled in with either the actual number of rows in the cursor (a value greater than *or equal to* 0) or an estimate thereof (a negative number whose absolute value is the estimate). It is the actual number of rows if the database server can compute it without counting the rows. The database can also be configured to always return the actual number of rows using the ROW_COUNT option.

- **FETCH cursor statement**   The SQLCOUNT field is filled if a
  SQLE_NOTFOUND warning is returned. It contains the number of
  rows by which a FETCH RELATIVE or FETCH ABSOLUTE
  statement goes outside the range of possible cursor positions (a cursor
  can be on a row, before the first row, or after the last row). In the case
  of a wide fetch, SQLCOUNT is the number of rows actually fetched,
  and is less than or equal to the number of rows requested. During a
  wide fetch, SQLE_NOTFOUND is *not* set.

  ☞ For more information on wide fetches, see "Fetching more than
  one row at a time" on page 170.

  The value is 0 if the row was not found but the position is valid, for
  example, executing FETCH RELATIVE 1 when positioned on the last
  row of a cursor. The value is positive if the attempted fetch was beyond
  the end of the cursor, and negative if the attempted fetch was before the
  beginning of the cursor.

- **GET DATA statement**   The SQLCOUNT field holds the actual length
  of the value.

- **DESCRIBE statement**   In the WITH VARIABLE RESULT clause
  used to describe procedures that may have more than one result set,
  SQLCOUNT is set to one of the following values:

  - ♦ **0**   The result set may change: the procedure call should be
    described again following each OPEN statement.
  - ♦ **1**   The result set is fixed. No re-describing is required.

  In the case of a syntax error, SQLE_SYNTAX_ERROR, this field
  contains the approximate character position within the command string
  where the error was detected.

♦ **sqlerrd[3] (SQLIOESTIMATE)**   The estimated number of input/output
operations that are to complete the command. This field is given a value
on an OPEN or EXPLAIN command.

SQLCA management for multi-threaded or reentrant code

You can use embedded SQL statements in multi-threaded or reentrant code.
However, if you use a single connection, you are restricted to one active
request per connection. In a multi-threaded application, you should not use
the same connection to the database on each thread unless you use a
semaphore to control access.

There are no restrictions on using separate connections on each thread that
wishes to use the database. The SQLCA is used by the runtime library to
distinguish between the different thread contexts. Thus, each thread wishing
to use the database must have its own SQLCA.

Any given database connection is accessible only from one SQLCA, with the exception of the cancel instruction, which must be issued from a separate thread.

☞ For information on canceling requests, see "Implementing request management" on page 201.

Using multiple SQLCAs

❖ **To manage multiple SQLCAs in your application**

1. You must use the option on the SQL preprocessor that generates reentrant code (-r). The reentrant code is a little larger and a little slower because statically initialized global variables cannot be used. However, these effects are minimal.

2. Each SQLCA used in your program must be initialized with a call to **db_init** and cleaned up at the end with a call to **db_fini**.

   > *Caution*
   > *Failure to call db_fini for each db_init on NetWare can cause the database server to fail and the NetWare file server to fail.*

3. The embedded SQL statement SET SQLCA ("SET SQLCA statement [ESQL]" [*ASA SQL Reference,* page 619]) is used to tell the SQL preprocessor to use a different SQLCA for database requests. Usually, a statement such as: EXEC SQL SET SQLCA 'task_data->sqlca'; is used at the top of your program or in a header file to set the SQLCA reference to point at task specific data. This statement does not generate any code and thus has no performance impact. It changes the state within the preprocessor so that any reference to the SQLCA uses the given string.

☞ For information about creating SQLCAs, see "SET SQLCA statement [ESQL]" [*ASA SQL Reference,* page 619].

When to use multiple SQLCAs

You can use the multiple SQLCA support in any of the supported embedded SQL environments, but it is only required in reentrant code.

The following list details the environments where multiple SQLCAs must be used:

♦ **Multi-threaded applications** If more than one thread uses the same SQLCA, a context option can cause more than one thread to be using the SQLCA at the same time. Each thread must have its own SQLCA. This

can also happen when you have a DLL that uses embedded SQL and is
called by more than one thread in your application.

♦ **Dynamic link libraries and shared libraries**   A DLL has only one data
segment. While the database server is processing a request from one
application, it may yield to another application that makes a request to the
database server. If your DLL uses the global SQLCA, both applications
are using it at the same time. Each Windows application must have its
own SQLCA.

♦ **A DLL with one data segment**   A DLL can be created with only one
data segment or one data segment for each application. If your DLL has
only one data segment, you cannot use the global SQLCA for the same
reason that a DLL cannot use the global SQLCA. Each application must
have its own SQLCA.

Connection management with multiple SQLCAs

You do not need to use multiple SQLCAs to connect to more than one
database or have more than one connection to a single database.

Each SQLCA can have one unnamed connection. Each SQLCA has an
active or current connection (see "SET CONNECTION statement
[Interactive SQL] [ESQL]" [*ASA SQL Reference,* page 610]). All operations on
a given database connection must use the same SQLCA that was used when
the connection was established.

> **Record locking**
> Operations on different connections are subject to the normal record
> locking mechanisms and may cause each other to block and possibly to
> deadlock. For information on locking, see the chapter "Using Transactions
> and Isolation Levels" [*ASA SQL User's Guide,* page 101].

Fetching data

Fetching data in embedded SQL is done using the SELECT statement. There are two cases:

♦ **The SELECT statement returns at most one row**   Use an INTO clause to assign the returned values directly to host variables.

  ☞ For information, see "SELECT statements that return at most one row" on page 166.

♦ **The SELECT statement may return multiple rows**   Use cursors to manage the rows of the result set.

  ☞ For more information, see "Using cursors in embedded SQL" on page 167.

☞ LONG VARCHAR and LONG BINARY data types are handled differently to other data types. For more information, see "Retrieving LONG data" on page 191.

SELECT statements that return at most one row

A single row query retrieves at most one row from the database. A single-row query SELECT statement has an INTO clause following the select list and before the FROM clause. The INTO clause contains a list of host variables to receive the value for each select list item. There must be the same number of host variables as there are select list items. The host variables may be accompanied by indicator variables to indicate NULL results.

When the SELECT statement is executed, the database server retrieves the results and places them in the host variables. If the query results contain more than one row, the database server returns an error.

If the query results in no rows being selected, a `Row Not Found` warning is returned. Errors and warnings are returned in the SQLCA structure, as described in "The SQL Communication Area (SQLCA)" on page 161.

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

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

Using cursors in embedded SQL

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

☞ For an introduction to cursors, see "Working with cursors" on page 21.

❖ **To manage a cursor in embedded SQL**

1. Declare a cursor for a particular SELECT statement, using the DECLARE statement.

2. Open the cursor using the OPEN statement.

3. Retrieve results one row at a time from the cursor using the FETCH statement.

4. Fetch rows until the Row Not Found warning is returned.

   Errors and warnings are returned in the SQLCA structure, described in "The SQL Communication Area (SQLCA)" on page 161.

5. Close the cursor, using the CLOSE statement.

By default, cursors are automatically closed at the end of a transaction (on COMMIT or ROLLBACK). Cursors that are opened with a WITH HOLD clause are kept open for subsequent transactions until they are explicitly closed.

The following is a simple example of cursor usage:

```
void print_employees( void )
{
    EXEC SQL BEGIN DECLARE SECTION;
    char name[50];
    char sex;
    char birthdate[15];
    short int ind_birthdate;
    EXEC SQL END DECLARE SECTION;
    EXEC SQL DECLARE C1 CURSOR FOR
        SELECT   emp_fname || ' ' || emp_lname,
                 sex, birth_date
        FROM "DBA".employee;
    EXEC SQL OPEN C1;
    for( ;; ) {
        EXEC SQL FETCH C1 INTO :name, :sex,
          :birthdate:ind_birthdate;
        if( SQLCODE == SQLE_NOTFOUND ) {
           break;
        } else if( SQLCODE < 0 ) {
           break;
        }
        if( ind_birthdate < 0 ) {
           strcpy( birthdate, "UNKNOWN" );
        }
        printf( "Name: %s Sex: %c Birthdate:
                %s.n",name, sex, birthdate );
    }
  EXEC SQL CLOSE C1;
}
```

☞ For complete examples using cursors, see "Static cursor sample" on page 145 and "Dynamic cursor sample" on page 146.

Cursor positioning   A cursor is positioned in one of three places:

♦ On a row

♦ Before the first row

♦ After the last row

Absolute row
from start

Absolute row
from end

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

When a cursor is opened, it is positioned before the first row. The cursor position can be moved using the FETCH command (see "FETCH statement [ESQL] [SP]" [*ASA SQL Reference,* page 482]). It can be positioned to an absolute position either from the start or from the end of the query results. It can also be moved relative to the current cursor position.

There are special *positioned* versions of the UPDATE and DELETE statements that can be used to update or delete the row at the current position of the cursor. If the cursor is positioned before the first row or after the last row, a No Current Row of Cursor error is returned.

The PUT statement can be used to insert a row into a cursor.

Cursor positioning
problems

Inserts and some updates to DYNAMIC SCROLL cursors can cause problems with cursor positioning. The database server does not put inserted rows at a predictable position within a cursor unless there is an ORDER BY clause on the SELECT statement. In some cases, the inserted row does not appear at all until the cursor is closed and opened again.

With Adaptive Server Anywhere, this occurs if a temporary table had to be created to open the cursor.

☞ For a description, see "Use of work tables in query processing" [*ASA SQL User's Guide,* page 190].

The UPDATE statement may cause a row to move in the cursor. This happens if the cursor has an ORDER BY clause that uses an existing index (a temporary table is not created).

Fetching more than one row at a time

The FETCH statement can be modified to fetch more than one row at a time, which may improve performance. This is called a **wide fetch** or an **array fetch**.

☞ Adaptive Server Anywhere also supports wide puts and inserts. For information on these, see "PUT statement [ESQL]" [*ASA SQL Reference,* page 567] and "EXECUTE statement [ESQL]" [*ASA SQL Reference,* page 471].

To use wide fetches in embedded SQL, include the fetch statement in your code as follows:

```
EXEC SQL FETCH . . . ARRAY nnn
```

where ARRAY *nnn* is the last item of the FETCH statement. The fetch count *nnn* can be a host variable. The number of variables in the SQLDA must be the product of *nnn* and the number of columns per row. The first row is placed in SQLDA variables 0 to (columns per row) – 1, and so on.

Each column must be of the same type in each row of the SQLDA, or a SQLDA_INCONSISTENT error is returned.

The server returns in SQLCOUNT the number of records that were fetched, which is always greater than zero unless there is an error or warning. On a wide fetch, a SQLCOUNT of one with no error condition indicates that one valid row has been fetched.

Example   The following example code illustrates the use of wide fetches. You can also find this code as *samples\ASA\esqlwidefetch\widefetch.sqc* in your SQL Anywhere directory.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "sqldef.h"
EXEC SQL INCLUDE SQLCA;

EXEC SQL WHENEVER SQLERROR { PrintSQLError();
              goto err; };

static void PrintSQLError()
/***********************/
{
    char buffer[200];

    printf( "SQL error %d -- %s\n",
        SQLCODE,
        sqlerror_message( &sqlca,
               buffer,
               sizeof( buffer ) ) );
}
static SQLDA * PrepareSQLDA(
  a_sql_statement_number   stat0,
  unsigned        width,
  unsigned        *cols_per_row )
/*******************************************/
/* Allocate a SQLDA to be used for fetching from
   the statement identified by "stat0". "width"
   rows will be retrieved on each FETCH request.
   The number of columns per row is assigned to
   "cols_per_row". */
{
    int                  num_cols;
    unsigned             row, col, offset;
    SQLDA *              sqlda;
    EXEC SQL BEGIN DECLARE SECTION;
    a_sql_statement_number  stat;
    EXEC SQL END DECLARE SECTION;

    stat = stat0;
    sqlda = alloc_sqlda( 100 );
    if( sqlda == NULL ) return( NULL );
    EXEC SQL DESCRIBE :stat INTO sqlda;
    *cols_per_row = num_cols = sqlda->sqld;
    if( num_cols * width > sqlda->sqln ) {
        free_sqlda( sqlda );
        sqlda = alloc_sqlda( num_cols * width );
        if( sqlda == NULL ) return( NULL );
        EXEC SQL DESCRIBE :stat INTO sqlda;
    }
```

```
        // copy first row in SQLDA setup by describe
        // to following (wide) rows
        sqlda->sqld = num_cols * width;
        offset = num_cols;
        for( row = 1; row < width; row++ ) {
            for( col = 0;
            col < num_cols;
                col++, offset++ ) {
                sqlda->sqlvar[offset].sqltype =
                        sqlda->sqlvar[col].sqltype;
                sqlda->sqlvar[offset].sqllen =
                        sqlda->sqlvar[col].sqllen;
          // optional: copy described column name
                memcpy( &sqlda->sqlvar[offset].sqlname,
                        &sqlda->sqlvar[col].sqlname,
                        sizeof( sqlda->sqlvar[0].sqlname ) );
            }
        }
        fill_s_sqlda( sqlda, 40 );
        return( sqlda );
err:
        return( NULL );
}

static void PrintFetchedRows( SQLDA * sqlda,
                unsigned cols_per_row )
/*****************************************/
/* Print rows already wide fetched in the SQLDA */
{
        long                    rows_fetched;
        int             row, col, offset;

        if( SQLCOUNT == 0 ) {
            rows_fetched = 1;
        } else {
            rows_fetched = SQLCOUNT;
        }
        printf( "Fetched %d Rows:\n", rows_fetched );
        for( row = 0; row < rows_fetched; row++ ) {
            for( col = 0; col < cols_per_row; col++ ) {
                offset = row * cols_per_row + col;
                printf( " \"%s\"",
                    (char *)sqlda->sqlvar[offset]
                        .sqldata );
            }
            printf( "\n" );
        }
}
```

```
            static int DoQuery( char * query_str0,
                          unsigned fetch_width0 )
            /****************************************/
            /* Wide Fetch "query_str0" select statement
             * using a width of "fetch_width0" rows" */
            {
                SQLDA *                  sqlda;
                unsigned         cols_per_row;
                EXEC SQL BEGIN DECLARE SECTION;
                a_sql_statement_number  stat;
                char *            query_str;
                unsigned            fetch_width;
                EXEC SQL END DECLARE SECTION;

                query_str = query_str0;
                fetch_width = fetch_width0;

                EXEC SQL PREPARE :stat FROM :query_str;
                EXEC SQL DECLARE QCURSOR CURSOR FOR :stat
                        FOR READ ONLY;
                EXEC SQL OPEN QCURSOR;
                sqlda = PrepareSQLDA( stat,
                        fetch_width,
                        &cols_per_row );
                if( sqlda == NULL ) {
                    printf( "Error allocating SQLDA\n" );
                    return( SQLE_NO_MEMORY );
                }
                for( ;; ) {
                    EXEC SQL FETCH QCURSOR INTO DESCRIPTOR sqlda
                        ARRAY :fetch_width;
                    if( SQLCODE != SQLE_NOERROR ) break;
                        PrintFetchedRows( sqlda, cols_per_row );
                }
                EXEC SQL CLOSE QCURSOR;
                EXEC SQL DROP STATEMENT :stat;
                free_filled_sqlda( sqlda );
            err:
                return( SQLCODE );
            }
```

```
void main( int argc, char *argv[] )
/*******************************/
/* Optional first argument is a select statement,
 * optional second argument is the fetch width */
{
    char *query_str =
     "select emp_fname, emp_lname from employee";
    unsigned fetch_width = 10;

    if( argc > 1 ) {
        query_str = argv[1];
        if( argc > 2 ) {
          fetch_width = atoi( argv[2] );
          if( fetch_width < 2 ) {
              fetch_width = 2;
          }
        }
    }
    db_init( &sqlca );
    EXEC SQL CONNECT "dba" IDENTIFIED BY "sql";

    DoQuery( query_str, fetch_width );

    EXEC SQL DISCONNECT;
err:
    db_fini( &sqlca );
}
```

Notes on using wide fetches

♦ In the function **PrepareSQLDA**, the SQLDA memory is allocated using the **alloc_sqlda** function. This allows space for indicator variables, rather than using the **alloc_sqlda_noind** function.

♦ If the number of rows fetched is fewer than the number requested, but is not zero (at the end of the cursor for example), the SQLDA items corresponding to the rows that were not fetched are returned as NULL by setting the indicator value. If no indicator variables are present, an error is generated (SQLE_NO_INDICATOR: no indicator variable for NULL result).

♦ If a row being fetched has been updated, generating a SQLE_ROW_UPDATED_WARNING warning, the fetch stops on the row that caused the warning. The values for all rows processed to that point (including the row that caused the warning) are returned. SQLCOUNT contains the number of rows that were fetched, including the row that caused the warning. All remaining SQLDA items are marked as NULL.

♦ If a row being fetched has been deleted or is locked, generating an SQLE_NO_CURRENT_ROW or SQLE_LOCKED error, SQLCOUNT contains the number of rows that were read prior to the error. This does

not include the row that caused the error. The SQLDA does not contain values for any of the rows since SQLDA values are not returned on errors. The SQLCOUNT value can be used to reposition the cursor, if necessary, to read the rows.

Static and dynamic SQL

There are two ways to embed SQL statements into a C program:

♦ Static statements

♦ Dynamic statements

Until now, we have been discussing static SQL. This section compares static and dynamic SQL.

Static SQL statements

All standard SQL data manipulation and data definition statements can be embedded in a C program by prefixing them with EXEC SQL and suffixing the command with a semicolon (;). These statements are referred to as **static** statements.

Static statements can contain references to host variables, as described in "Using host variables" on page 153. All examples to this point have used static embedded SQL statements.

Host variables can only be used in place of string or numeric constants. They cannot be used to substitute column names or table names; dynamic statements are required to perform those operations.

Dynamic SQL statements

In the C language, strings are stored in arrays of characters. Dynamic statements are constructed in C language strings. These statements can then be executed using the PREPARE and EXECUTE statements. These SQL statements cannot reference host variables in the same manner as static statements since the C language variables are not accessible by name when the C program is executing.

To pass information between the statements and the C language variables, a data structure called the **SQL Descriptor Area** (**SQLDA**) is used. This structure is set up for you by the SQL preprocessor if you specify a list of host variables on the EXECUTE command in the USING clause. These variables correspond by position to place holders in the appropriate positions of the prepared command string.

☞ For information on the SQLDA, see "The SQL descriptor area (SQLDA)" on page 181.

A **place holder** is put in the statement to indicate where host variables are to be accessed. A place holder is either a question mark (?) or a host variable reference as in static statements (a host variable name preceded by a colon).

In the latter case, the host variable name used in the actual text of the statement serves only as a place holder indicating a reference to the SQL descriptor area.

A host variable used to pass information to the database is called a **bind variable**.

Example

For example:

```
EXEC SQL BEGIN DECLARE SECTION;
   char comm[200];
   char address[30];
   char city[20];
   short int cityind;
   long empnum;
EXEC SQL END DECLARE SECTION;
. . .
   sprintf( comm, "update %s set address = :?,
                       city = :?"
          " where employee_number = :?",
          tablename );
EXEC SQL PREPARE S1 FROM :comm;
EXEC SQL EXECUTE S1 USING :address, :city:cityind, :empnum;
```

This method requires the programmer to know how many host variables there are in the statement. Usually, this is not the case. So, you can set up your own SQLDA structure and specify this SQLDA in the USING clause on the EXECUTE command.

The DESCRIBE BIND VARIABLES statement returns the host variable names of the bind variables that are found in a prepared statement. This makes it easier for a C program to manage the host variables. The general method is as follows:

```
EXEC SQL BEGIN DECLARE SECTION;
   char comm[200];
EXEC SQL END DECLARE SECTION;
. . .
sprintf( comm, "update %s set address = :address,
         city = :city"
         " where employee_number = :empnum",
         tablename );
EXEC SQL PREPARE S1 FROM :comm;
/* Assume that there are no more than 10 host variables. See
       next example if you can't put
a limit on it */
sqlda = alloc_sqlda( 10 );
EXEC SQL DESCRIBE BIND VARIABLES FOR S1 USING DESCRIPTOR sqlda;
/* sqlda->sqld will tell you how many host variables there were.
       */
/* Fill in SQLDA_VARIABLE fields with values based on
name fields in sqlda */
. . .
EXEC SQL EXECUTE S1 USING DESCRIPTOR sqlda;
free_sqlda( sqlda );
```

SQLDA contents

The SQLDA consists of an array of variable descriptors. Each descriptor describes the attributes of the corresponding C program variable or the location that the database stores data into or retrieves data from:

♦ data type

♦ length if **type** is a string type

♦ precision and scale if **type** is a numeric type

♦ memory address

♦ indicator variable

☞ For a complete description of the SQLDA structure, see "The SQL descriptor area (SQLDA)" on page 181

Indicator variables and NULL

The indicator variable is used to pass a NULL value to the database or retrieve a NULL value from the database. The indicator variable is also used by the database server to indicate truncation conditions encountered during a database operation. The indicator variable is set to a positive value when not enough space was provided to receive a database value.

☞ For more information, see "Indicator variables" on page 157.

Dynamic SELECT statement

A SELECT statement that returns only a single row can be prepared dynamically, followed by an EXECUTE with an INTO clause to retrieve the

one-row result. SELECT statements that return multiple rows, however, are managed using dynamic cursors.

With dynamic cursors, results are put into a host variable list or a SQLDA that is specified on the FETCH statement (FETCH INTO and FETCH USING DESCRIPTOR). Since the number of select list items is usually unknown to the C programmer, the SQLDA route is the most common. The DESCRIBE SELECT LIST statement sets up a SQLDA with the types of the select list items. Space is then allocated for the values using the **fill_sqlda()** function, and the information is retrieved by the FETCH USING DESCRIPTOR statement.

The typical scenario is as follows:

```
EXEC SQL BEGIN DECLARE SECTION;
   char comm[200];
EXEC SQL END DECLARE SECTION;
   int actual_size;
   SQLDA * sqlda;

. . .
sprintf( comm, "select * from %s", table_name );
EXEC SQL PREPARE S1 FROM :comm;
/* Initial guess of 10 columns in result. If it is
   wrong, it is corrected right after the first
   DESCRIBE by reallocating sqlda and doing DESCRIBE    again.
        */
sqlda = alloc_sqlda( 10 );
EXEC SQL DESCRIBE SELECT LIST FOR S1 USING DESCRIPTOR sqlda;
if( sqlda->sqld > sqlda->sqln ){
   actual_size = sqlda->sqld;
   free_sqlda( sqlda );
   sqlda = alloc_sqlda( actual_size );
   EXEC SQL DESCRIBE SELECT LIST FOR S1
      USING DESCRIPTOR sqlda;
}
fill_sqlda( sqlda );
EXEC SQL DECLARE C1 CURSOR FOR S1;
EXEC SQL OPEN C1;
EXEC SQL WHENEVER NOTFOUND {break};
for( ;; ){
   EXEC SQL FETCH C1 USING DESCRIPTOR sqlda;
   /* do something with data */
}
EXEC SQL CLOSE C1;
EXEC SQL DROP STATEMENT S1;
```

Drop statements after use

To avoid consuming unnecessary resources, ensure that statements are dropped after use.

☞ For a complete example using cursors for a dynamic select statement, see "Dynamic cursor sample" on page 146.

☞ For details of the functions mentioned above, see "Library function reference" on page 207.

The SQL descriptor area (SQLDA)

The SQLDA (SQL Descriptor Area) is an interface structure that is used for dynamic SQL statements. The structure passes information regarding host variables and SELECT statement results to and from the database. The SQLDA is defined in the header file *sqlda.h*.

☞ There are functions in the database interface library or DLL that you can use to manage SQLDAs. For descriptions, see "Library function reference" on page 207.

When host variables are used with static SQL statements, the preprocessor constructs a SQLDA for those host variables. It is this SQLDA that is actually passed to and from the database server.

The SQLDA header file

The contents of *sqlda.h* are as follows:

```
#ifndef _SQLDA_H_INCLUDED
#define _SQLDA_H_INCLUDED
#define II_SQLDA

#include "sqlca.h"

#if defined( _SQL_PACK_STRUCTURES )
#include "pshpk1.h"
#endif

#define SQL_MAX_NAME_LEN  30

#define _sqldafar
typedef   short int   a_SQL_type;
struct sqlname {
    short int length; /* length of char data */
    char      data[ SQL_MAX_NAME_LEN ]; /* data */
};
struct sqlvar {    /* array of variable descriptors   */
    short int  sqltype; /* type of  host variable    */
    short int   sqllen; /* length of host variable    */
    void      *sqldata; /* address of variable  */
    short int  *sqlind; /* indicator variable pointer */
    struct sqlname  sqlname;
};
```

```
struct sqlda{
    unsigned char  sqldaid[8]; /* eye catcher "SQLDA"*/
    a_SQL_int32  sqldabc; /* length of sqlda structure*/
    short int  sqln;
            /* descriptor size in number of entries */
    short int  sqld;
            /* number of variables found by DESCRIBE*/
    struct sqlvar      sqlvar[1];
            /* array of variable descriptors */
};
#define SCALE(sqllen)          ((sqllen)/256)
#define PRECISION(sqllen)      ((sqllen)&0xff)
#define SET_PRECISION_SCALE(sqllen,precision,scale)     \
                        sqllen = (scale)*256 + (precision)
#define DECIMALSTORAGE(sqllen)  (PRECISION(sqllen)/2 + 1)

typedef struct sqlda    SQLDA;
typedef struct sqlvar   SQLVAR, SQLDA_VARIABLE;
typedef struct sqlname  SQLNAME, SQLDA_NAME;

#ifndef SQLDASIZE
#define SQLDASIZE(n)    ( sizeof( struct sqlda ) + \
                        (n-1) * sizeof( struct sqlvar) )
#endif
#if defined( _SQL_PACK_STRUCTURES )
#include "poppk.h"
#endif
#endif
```

SQLDA fields

The SQLDA fields have the following meanings:

Field	Description
sqldaid	An 8-byte character field that contains the string **SQLDA** as an identification of the SQLDA structure. This field helps in debugging when you are looking at memory contents.
sqldabc	A long integer containing the length of the SQLDA structure.
sqln	The number of variable descriptors in the **sqlvar** array.
sqld	The number of variable descriptors which are valid (contain information describing a host variable). This field is set by the DESCRIBE statement and sometimes by the programmer when supplying data to the database server.
sqlvar	An array of descriptors of type **struct sqlvar,** each describing a host variable.

SQLDA host variable descriptions

Each **sqlvar** structure in the SQLDA describes a host variable. The fields of the **sqlvar** structure have the following meanings:

♦ **sqltype**   The type of the variable that is described by this descriptor (see "Embedded SQL data types" on page 149).

The low order bit indicates whether NULL values are allowed. Valid types and constant definitions can be found in the *sqldef.h* header file.

This field is filled by the DESCRIBE statement. You can set this field to any type when supplying data to the database server or retrieving data from the database server. Any necessary type conversion is done automatically.

♦ **sqllen**   The length of the variable. What the length actually means depends upon the type information and how the SQLDA is being used.

For DECIMAL types, this field is divided into two 1-byte fields. The high byte is the precision and the low byte is the scale. The precision is the total number of digits. The scale is the number of digits that appear after the decimal point.

For LONG VARCHAR and LONG BINARY data types, the **array_len** field of the DT_LONGBINARY and DT_LONGVARCHAR data type structure is used instead of the **sqllen** field.

☞ For more information on the length field, see "SQLDA sqllen field values" on page 184.

♦ **sqldata**   A four-byte pointer to the memory occupied by this variable. This memory must correspond to the **sqltype** and **sqllen** fields.

☞ For storage formats, see "Embedded SQL data types" on page 149.

For UPDATE and INSERT commands, this variable is not involved in the operation if the **sqldata** pointer is a null pointer. For a FETCH, no data is returned if the **sqldata** pointer is a null pointer. In other words, the column returned by the **sqldata** pointer is an **unbound column**.

If the DESCRIBE statement uses LONG NAMES, this field holds the long name of the result set column. If, in addition, the DESCRIBE statement is a DESCRIBE USER TYPES statement, then this field holds the long name of the user-defined data type, instead of the column. If the type is a base type, the field is empty.

♦ **sqlind**   A pointer to the indicator value. An indicator value is a **short int**. A negative indicator value indicates a NULL value. A positive indicator value indicates that this variable has been truncated by a

FETCH statement, and the indicator value contains the length of the data before truncation. A value of –2 indicates a conversion error if the CONVERSION_ERROR database option is set to OFF.

☞ For more information, see "Indicator variables" on page 157.

If the **sqlind** pointer is the null pointer, no indicator variable pertains to this host variable.

The **sqlind** field is also used by the DESCRIBE statement to indicate parameter types. If the type is a user-defined data type, this field is set to DT_HAS_USERTYPE_INFO. In such a case, you may wish to carry out a DESCRIBE USER TYPES to obtain information on the user-defined data types.

♦ **sqlname**   A VARCHAR-like structure, as follows:

```
struct sqlname {
    short int  length;
    char       data[ SQL_MAX_NAME_LEN ];
};
```

It is filled by a DESCRIBE statement and is not otherwise used. This field has a different meaning for the two formats of the DESCRIBE statement:

- **SELECT LIST**   The name buffer is filled with the column heading of the corresponding item in the select list.
- **BIND VARIABLES**   The name buffer is filled with the name of the host variable that was used as a bind variable, or "?" if an unnamed parameter marker is used.

On a DESCRIBE SELECT LIST command, any indicator variables present are filled with a flag indicating whether the select list item is updatable or not. More information on this flag can be found in the *sqldef.h* header file.

If the DESCRIBE statement is a DESCRIBE USER TYPES statement, then this field holds the long name of the user-defined data type instead of the column. If the type is a base type, the field is empty.

SQLDA sqllen field values

The **sqllen** field length of the **sqlvar** structure in a SQLDA is used in the following kinds of interactions with the database server:

♦ **describing values**   The DESCRIBE statement gets information about the host variables required to store data retrieved from the database, or host variables required to pass data to the database.

☞ See "Describing values" on page 185.

♦ **retrieving values**   Retrieving values from the database.

   ☞ See "Retrieving values" on page 187.

♦ **sending values**   Sending information to the database.

   ☞ See "Sending values" on page 186.

♦ These interactions are described in this section.

The following tables detail each of these interactions. These tables list the interface constant types (the **DT_** types) found in the *sqldef.h* header file. These constants would be placed in the SQLDA **sqltype** field.

☞ For information about **sqltype** field values, see "Embedded SQL data types" on page 149.

In static SQL, a SQLDA is still used but it is generated and completely filled in by the SQL preprocessor. In this static case, the tables give the correspondence between the static C language host variable types and the interface constants.

Describing values

The following table indicates the values of the **sqllen** and **sqltype** structure members returned by the DESCRIBE command for the various database types (both SELECT LIST and BIND VARIABLE DESCRIBE statements). In the case of a user-defined database data type, the base type is described.

Your program can use the types and lengths returned from a DESCRIBE, or you may use another type. The database server performs type conversions between any two types. The memory pointed to by the **sqldata** field must correspond to the **sqltype** and **sqllen** fields.

☞ For information on embedded SQL data types, see "Embedded SQL data types" on page 149.

Database field type	Embedded SQL type returned	Length returned on describe
BIGINT	DT_BIGINT	8
BINARY(n)	DT_BINARY	n
BIT	DT_BIT	1
CHAR(n)	DT_FIXCHAR	n
DATE	DT_DATE	length of longest for-matted string

Database field type	Embedded SQL type returned	Length returned on describe
DECIMAL(p,s)	DT_DECIMAL	high byte of length field in SQLDA set to p, and low byte set to s
DOUBLE	DT_DOUBLE	8
FLOAT	DT_FLOAT	4
INT	DT_INT	4
LONG BINARY	DT_LONGBINARY	32767
LONG VARCHAR	DT_LONGVARCHAR	32767
REAL	DT_FLOAT	4
SMALLINT	DT_SMALLINT	2
TIME	DT_TIME	length of longest formatted string
TIMESTAMP	DT_TIMESTAMP	length of longest formatted string
TINYINT	DT_TINYINT	1
UNSIGNED BIGINT	DT_UNSBIGINT	8
UNSIGNED INT	DT_UNSINT	4
UNSIGNED SMALL-INT	DT_UNSSMALLINT	2
VARCHAR(n)	DT_VARCHAR	n

Sending values

The following table indicates how you specify lengths of values when you supply data to the database server in the SQLDA.

Only the data types displayed in the table are allowed in this case. The DT_DATE, DT_TIME, and DT_TIMESTAMP types are treated the same as DT_STRING when supplying information to the database; the value must be a NULL-terminated character string in an appropriate date format.

Embedded SQL Data Type	Program action to set the length
DT_BIGINT	No action required

Embedded SQL Data Type	Program action to set the length
DT_BINARY(n)	Length taken from field in BINARY structure
DT_BIT	No action required
DT_DATE	Length determined by terminating \0
DT_DECIMAL(p,s)	high byte of length field in SQLDA set to p, and low byte set to s
DT_DOUBLE	No action required
DT_FIXCHAR(n)	Length field in SQLDA determines length of string
DT_FLOAT	No action required
DT_INT	No action required
DT_LONGBINARY	Length field ignored. See "Sending LONG data" on page 193
DT_LONGVARCHAR	Length field ignored. See "Sending LONG data" on page 193
DT_SMALLINT	No action required
DT_STRING	Length determined by terminating \0
DT_TIME	Length determined by terminating \0
DT_TIMESTAMP	Length determined by terminating \0
DT_TIMESTAMP_STRUCT	No action required
DT_UNSBIGINT	No action required
DT_UNSINT	No action required
DT_UNSSMALLINT	No action required
DT_VARCHAR(n)	Length taken from field in VARCHAR structure
DT_VARIABLE	Length determined by terminating \0

Retrieving values

The following table indicates the values of the length field when you retrieve data from the database using a SQLDA. The **sqllen** field is never modified

187

when you retrieve data.

Only the interface data types displayed in the table are allowed in this case. The DT_DATE, DT_TIME, and DT_TIMESTAMP data types are treated the same as DT_STRING when you retrieve information from the database. The value is formatted as a character string in the current date format.

Embedded SQL Data Type	What the program must set length field to when receiving	How the database returns length information after fetching a value
DT_BIGINT	No action required	No action required
DT_BINARY(n)	Maximum length of BINARY structure (n+2)	**len** field of BINARY structure set to actual length
DT_BIT	No action required	No action required
DT_DATE	Length of buffer	\0 at end of string
DT_DECIMAL(p,s)	High byte set to p and low byte set to s	No action required
DT_DOUBLE	No action required	No action required
DT_FIXCHAR(n)	Length of buffer	Padded with blanks to length of buffer
DT_FLOAT	No action required	No action required
DT_INT	No action required	No action required
DT_LONGBINARY	Length field ignored. See "Retrieving LONG data" on page 191	Length field ignored. See "Retrieving LONG data" on page 191
DT_LONGVARCHAR	Length field ignored. See "Retrieving LONG data" on page 191	Length field ignored. See "Retrieving LONG data" on page 191
DT_SMALLINT	No action required	No action required
DT_STRING	Length of buffer	\0 at end of string
DT_TIME	Length of buffer	\0 at end of string
DT_TIMESTAMP	Length of buffer	\0 at end of string
DT_TIMESTAMP_STRUCT	No action required	No action required

Embedded SQL Data Type	What the program must set length field to when receiving	How the database returns length information after fetching a value
DT_UNSBIGINT	No action required	No action required
DT_UNSINT	No action required	No action required
DT_UNSSMALLINT	No action required	No action required
DT_VARCHAR(n)	Maximum length of VARCHAR structure (n+2)	**len** field of VARCHAR structure set to actual length

Sending and retrieving long values

The method for sending and retrieving LONG VARCHAR and LONG BINARY values in embedded SQL applications is different from that for other data types. Although the standard SQLDA fields can be used, they are limited to 32 kb data as the fields holding the information (sqldata, sqllen, sqlind) are 16-bit values. Changing these values to 32-bit values would break existing applications.

The method of describing LONG VARCHAR and LONG BINARY values is the same as for other data types.

☞ For information about how to retrieve and send values, see "Retrieving LONG data" on page 191, and "Sending LONG data" on page 193.

Static SQL usage   Separate structures are used to hold the allocated, stored, and untruncated lengths of LONG BINARY and LONG VARCHAR data types. The static SQL data types are defined in *sqlca.h* as follows:

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

Dynamic SQL usage   For dynamic SQL, set the **sqltype** field to DT_LONGVARCHAR or DT_LONGBINARY as appropriate. The associated LONGBINARY and LONGVARCHAR structures are as follows:

```
typedef struct LONGVARCHAR {
    a_sql_uint32    array_len;
          /* number of allocated bytes in array */
    a_sql_uint32    stored_len;
          /* number of bytes stored in array
           * (never larger than array_len)
           */
    a_sql_uint32    untrunc_len;
          /* number of bytes in untruncated expression
           * (may be larger than array_len)
           */
    char            array[1];   /* the data */
} LONGVARCHAR, LONGBINARY;
```

☞ For information about how to implement this feature in your

applications, see "Retrieving LONG data" on page 191, and "Sending LONG data" on page 193.

Retrieving LONG data

This section describes how to retrieve LONG values from the database. For background information, see "Sending and retrieving long values" on page 190.

The procedures are different depending on whether you are using static or dynamic SQL.

❖ **To receive a LONG VARCHAR or LONG BINARY value (static SQL)**

1. Declare a host variable of type DECL_LONGVARCHAR or DECL_LONGBINARY, as appropriate.

2. Retrieve the data using FETCH, GET DATA, or EXECUTE INTO. Adaptive Server Anywhere sets the following information:

   ♦ **indicator variable** The indicator variable is negative if the value is NULL, 0 if there is no truncation, and is the positive untruncated length in bytes up to a maximum of 32767.

   ☞ For more information, see "Indicator variables" on page 157.

   ♦ **stored_len** This DECL_LONGVARCHAR or DECL_LONGBINARY field holds the number of bytes retrieved into the array. It is never greater than **array_len**.

   ♦ **untrunc_len** This DECL_LONGVARCHAR or DECL_LONGBINARY field holds the number of bytes held by the database server. It is at least equal to the **stored_len** value. It is set even if the value is not truncated.

❖ **To receive a value into a LONGVARCHAR or LONGBINARY structure (dynamic SQL)**

1. Set the **sqltype** field to DT_LONGVARCHAR or DT_LONGBINARY as appropriate.

2. Set the **sqldata** field to point to the LONGVARCHAR or LONGBINARY structure.

   You can use the LONGVARCHARSIZE(n) or LONGBINARYSIZE(n) macros to determine the total number of bytes to allocate to hold n bytes of data in the array field.

3. Set the **array_len** field of the LONGVARCHAR or LONGBINARY structure to the number of bytes allocated for the array field.

4. Retrieve the data using FETCH, GET DATA, or EXECUTE INTO. Adaptive Server Anywhere sets the following information:

♦ *** sqlind**   This **sqlda** field is negative if the value is NULL, 0 if there is no truncation, and is the positive untruncated length in bytes up to a maximum of 32767.

♦ **stored_len**   This LONGVARCHAR or LONGBINARY field holds the number of bytes retrieved into the array. It is never greater than **array_len**.

♦ **untrunc_len**   This LONGVARCHAR or LONGBINARY field holds the number of bytes held by the database server. It is at least equal to the stored_len value. It is set even if the value is not truncated.

The following code snippet illustrates the mechanics of retrieving LONG VARCHAR data using dynamic embedded SQL. It is not intended to be a practical application:

```
#define DATA_LEN 128000
void get_test_var()
/*****************/
{
    LONGVARCHAR *longptr;
    SQLDA *sqlda;
    SQLVAR *sqlvar;

    sqlda = alloc_sqlda( 1 );
    longptr = (LONGVARCHAR *)malloc(
                 LONGVARCHARSIZE( DATA_LEN ) );
    if( sqlda == NULL || longptr == NULL ) {
        fatal_error( "Allocation failed" );
    }

    // init longptr for receiving data
    longptr->array_len = DATA_LEN;

    // init sqlda for receiving data
    // (sqllen is unused with DT_LONG types)
    sqlda->sqld = 1;   // using 1 sqlvar
    sqlvar = &sqlda->sqlvar[0];
    sqlvar->sqltype = DT_LONGVARCHAR;
    sqlvar->sqldata = longptr;

    printf( "fetching test_var\n" );
    EXEC SQL PREPARE select_stmt FROM 'SELECT test_var';
    EXEC SQL EXECUTE select_stmt INTO DESCRIPTOR sqlda;
    EXEC SQL DROP STATEMENT select_stmt;
    printf( "stored_len: %d, untrunc_len: %d,
       1st char: %c, last char: %c\n",
       longptr->stored_len,
       longptr->untrunc_len,
       longptr->array[0],
       longptr->array[DATA_LEN-1] );
    free_sqlda( sqlda );
    free( longptr );
}
```

Sending LONG data

This section describes how to send LONG values to the database from embedded SQL applications. For background information, see "Sending and retrieving long values" on page 190.

The procedures are different depending on whether you are using static or dynamic SQL.

❖ **To send a LONG VARCHAR or LONG BINARY value (static SQL)**

1. Declare a host variable of type DECL_LONGVARCHAR or DECL_LONGBINARY, as appropriate.

2. If you are sending NULL and using an indicator variable, set the indicator variable to a negative value.

   ☞ For more information, see "Indicator variables" on page 157.

3. Set the **stored_len** field of the DECL_LONGVARCHAR or DECL_LONGBINARY structure to the number of bytes of data in the array field.

4. Send the data by opening the cursor or executing the statement.

The following code snippet illustrates the mechanics of sending a LONG VARCHAR using static embedded SQL. It is not intended to be a practical application.

```
#define DATA_LEN 12800
EXEC SQL BEGIN DECLARE SECTION;
// SQLPP initializes longdata.array_len
DECL_LONGVARCHAR(128000) longdata;
EXEC SQL END DECLARE SECTION;

void set_test_var()
/*****************/
{
    // init longdata for sending data
    memset( longdata.array, 'a', DATA_LEN );
    longdata.stored_len = DATA_LEN;

    printf( "Setting test_var to %d a's\n", DATA_LEN );
    EXEC SQL SET test_var = :longdata;
}
```

❖ **To send a value using a LONGVARCHAR or LONGBINARY structure (dynamic SQL)**

1. Set the sqltype field to DT_LONGVARCHAR or DT_LONGBINARY as appropriate.

2. If you are sending NULL, set *** sqlind** to a negative value.

3. Set the **sqldata** field to point to the LONGVARCHAR or LONGBINARY structure.

   You can use the LONGVARCHARSIZE(n) or LONGBINARYSIZE(n) macros to determine the total number of bytes to allocate to hold n bytes of data in the array field.

4. Set the **array_len** field of the LONGVARCHAR or LONGBINARY structure to the number of bytes allocated for the array field.

5. Set the **stored_len** field of the LONGVARCHAR or LONGBINARY structure to the number of bytes of data in the array field. This must not be more than **array_len**.

6. Send the data by opening the cursor or executing the statement.

Using stored procedures

This section describes the use of SQL procedures in embedded SQL.

Using simple stored procedures

You can create and call stored procedures in embedded SQL.

You can embed a CREATE PROCEDURE just like any other data definition statement, such as CREATE TABLE. You can also embed a CALL statement to execute a stored procedure. The following code fragment illustrates both creating and executing a stored procedure in embedded SQL:

```
EXEC SQL CREATE PROCEDURE pettycash( IN amount
   DECIMAL(10,2) )
BEGIN
   UPDATE account
   SET balance = balance - amount
   WHERE name = 'bank';

   UPDATE account
   SET balance = balance + amount
   WHERE name = 'pettycash expense';
END;
EXEC SQL CALL pettycash( 10.72 );
```

If you wish to pass host variable values to a stored procedure or to retrieve the output variables, you prepare and execute a CALL statement. The following code fragment illustrates the use of host variables. Both the USING and INTO clauses are used on the EXECUTE statement.

```
EXEC SQL BEGIN DECLARE SECTION;
   double  hv_expense;
   double  hv_balance;
EXEC SQL END DECLARE SECTION;

// code here
EXEC SQL CREATE PROCEDURE pettycash(
        IN expense    DECIMAL(10,2),
        OUT endbalance DECIMAL(10,2) )
   BEGIN
      UPDATE account
      SET balance = balance - expense
      WHERE name = 'bank';

      UPDATE account
      SET balance = balance + expense
      WHERE name = 'pettycash expense';

      SET endbalance = ( SELECT balance FROM account
                           WHERE name = 'bank' );
   END;

EXEC SQL PREPARE S1 FROM 'CALL pettycash( ?, ? )';
EXEC SQL EXECUTE S1 USING :hv_expense INTO :hv_balance;
```

☞ For more information, see "EXECUTE statement [ESQL]" [*ASA SQL Reference,* page 471], and "PREPARE statement [ESQL]" [*ASA SQL Reference,* page 563].

Stored procedures with result sets

Database procedures can also contain SELECT statements. The procedure is declared using a RESULT clause to specify the number, name, and types of the columns in the result set. Result set columns are different from output parameters. For procedures with result sets, the CALL statement can be used in place of a SELECT statement in the cursor declaration:

```
EXEC SQL BEGIN DECLARE SECTION;
    char   hv_name[100];
EXEC SQL END DECLARE SECTION;

EXEC SQL CREATE PROCEDURE female_employees()
   RESULT( name char(50) )
   BEGIN
      SELECT emp_fname || emp_lname FROM employee
      WHERE sex = 'f';
   END;

EXEC SQL PREPARE S1 FROM 'CALL female_employees()';

EXEC SQL DECLARE C1 CURSOR FOR S1;
EXEC SQL OPEN C1;
for(;;) {
   EXEC SQL FETCH C1 INTO :hv_name;
   if( SQLCODE != SQLE_NOERROR ) break;
   printf( "%s\\n", hv_name );
}
EXEC SQL CLOSE C1;
```

In this example, the procedure has been invoked with an OPEN statement
rather than an EXECUTE statement. The OPEN statement causes the
procedure to execute until it reaches a SELECT statement. At this point, C1
is a cursor for the SELECT statement within the database procedure. You
can use all forms of the FETCH command (backward and forward scrolling)
until you are finished with it. The CLOSE statement terminates execution of
the procedure.

If there had been another statement following the SELECT in the procedure,
it would not have been executed. In order to execute statements following a
SELECT, use the RESUME cursor-name command. The RESUME
command either returns the warning SQLE_PROCEDURE_COMPLETE or
it returns SQLE_NOERROR indicating that there is another cursor. The
example illustrates a two-select procedure:

```
EXEC SQL CREATE PROCEDURE people()
RESULT( name char(50) )
BEGIN

   SELECT emp_fname || emp_lname
   FROM employee;

   SELECT fname || lname
   FROM customer;
END;

EXEC SQL PREPARE S1 FROM 'CALL people()';

EXEC SQL DECLARE C1 CURSOR FOR S1;
EXEC SQL OPEN C1;
while( SQLCODE == SQLE_NOERROR ) {
   for(;;) {
      EXEC SQL FETCH C1 INTO :hv_name;
      if( SQLCODE != SQLE_NOERROR ) break;
      printf( "%s\\n", hv_name );
   }
   EXEC SQL RESUME C1;
}
EXEC SQL CLOSE C1;
```

Dynamic cursors for CALL statements

These examples have used static cursors. Full dynamic cursors can also be used for the CALL statement.

☞ For a description of dynamic cursors, see "Dynamic SELECT statement" on page 178.

The DESCRIBE statement works fully for procedure calls. A DESCRIBE OUTPUT produces a SQLDA that has a description for each of the result set columns.

If the procedure does not have a result set, the SQLDA has a description for each INOUT or OUT parameter for the procedure. A DESCRIBE INPUT statement produces a SQLDA having a description for each IN or INOUT parameter for the procedure.

DESCRIBE ALL

DESCRIBE ALL describes IN, INOUT, OUT, and RESULT set parameters. DESCRIBE ALL uses the indicator variables in the SQLDA to provide additional information.

The DT_PROCEDURE_IN and DT_PROCEDURE_OUT bits are set in the indicator variable when a CALL statement is described. DT_PROCEDURE_IN indicates an IN or INOUT parameter and DT_PROCEDURE_OUT indicates an INOUT or OUT parameter. Procedure RESULT columns have both bits clear.

After a describe OUTPUT, these bits can be used to distinguish between

statements that have result sets (need to use OPEN, FETCH, RESUME, CLOSE) and statements that do not (need to use EXECUTE).

☞ For a complete description, see "DESCRIBE statement [ESQL]" [*ASA SQL Reference,* page 449].

Multiple result sets   If you have a procedure that returns multiple result sets, you must re-describe after each RESUME statement if the result sets change shapes.

You need to describe the cursor, not the statement, to re-describe the current position of the cursor.

Embedded SQL programming techniques

This section contains a set of tips for developers of embedded SQL programs.

Implementing request management

The default behavior of the interface DLL is for applications to wait for completion of each database request before carrying out other functions. This behavior can be changed using request management functions. For example, when using Interactive SQL, the operating system is still active while Interactive SQL is waiting for a response from the database and Interactive SQL carries out some tasks in that time.

You can achieve application activity while a database request is in progress by providing a **callback function**. In this callback function you must not do another database request except **db_cancel_request**. You can use the **db_is_working** function in your message handlers to determine if you have a database request in progress.

The **db_register_a_callback** function is used to register your application callback functions.

☞ For more information, see the following:

♦ "db_register_a_callback function" on page 215

♦ "db_cancel_request function" on page 210

♦ "db_is_working function" on page 213

Backup functions

The **db_backup** function provides support for online backup in embedded SQL applications. The backup utility makes use of this function. You should only need to write a program to use this function if your backup requirements are not satisfied by the Adaptive Server Anywhere backup utility.

BACKUP statement is recommended
Although this function provides one way to add backup features to an application, the recommended way to accomplish this task is to use the BACKUP statement. For more information, see "BACKUP statement" [*ASA SQL Reference,* page 307].

☞ You can also access the backup utility directly using the Database Tools DBBackup function. For more information on this function, see "DBBackup

function" on page 269.

☞ For more information, see "db_backup function" on page 207.

The SQL preprocessor

The SQL preprocessor processes a C or C++ program containing embedded SQL, before the compiler is run.

Syntax **sqlpp** [*options*] *input-file* [*output-file*]

Option	Description
–c *"key-word=value;..."*	Supply reference database connection parameters [UltraLite]
–d	Favor data size
–e *level*	Flag non-conforming SQL syntax as an error
–f	Put the far keyword on generated static data
–g	Do not display UltraLite warnings
–h *line–width*	Limit the maximum line length of output
–k	Include user declaration of SQLCODE
–m *version*	Specify the version name for generated synchronization scripts
–n	Line numbers
–o *operating–sys*	Target operating system.
–p *project*	UltraLite project name
–q	Quiet mode—do not print banner
–r	Generate reentrant code
–s *string–len*	Maximum string length for the compiler
–w *level*	Flag non-conforming SQL syntax as a warning
–x	Change multibyte SQL strings to escape sequences
–z *sequence*	Specify collation sequence

See also

Description The SQL preprocessor processes a C or C++ program containing embedded SQL before the compiler is run. SQLPP translates the SQL statements in the *input-file* into C language source that is put into the *output-file*. The normal extension for source programs with embedded SQL is *.sqc*. The default output filename is the *input-file* with an extension of *.c*. If *input-file* has a *.c*

extension, the default output filename extension is *.cc*.

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

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

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

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

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

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

♦ **f** flag syntax that is not full-SQL/92 syntax

♦ **t** flag non-standard host variable types

♦ **u** flag syntax that is not supported by UltraLite

♦ **w** allow all supported syntax

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

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

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

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

–n Generate line number information in the C file. This consists of *#line* directives in the appropriate places in the generated C code. If the compiler that you are using supports the *#line* directive, this option makes the compiler report errors on line numbers in the SQC file (the one with the embedded SQL) as opposed to reporting errors on line numbers in the C file generated by the SQL preprocessor. Also, the *#line* directives are used indirectly by the source level debugger so that you can debug while viewing the SQC source file.

–o Specify the target operating system. Note that this option must match the operating system where you run the program. A reference to a special symbol is generated in your program. This symbol is defined in the interface

library. If you use the wrong operating system specification or the wrong
library, an error is detected by the linker. The supported operating systems
are:

♦ **WINDOWS** Windows 95/98/Me, Windows CE

♦ **WINNT** Microsoft Windows NT/2000/XP

♦ **NETWARE** Novell NetWare

♦ **UNIX** UNIX

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

–q Do not print the banner.

–r For more information on re-entrant code, see "SQLCA management for
multi-threaded or reentrant code" on page 163.

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

–w This option flags any embedded SQL that is not part of a specified set
of SQL/92 as a warning.

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

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

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

♦ **f** flag syntax that is not full-SQL/92 syntax

♦ **t** flag non-standard host variable types

♦ **u** flag syntax that is not supported by UltraLite

♦ **w** allow all supported syntax

–x Change multibyte strings to escape sequences so that they can pass
through compilers.

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

The collation sequence is used to help the preprocessor understand the
characters used in the source code of the program, for example, in

identifying alphabetic characters suitable for use in identifiers. If -z is not specified, the preprocessor attempts to determine a reasonable collation to use based on the operating system and SQLLOCALE environment variable.

Library function reference

The SQL preprocessor generates calls to functions in the interface library or DLL. In addition to the calls generated by the SQL preprocessor, a set of library functions is provided to make database operations easier to perform. Prototypes for these functions are included by the EXEC SQL INCLUDE SQLCA command.

This section contains a reference description of these various functions.

DLL entry points   The DLL entry points are the same except that the prototypes have a modifier appropriate for DLLs.

You can declare the entry points in a portable manner using **_esqlentry_**, which is defined in *sqlca.h.* It resolves to the value __stdcall:

alloc_sqlda function

Prototype   SQLDA ***alloc_sqlda(** unsigned *numvar* **);**

Description   Allocates a SQLDA with descriptors for *numvar* variables. The **sqln** field of the SQLDA is initialized to *numvar.* Space is allocated for the indicator variables, the indicator pointers are set to point to this space, and the indicator value is initialized to zero. A null pointer is returned if memory cannot be allocated. It is recommended that you use this function instead of **alloc_sqlda_noind function**.

alloc_sqlda_noind function

Prototype   SQLDA ***alloc_sqlda_noind(** unsigned *numvar* **);**

Description   Allocates a SQLDA with descriptors for *numvar* variables. The **sqln** field of the SQLDA is initialized to *numvar.* Space is not allocated for indicator variables; the indicator pointers are set to the null pointer. A null pointer is returned if memory cannot be allocated.

db_backup function

Prototype   void **db_backup(**
SQLCA * *sqlca***,**
int *op***,**
*int file_num***,**
unsigned long *page_num***,**
SQLDA * *sqlda* **);**

Authorization   Must be connected to a user ID with DBA authority or REMOTE DBA authority (SQL Remote).

> **BACKUP statement is recommended**
> Although this function provides one way to add backup features to an application, the recommended way to accomplish this task is to use the BACKUP statement. For more information, see "BACKUP statement" [*ASA SQL Reference,* page 307].

The action performed depends on the value of the *op* parameter:

♦ **DB_BACKUP_START**   Must be called before a backup can start. Only one backup can be running at one time against any given database server. Database checkpoints are disabled until the backup is complete (**db_backup** is called with an *op* value of DB_BACKUP_END). If the backup cannot start, the SQLCODE is SQLE_BACKUP_NOT_STARTED. Otherwise, the SQLCOUNT field of the *sqlca* is set to the size of each database page. (Backups are processed one page at a time.)

The *file_num*, *page_num* and *sqlda* parameters are ignored.

♦ **DB_BACKUP_OPEN_FILE**   Open the database file specified by *file_num*, which allows pages of the specified file to be backed up using DB_BACKUP_READ_PAGE. Valid file numbers are 0 through DB_BACKUP_MAX_FILE for the root database files, DB_BACKUP_TRANS_LOG_FILE for the transaction log file, and DB_BACKUP_WRITE_FILE for the database write file if it exists. If the specified file does not exist, the SQLCODE is SQLE_NOTFOUND. Otherwise, SQLCOUNT contains the number of pages in the file, SQLIOESTIMATE contains a 32-bit value (POSIX **time_t**) which identifies the time that the database file was created, and the operating system file name is in the *sqlerrmc* field of the SQLCA.

The *page_num* and *sqlda* parameters are ignored.

♦ **DB_BACKUP_READ_PAGE**   Read one page of the database file specified by *file_num*. The *page_num* should be a value from 0 to one less than the number of pages returned in SQLCOUNT by a successful call to **db_backup** with the DB_BACKUP_OPEN_FILE operation. Otherwise, SQLCODE is set to SQLE_NOTFOUND. The *sqlda* descriptor should be set up with one variable of type DT_BINARY pointing to a buffer. The buffer should be large enough to hold binary data of the size returned in the SQLCOUNT field on the call to **db_backup** with the DB_BACKUP_START operation.

DT_BINARY data contains a two-byte length followed by the actual binary data, so the buffer must be two bytes longer than the page size.

> **Application must save buffer**
> This call makes a copy of the specified database page into the buffer,
> but it is up to the application to save the buffer on some backup media.

♦ **DB_BACKUP_READ_RENAME_LOG** This action is the same as
DB_BACKUP_READ_PAGE, except that after the last page of the
transaction log has been returned, the database server renames the
transaction log and starts a new one.

If the database server is unable to rename the log at the current time (for
example in version 7.x or earlier databases there may be incomplete
transactions), the SQLE_BACKUP_CANNOT_RENAME_LOG_YET
error is set. In this case, do not use the page returned, but instead reissue
the request until you receive SQLE_NOERROR and then write the page.
Continue reading the pages until you receive the SQLE_NOTFOUND
condition.

The SQLE_BACKUP_CANNOT_RENAME_LOG_YET error may be
returned multiple times and on multiple pages. In your retry loop, you
should add a delay so as not to slow the server down with too many
requests.

When you receive the SQLE_NOTFOUND condition, the transaction log
has been backed up successfully and the file has been renamed. The name
for the old transaction file is returned in the *sqlerrmc* field of the SQLCA.

You should check the **sqlda->sqlvar[0].sqlind** value after a **db_backup**
call. If this value is greater than zero, the last log page has been written
and the log file has been renamed. The new name is still in
sqlca.sqlerrmc, but the SQLCODE value is SQLE_NOERROR.

You should not call **db_backup** again after this, except to close files and
finish the backup. If you do, you get a second copy of your backed up log
file and you receive SQLE_NOTFOUND.

♦ **DB_BACKUP_CLOSE_FILE** Must be called when processing of one
file is complete to close the database file specified by *file_num*.

The *page_num* and *sqlda* parameters are ignored.

♦ **DB_BACKUP_END** Must be called at the end of the backup. No other
backup can start until this backup has ended. Checkpoints are enabled
again.

The *file_num*, *page_num* and *sqlda* parameters are ignored.

The *dbbackup* program uses the following algorithm. Note that this is *not* C
code, and does not include error checking.

```
db_backup( ... DB_BACKUP_START ... )
allocate page buffer based on page size in SQLCODE
sqlda = alloc_sqlda( 1 )
sqlda->sqld = 1;
sqlda->sqlvar[0].sqltype = DT_BINARY
sqlda->sqlvar[0].sqldata = allocated buffer
for file_num = 0 to DB_BACKUP_MAX_FILE
  db_backup( ... DB_BACKUP_OPEN_FILE, file_num ... )
  if SQLCODE == SQLE_NO_ERROR
    /* The file exists */
    num_pages = SQLCOUNT
    file_time = SQLE_IO_ESTIMATE
    open backup file with name from sqlca.sqlerrmc
    for page_num = 0 to num_pages - 1
      db_backup( ... DB_BACKUP_READ_PAGE,
                 file_num, page_num, sqlda )
    write page buffer out to backup file
    next page_num
    close backup file
    db_backup( ... DB_BACKUP_CLOSE_FILE, file_num ... )
  end if
next file_num
backup up file DB_BACKUP_WRITE_FILE as above
backup up file DB_BACKUP_TRANS_LOG_FILE as above
free page buffer
db_backup( ... DB_BACKUP_END ... )
```

db_cancel_request function

Prototype   int **db_cancel_request(** SQLCA *_sqlca_ **);**

Description   Cancels the currently active database server request. This function checks to
                     make sure a database server request is active before sending the cancel
                     request. If the function returns 1, then the cancel request was sent; if it
                     returns 0, then no request was sent.

                     A non-zero return value does not mean that the request was canceled. There
                     are a few critical timing cases where the cancel request and the response
                     from the database or server "cross". In these cases, the cancel simply has no
                     effect, even though the function still returns TRUE.

                     The **db_cancel_request** function can be called asynchronously. This
                     function and **db_is_working** are the only functions in the database interface
                     library that can be called asynchronously using an SQLCA that might be in
                     use by another request.

                     If you cancel a request that is carrying out a cursor operation, the position of
                     the cursor is indeterminate. You must locate the cursor by its absolute
                     position or close it, following the cancel.

db_delete_file function

Prototype	void **db_delete_file(** SQLCA * *sqlca***,** char * filename **);**
Authorization	Must be connected to a user ID with DBA authority or REMOTE DBA authority (SQL Remote).
Description	The **db_delete_file** function requests the database server to delete *filename*. This can be used after backing up and renaming the transaction log (see DB_BACKUP_READ_RENAME_LOG in "db_backup function" on page 207) to delete the old transaction log. You must be connected to a user ID with DBA authority.

db_find_engine function

Prototype	unsigned short **db_find_engine(** SQLCA * *sqlca***,** char * *name* **);**
Description	Returns an unsigned short value, which indicates status information about the database server whose name is *name*. If no server can be found with the specified name, the return value is 0. A non-zero value indicates that the server is currently running.
	Each bit in the return value conveys some information. Constants that represent the bits for the various pieces of information are defined in the *sqldef.h* header file. If a null pointer is specified for *name*, information is returned about the default database environment.

db_fini function

Prototype	unsigned short **db_fini(** SQLCA * *sqlca* **);**
Description	This function frees resources used by the database interface or DLL. You must not make any other library calls or execute any embedded SQL commands after **db_fini** is called. If an error occurs during processing, the error code is set in SQLCA and the function returns 0. If there are no errors, a non-zero value is returned.
	You need to call **db_fini** once for each SQLCA being used.

> **Caution**
> *Failure to call **db_fini** for each **db_init** on NetWare can cause the database server to fail and the NetWare file server to fail.*

db_get_property function

Prototype

unsigned int **db_get_property(**
SQLCA * *sqlca***,**
a_db_property *property***,**
char * *value_buffer***,**
int *value_buffer_size* **);**

Description

This function is used to obtain the address of the server to which you are currently connected. It is used by the *dbping* utility to print out the server address.

The function can also be used to obtain the value of database properties. Database properties can also be obtained in an interface-independent manner by executing a SELECT statement, as described in "Database properties" [*ASA Database Administration Guide,* page 713].

The arguments are as follows:

- ♦ **a_db_property**   An **enum** with the value DB_PROP_SERVER_ADDRESS. DB_PROP_SERVER_ADDRESS gets the current connection's server network address as a printable string. Shared memory and NamedPipes protocols always return the empty string for the address. TCP/IP and SPX protocols return non-empty string addresses.

- ♦ **value_buffer**   This argument is filled with the property value as a null terminated string.

- ♦ **value_buffer_size**   The maximum length of the string **value_buffer**, including the terminating null character.

See also

"Database properties" [*ASA Database Administration Guide,* page 713]

db_init function

Prototype

unsigned short **db_init(** SQLCA *sqlca* **);**

Description

This function initializes the database interface library. This function must be called before any other library call is made and before any embedded SQL command is executed. The resources the interface library requires for your program are allocated and initialized on this call.

Use **db_fini** to free the resources at the end of your program. If there are any errors during processing, they are returned in the SQLCA and 0 is returned.

If there are no errors, a non-zero value is returned and you can begin using embedded SQL commands and functions.

In most cases, this function should be called only once (passing the address of the global **sqlca** variable defined in the *sqlca.h* header file). If you are writing a DLL or an application that has multiple threads using embedded SQL, call **db_init** once for each SQLCA that is being used.

☞ For more information, see "SQLCA management for multi-threaded or reentrant code" on page 163.

Caution

*Failure to call **db_fini** for each **db_init** on NetWare can cause the database server to fail, and the NetWare file server to fail.*

See also

For information on using db_init in UltraLite applications, see "db_init function" [*UltraLite C/C++ User's Guide,* page 360].

db_is_working function

Prototype

unsigned **db_is_working(** SQLCA *sqlca **);**

Description

Returns 1 if your application has a database request in progress that uses the given sqlca and 0 if there is no request in progress that uses the given sqlca.

This function can be called asynchronously. This function and **db_cancel_request** are the only functions in the database interface library that can be called asynchronously using an SQLCA that might be in use by another request.

db_locate_servers function

Prototype

unsigned int **db_locate_servers(**
SQLCA *sqlca**,**
SQL_CALLBACK_PARM *callback_address**,**
void *callback_user_data **);**

Description

Provides programmatic access to the information displayed by the *dblocate* utility, listing all the Adaptive Server Anywhere database servers on the local network that are listening on TCP/IP.

The callback function must have the following prototype:

int (*)(SQLCA *sqlca,
a_server_address *server_addr,
void *callback_user_data);

The callback function is called for each server found. If the callback

213

function returns 0, **db_locate_servers** stops iterating through servers.

The **sqlca** and **callback_user_data** passed to the callback function are those passed into **db_locate_servers**. The second parameter is a pointer to an **a_server_address** structure. **a_server_address** is defined in *sqlca.h*, with the following definition:

```
typedef struct a_server_address {
    a_SQL_uint32    port_type;
    a_SQL_uint32    port_num;
    char         *name;
    char         *address;
} a_server_address;
```

♦ **port_type** Is always PORT_TYPE_TCP at this time (defined to be 6 in *sqlca.h*).

♦ **port_num** Is the TCP port number on which this server is listening.

♦ **name** Points to a buffer containing the server name.

♦ **address** Points to a buffer containing the IP address of the server.

☞ For more information, see "The Server Location utility" [*ASA Database Administration Guide,* page 567].

db_locate_servers_ex function

Prototype	unsigned int **db_locate_servers_ex(** SQLCA *_sqlca_**,** SQL_CALLBACK_PARM _callback_address_**,,** void *_callback_user_data_ unsigned int _bitmask_**);**
Description	Provides programmatic access to the information displayed by the *dblocate* utility, listing all the Adaptive Server Anywhere database servers on the local network that are listening on TCP/IP, but provides a mask parameter used to select addresses passed to the callback routine.

The callback function must have the following prototype:

int (*)(SQLCA *_sqlca_,
a_server_address *_server_addr_,
void *_callback_user_data_);

The callback function is called for each server found that satisfies the bit mask. If the callback function returns 0, **db_locate_servers_ex** stops iterating through servers.

The **sqlca** and **callback_user_data** passed to the callback function are those passed into **db_locate_servers**. The second parameter is a pointer to an

a_server_address structure. **a_server_address** is defined in *sqlca.h*, with the following definition:

```
typedef struct a_server_address {
    a_SQL_uint32  port_type;
    a_SQL_uint32  port_num;
    char        *name;
    char        *address;
} a_server_address;
```

♦ **port_type**   Is always PORT_TYPE_TCP at this time (defined to be 6 in *sqlca.h*).

♦ **port_num**   Is the TCP port number on which this server is listening.

♦ **name**   Points to a buffer containing the server name.

♦ **address**   Points to a buffer containing the IP address of the server.

Only one bit mask flag is currently supported, DB_LOOKUP_FLAG_NUMERIC. This flag is defined in sqlca.h. It ensures that the addresses passed to the callback routine are IP addresses, instead of host names.

☞ For more information, see "The Server Location utility" [*ASA Database Administration Guide,* page 567].

db_register_a_callback function

Prototype

void **db_register_a_callback(**
SQLCA *sqlca**,**
a_db_callback_index *index***,**
(SQL_CALLBACK_PARM) *callback* **);**

Description

This function registers callback functions.

If you do not register a DB_CALLBACK_WAIT callback, the default action is to do nothing. Your application blocks, waiting for the database response, and Windows changes the cursor to an hourglass.

To remove a callback, pass a null pointer as the *callback* function.

The following values are allowed for the *index* parameter:

♦ **DB_CALLBACK_DEBUG_MESSAGE**   The supplied function is called once for each debug message and is passed a null-terminated string containing the text of the debug message. The string normally has a newline character (\n) immediately before the terminating null character. The prototype of the callback function is as follows:

```
void SQL_CALLBACK debug_message_callback(
SQLCA *sqlca,
char * message_string );
```

♦ **DB_CALLBACK_START**   The prototype is as follows:

```
void SQL_CALLBACK start_callback( SQLCA *sqlca );
```

This function is called just before a database request is sent to the server.
DB_CALLBACK_START is used only on Windows.

♦ **DB_CALLBACK_FINISH**   The prototype is as follows:

```
void SQL_CALLBACK finish_callback( SQLCA * sqlca );
```

This function is called after the response to a database request has been
received by the interface DLL. DB_CALLBACK_FINISH is used only
on Windows operating systems.

♦ **DB_CALLBACK_CONN_DROPPED**   The prototype is as follows:

```
void SQL_CALLBACK conn_dropped_callback (
SQLCA *sqlca,
char *conn_name );
```

This function is called when the database server is about to drop a
connection because of a liveness timeout, through a DROP
CONNECTION statement, or because the database server is being shut
down. The connection name **conn_name** is passed in to allow you to
distinguish between connections. If the connection was not named, it has
a value of NULL.

♦ **DB_CALLBACK_WAIT**   The prototype is as follows:

```
void SQL_CALLBACK wait_callback( SQLCA *sqlca );
```

This function is called repeatedly by the interface library while the
database server or client library is busy processing your database request.

You would register this callback as follows:

```
db_register_a_callback( &sqlca,
    DBCALLBACK_WAIT,
    (SQL_CALLBACK_PARM)&db_wait_request );
```

♦ **DB_CALLBACK_MESSAGE**   This is used to enable the application to
handle messages received from the server during the processing of a
request.

The callback prototype is as follows:

```

```
void SQL_CALLBACK message_callback(
SQLCA* sqlca,
unsigned char msg_type,
an_SQL_code code,
unsigned short length,
char* msg
);
```

The **msg_type** parameter states how important the message is and you may wish to handle different message types in different ways. The available message types are MESSAGE_TYPE_INFO, MESSAGE_TYPE_WARNING, MESSAGE_TYPE_ACTION, and MESSAGE_TYPE_STATUS. These constants are defined in *sqldef.h*. The **code** field is an identifier. The **length** field tells you how long the message is. The message is *not* null-terminated.

For example, the Interactive SQL callback displays STATUS and INFO message in the Messages pane, while messages of type ACTION and WARNING go to a dialog. If an application does not register this callback, there is a default callback, which causes all messages to be written to the server logfile (if debugging is on and a logfile is specified). In addition, messages of type MESSAGE_TYPE_WARNING and MESSAGE_TYPE_ACTION are more prominently displayed, in an operating system-dependent manner.

## db_start_database function

Prototype            unsigned int **db_start_database(** SQLCA * *sqlca***,** char * *parms* **);**

Arguments           **sqlca**    A pointer to a SQLCA structure. For information, see "The SQL Communication Area (SQLCA)" on page 161.

                           **parms**    A NULL-terminated string containing a semi-colon-delimited list of parameter settings, each of the form KEYWORD=**value**. For example,

```
"UID=DBA;PWD=SQL;DBF=c:\\db\\mydatabase.db"
```

☞ For an available list of connection parameters, see "Connection parameters" [*ASA Database Administration Guide,* page 176].

Description   If the database is not already running. The database is started on an existing server, if possible. Otherwise, a new server is started. The steps carried out to start a database are described in "Locating a server" [*ASA Database Administration Guide,* page 69].

                           If the database was already running or was successfully started, the return value is true (non-zero) and SQLCODE is set to 0. Error information is returned in the SQLCA.

If a user ID and password are supplied in the parameters, they are ignored.

☞ The permission required to start and stop a database is set on the server command line. For information, see "The database server" [*ASA Database Administration Guide,* page 116].

db_start_engine function

Prototype   unsigned int **db_start_engine(** SQLCA * *sqlca*, char * *parms* **);**

Arguments   **sqlca**   A pointer to a SQLCA structure. For information, see "The SQL Communication Area (SQLCA)" on page 161.

parms   A NULL-terminated string containing a semi-colon-delimited list of parameter settings, each of the form KEYWORD=**value**. For example,

```
"UID=DBA;PWD=SQL;DBF=c:\\db\\mydatabase.db"
```

☞ For an available list of connection parameters, see "Connection parameters" [*ASA Database Administration Guide,* page 176].

Description   Starts the database server if it is not running and the DBF parameter is provided. The steps carried out by this function are those listed in "Locating a server" [*ASA Database Administration Guide,* page 69].

If the database server was already running or was successfully started, the return value is true (non-zero) and SQLCODE is set to 0. Error information is returned in the SQLCA.

The following call to **db_start_engine** starts the database server and names it **asademo**, but does not load the database, despite the DBF connection parameter:

```
db_start_engine( &sqlca,
    "DBF=c:\\asa9\\asademo.db; Start=dbeng9" );
```

If you wish to start a database as well as the server, include the database file in the START connection parameter:

```
db_start_engine( &sqlca,
    "ENG=eng_name;START=dbeng9 c:\\asa\\asademo.db" );
```

This call starts the server, names it **eng_name**, and starts the **asademo** database on that server.

The **db_start_engine** function attempts to connect to a server before starting one, to avoid attempting to start a server that is already running.

The FORCESTART connection parameter is used only by the **db_start_engine** function. When set to YES, there is no attempt to connect

to a server before trying to start one. This enables the following pair of commands to work as expected:

1. Start a database server named server_1:

```
start dbeng9 -n server_1 asademo.db
```

2. Force a new server to start and connect to it:

```
db_start_engine( &sqlda,
  "START=dbeng9 -n server_2 asademo.db;ForceStart=YES" )
```

If FORCESTART was not used, and without an ENG parameter, the second command would have attempted to connect to server_1. The **db_start_engine** function does not pick up the server name from the -n option of the START parameter.

db_stop_database function

Prototype   unsigned int **db_stop_database(** SQLCA * *sqlca,* char * *parms* **);**

Arguments   **sqlca**   A pointer to a SQLCA structure. For information, see "The SQL Communication Area (SQLCA)" on page 161.

parms   A NULL-terminated string containing a semi-colon-delimited list of parameter settings, each of the form KEYWORD=**value**. For example,

```
"UID=DBA;PWD=SQL;DBF=c:\\db\\mydatabase.db"
```

☞ For an available list of connection parameters, see "Connection parameters" [*ASA Database Administration Guide,* page 176].

Description   Stop the database identified by **DatabaseName** on the server identified by **EngineName**. If **EngineName** is not specified, the default server is used.

By default, this function does not stop a database that has existing connections. If **Unconditional** is **yes**, the database is stopped regardless of existing connections.

A return value of TRUE indicates that there were no errors.

☞ The permission required to start and stop a database is set on the server command line. For information, see "The database server" [*ASA Database Administration Guide,* page 116].

db_stop_engine function

Prototype   unsigned int **db_stop_engine(** SQLCA * *sqlca,* char * *parms* **);**

Arguments	**sqlca** A pointer to a SQLCA structure. For information, see "The SQL Communication Area (SQLCA)" on page 161.

parms A NULL-terminated string containing a semi-colon-delimited list of parameter settings, each of the form KEYWORD=**value**. For example,

```
"UID=DBA;PWD=SQL;DBF=c:\\db\\mydatabase.db"
```

☞ For an available list of connection parameters, see "Connection parameters" [*ASA Database Administration Guide,* page 176].

Description	Terminates execution of the database server. The steps carried out by this function are:

♦ Look for a local database server that has a name that matches the **EngineName** parameter. If no **EngineName** is specified, look for the default local database server.

♦ If no matching server is found, this function fails.

♦ Send a request to the server to tell it to checkpoint and shut down all databases.

♦ Unload the database server.

By default, this function does not stop a database server that has existing connections. If **Unconditional** is *yes*, the database server is stopped regardless of existing connections.

A C program can use this function instead of spawning DBSTOP. A return value of TRUE indicates that there were no errors.

The use of **db_stop_engine** is subject to the permissions set with the -gk server option.

☞ For more information, see "-gk server option" [*ASA Database Administration Guide,* page 144].

db_string_connect function

Prototype	unsigned int **db_string_connect(** SQLCA * *sqlca,* char * *parms* **);**
Arguments	**sqlca** A pointer to a SQLCA structure. For information, see "The SQL Communication Area (SQLCA)" on page 161.

parms A NULL-terminated string containing a semi-colon-delimited list of parameter settings, each of the form KEYWORD=**value**. For example,

```
"UID=DBA;PWD=SQL;DBF=c:\\db\\mydatabase.db"
```

☞ For an available list of connection parameters, see "Connection parameters" [*ASA Database Administration Guide,* page 176].

Description

Provides extra functionality beyond the embedded SQL CONNECT command. This function carries out a connection using the algorithm described in "Troubleshooting connections" [*ASA Database Administration Guide,* page 66].

The return value is true (non-zero) if a connection was successfully established and false (zero) otherwise. Error information for starting the server, starting the database, or connecting is returned in the SQLCA.

db_string_disconnect function

Prototype

unsigned int **db_string_disconnect(**
  SQLCA * *sqlca***,**
  char * *parms* **);**

Arguments

sqlca   A pointer to a SQLCA structure. For information, see "The SQL Communication Area (SQLCA)" on page 161.

parms   A NULL-terminated string containing a semi-colon-delimited list of parameter settings, each of the form KEYWORD=**value**. For example,

```
"UID=DBA;PWD=SQL;DBF=c:\\db\\mydatabase.db"
```

☞ For an available list of connection parameters, see "Connection parameters" [*ASA Database Administration Guide,* page 176].

Description

This function disconnects the connection identified by the **ConnectionName** parameter. All other parameters are ignored.

If no **ConnectionName** parameter is specified in the string, the unnamed connection is disconnected. This is equivalent to the embedded SQL DISCONNECT command. The Boolean return value is true if a connection was successfully ended. Error information is returned in the SQLCA.

This function shuts down the database if it was started with the **AutoStop**=**yes** parameter and there are no other connections to the database. It also stops the server if it was started with the **AutoStop**=**yes** parameter and there are no other databases running.

db_string_ping_server function

Prototype

unsigned int **db_string_ping_server(**
SQLCA * *sqlca***,**
char * *connect_string***,**
unsigned int *connect_to_db* **);**

Description	The *connect_string* is a normal connect string that may or may not contain server and database information.
	If *connect_to_db* is non-zero (true), then the function attempts to connect to a database on a server. It returns a non-zero (true) value only if the connect string is sufficient to connect to the named database on the named server.
	If *connect_to_db* is zero, then the function only attempts to locate a server. It returns a non-zero value only if the connect string is sufficient to locate a server. It makes no attempt to connect to the database.

fill_s_sqlda function

Prototype	struct sqlda * **fill_s_sqlda(** struct sqlda * *sqlda*, unsigned int *maxlen* **);**
Description	The same as **fill_sqlda,** except that it changes all the data types in *sqlda* to type DT_STRING. Enough space is allocated to hold the string representation of the type originally specified by the SQLDA, up to a maximum of *maxlen* bytes. The length fields in the SQLDA (**sqllen**) are modified appropriately. Returns *sqlda* if successful and returns the null pointer if there is not enough memory available.

fill_sqlda function

Prototype	struct sqlda * **fill_sqlda(** struct sqlda * *sqlda* **);**
Description	Allocates space for each variable described in each descriptor of *sqlda*, and assigns the address of this memory to the **sqldata** field of the corresponding descriptor. Enough space is allocated for the database type and length indicated in the descriptor. Returns *sqlda* if successful and returns the null pointer if there is not enough memory available.

free_filled_sqlda function

Prototype	void **free_filled_sqlda(** struct sqlda * *sqlda* **);**
Description	Free the memory allocated to each **sqldata** pointer and the space allocated for the SQLDA itself. Any null pointer is not freed.
	Calling this function causes **free_sqlda** to be called automatically, and so any descriptors allocated by **alloc_sqlda** are freed.

free_sqlda function

Prototype	void **free_sqlda(** struct sqlda * *sqlda* **);**

Description   Free space allocated to this *sqlda* and free the indicator variable space, as allocated in **fill_sqlda**. Do not free the memory referenced by each **sqldata** pointer.

free_sqlda_noind function

Prototype   void **free_sqlda_noind(** struct sqlda * *sqlda* **);**

Description   Free space allocated to this *sqlda*. Do not free the memory referenced by each **sqldata** pointer. The indicator variable pointers are ignored.

"Database properties" [*ASA Database Administration Guide,* page 713]

"The Ping utility" [*ASA Database Administration Guide,* page 563]

sql_needs_quotes function

Prototype   unsigned int **sql_needs_quotes(** SQLCA **sqlca,* char **str* **);**

Description   Returns a Boolean value that indicates whether the string requires double quotes around it when it is used as a SQL identifier. This function formulates a request to the database server to determine if quotes are needed. Relevant information is stored in the **sqlcode** field.

There are three cases of return value/code combinations:

♦ **return = FALSE, sqlcode = 0**   In this case, the string definitely does not need quotes.

♦ **return = TRUE**   In this case, sqlcode is always SQLE_WARNING, and the string definitely does need quotes.

♦ **return = FALSE**   If sqlcode is something other than SQLE_WARNING, the test is inconclusive.

sqlda_storage function

Prototype   unsigned long **sqlda_storage(** struct sqlda **sqlda,* int *varno* **);**

Description   Returns the amount of storage required to store any value for the variable described in **sqlda->sqlvar[varno]**.

sqlda_string_length function

Prototype   unsigned long **sqlda_string_length(** SQLDA **sqlda,* int *varno* **);**

Description   Returns the length of the C string (type DT_STRING) that would be required to hold the variable **sqlda->sqlvar[varno]** (no matter what its type is).

sqlerror_message function

Prototype   char ***sqlerror_message(** SQLCA **sqlca***,** char * *buffer***,** int *max* **);**

Description   Return a pointer to a string that contains an error message. The error message contains text for the error code in the SQLCA. If no error was indicated, a null pointer is returned. The error message is placed in the buffer supplied, truncated to length *max* if necessary.

Embedded SQL command summary

> **EXEC SQL**
> ALL embedded SQL statements must be preceded with EXEC SQL and end with a semicolon (;).

There are two groups of embedded SQL commands. Standard SQL commands are used by simply placing them in a C program enclosed with EXEC SQL and a semi-colon (;). CONNECT, DELETE, SELECT, SET, and UPDATE have additional formats only available in embedded SQL. The additional formats fall into the second category of embedded SQL specific commands.

☞ For descriptions of the standard SQL commands, see "SQL Statements" [*ASA SQL Reference,* page 253].

Several SQL commands are specific to embedded SQL and can only be used in a C program.

☞ For more information about these embedded SQL commands, see "SQL Language Elements" [*ASA SQL Reference,* page 3].

☞ Standard data manipulation and data definition statements can be used from embedded SQL applications. In addition the following statements are specifically for embedded SQL programming:

♦ **ALLOCATE DESCRIPTOR**   allocate memory for a descriptor

   ☞ See "ALLOCATE DESCRIPTOR statement [ESQL]" [*ASA SQL Reference,* page 264]

♦ **CLOSE**   close a cursor

   ☞ See "CLOSE statement [ESQL] [SP]" [*ASA SQL Reference,* page 325]

♦ **CONNECT**   connect to the database

   ☞ See "CONNECT statement [ESQL] [Interactive SQL]" [*ASA SQL Reference,* page 332]

♦ **DEALLOCATE DESCRIPTOR**   reclaim memory for a descriptor

   ☞ See "DEALLOCATE DESCRIPTOR statement [ESQL]" [*ASA SQL Reference,* page 433]

♦ **Declaration Section**   declare host variables for database communication

   ☞ See "Declaration section [ESQL]" [*ASA SQL Reference,* page 434]

♦ **DECLARE CURSOR**   declare a cursor

☞ See "DECLARE CURSOR statement [ESQL] [SP]" [*ASA SQL Reference,* page 436]

♦ **DELETE (positioned)**   delete the row at the current position in a cursor

☞ See "DELETE (positioned) statement [ESQL] [SP]" [*ASA SQL Reference,* page 447]

♦ **DESCRIBE**   describe the host variables for a particular SQL statement

☞ See "DESCRIBE statement [ESQL]" [*ASA SQL Reference,* page 449]

♦ **DISCONNECT**   disconnect from database server

☞ See "DISCONNECT statement [ESQL] [Interactive SQL]" [*ASA SQL Reference,* page 453]

♦ **DROP STATEMENT**   free resources used by a prepared statement

☞ See "DROP STATEMENT statement [ESQL]" [*ASA SQL Reference,* page 463]

♦ **EXECUTE**   execute a particular SQL statement

☞ See "EXECUTE statement [ESQL]" [*ASA SQL Reference,* page 471]

♦ **EXPLAIN**   explain the optimization strategy for a particular cursor

☞ See "EXPLAIN statement [ESQL]" [*ASA SQL Reference,* page 480]

♦ **FETCH**   fetch a row from a cursor

☞ See "FETCH statement [ESQL] [SP]" [*ASA SQL Reference,* page 482]

♦ **GET DATA**   fetch long values from a cursor

☞ See "GET DATA statement [ESQL]" [*ASA SQL Reference,* page 497]

♦ **GET DESCRIPTOR**   retrieve information about a variable in a SQLDA.

☞ See "GET DESCRIPTOR statement [ESQL]" [*ASA SQL Reference,* page 499]

♦ **GET OPTION**   get the setting for a particular database option

☞ See "GET OPTION statement [ESQL]" [*ASA SQL Reference,* page 501]

♦ **INCLUDE**   include a file for SQL preprocessing

☞ See "INCLUDE statement [ESQL]" [*ASA SQL Reference,* page 522]

♦ **OPEN**   open a cursor

☞ See "OPEN statement [ESQL] [SP]" [*ASA SQL Reference,* page 553]

♦ **PREPARE**   prepare a particular SQL statement

☞ See "PREPARE statement [ESQL]" [*ASA SQL Reference,* page 563]

♦ **PUT**   insert a row into a cursor

☞ See "PUT statement [ESQL]" [*ASA SQL Reference,* page 567]

♦ **SET CONNECTION**   change active connection

☞ See "SET CONNECTION statement [Interactive SQL] [ESQL]" [*ASA SQL Reference,* page 610]

♦ **SET DESCRIPTOR**   describe the variables in a SQLDA and place data into the SQLDA

☞ See "SET DESCRIPTOR statement [ESQL]" [*ASA SQL Reference,* page 611]

♦ **SET SQLCA**   use an SQLCA other than the default global one

☞ See "SET SQLCA statement [ESQL]" [*ASA SQL Reference,* page 619]

♦ **UPDATE (positioned)**   update the row at the current location of a cursor

☞ See "UPDATE (positioned) statement [ESQL] [SP]" [*ASA SQL Reference,* page 655]

♦ **WHENEVER**   specify actions to occur on errors in SQL statements

☞ See "WHENEVER statement [ESQL]" [*ASA SQL Reference,* page 666]

ODBC Programming

About this chapter

This chapter presents information for developing applications that call the ODBC programming interface directly.

The primary documentation for ODBC application development is the Microsoft ODBC SDK documentation, available as part of the Microsoft Data Access Components (MDAC) SDK. This chapter provides introductory material and describes features specific to Adaptive Server Anywhere, but is not an exhaustive guide to ODBC application programming.

Some application development tools that already have ODBC support provide their own programming interface that hides the ODBC interface. This chapter is not intended for users of those tools.

Contents

Introduction to ODBC

The **Open Database Connectivity** (**ODBC**) interface is an application programming interface defined by Microsoft Corporation as a standard interface to database-management systems on Windows operating systems. ODBC is a call-based interface.

To write ODBC applications for Adaptive Server Anywhere, you need:

♦ Adaptive Server Anywhere.

♦ A C compiler capable of creating programs for your environment.

♦ Microsoft ODBC Software Development Kit. This is available on the Microsoft Developer Network, and provides documentation and additional tools for testing ODBC applications.

Supported platforms

Adaptive Server Anywhere supports the ODBC API on UNIX and Windows CE, in addition to Windows. Having multi-platform ODBC support makes portable database application development much easier.

☞ For information on enlisting the ODBC driver in distributed transactions, see "Three-Tier Computing and Distributed Transactions" on page 507.

ODBC conformance

Adaptive Server Anywhere provides support for ODBC 3.5, which is supplied as part of the Microsoft Data Access Kit 2.7.

Levels of ODBC support

ODBC features are arranged according to level of conformance. Features are either **Core**, **Level 1**, or **Level 2**, with Level 2 being the most complete level of ODBC support. These features are listed in the *ODBC Programmer's Reference*, which is available from Microsoft Corporation as part of the ODBC software development kit or from the Microsoft Web site, at *http://msdn.microsoft.com/library/en-us/odbc/htm/odbcabout_this_manual.-asp*.

Features supported by Adaptive Server Anywhere

Adaptive Server Anywhere supports the ODBC 3.5 specification.

♦ **Core conformance**   Adaptive Server Anywhere supports all Core level features.

♦ **Level 1 conformance**   Adaptive Server Anywhere supports all Level 1 features, except for asynchronous execution of ODBC functions.

Adaptive Server Anywhere supports multiple threads sharing a single connection. The requests from the different threads are serialized by Adaptive Server Anywhere.

♦ **Level 2 conformance**   Adaptive Server Anywhere supports all Level 2 features, except for the following:

- Three part names of tables and views. This is not applicable for Adaptive Server Anywhere.

- Asynchronous execution of ODBC functions for specified individual statements.

- Ability to time out login request and SQL queries.

ODBC backwards compatibility

Applications developed using older versions of ODBC continue to work with Adaptive Server Anywhere and the newer ODBC Driver Manager. The new ODBC features are not provided for older applications.

The ODBC Driver Manager

The ODBC Driver Manager is part of the ODBC software supplied with Adaptive Server Anywhere. The ODBC Version 3 Driver Manager has a new interface for configuring ODBC data sources.

Building ODBC applications

This section describes how to compile and link simple ODBC applications.

Including the ODBC header file

Every C source file that calls ODBC functions must include a platform-specific ODBC header file. Each platform-specific header file includes the main ODBC header file *odbc.h*, which defines all the functions, data types and constant definitions required to write an ODBC program.

❖ **To include the ODBC header file in a C source file**

1. Add an include line referencing the appropriate platform-specific header file to your source file. The lines to use are as follows:

Operating system	Include line
Windows	`#include "ntodbc.h"`
UNIX	`#include "unixodbc.h"`
Windows CE	`#include "ntodbc.h"`

2. Add the directory containing the header file to the include path for your compiler.

   Both the platform-specific header files and *odbc.h* are installed in the *h* subdirectory of your SQL Anywhere directory.

Linking ODBC applications on Windows

This section does not apply to Windows CE. For more information see

When linking your application, you must link against the appropriate import library file to have access to the ODBC functions. The import library defines entry points for the ODBC Driver Manager *odbc32.dll.* The Driver Manager in turn loads the Adaptive Server Anywhere ODBC driver *dbodbc9.dll.*

Separate import libraries are supplied for Microsoft, Watcom, and Borland compilers.

❖ **To link an ODBC application (Windows)**

1. Add the directory containing the platform-specific import library to the list of library directories.

   The import libraries are stored in the *lib* subdirectory of the directory containing your Adaptive Server Anywhere executables and are named as follows:

Operating system	Compiler	Import library
Windows	Microsoft	*odbc32.lib*
Windows	Watcom C/C++	*wodbc32.lib*
Windows	Borland	*bodbc32.lib*
Windows CE	Microsoft	*dbodbc9.lib*

Linking ODBC applications on Windows CE

On Windows CE operating systems there is no ODBC Driver Manager. The import library (*dbodbc9.lib*) defines entry points directly into the Adaptive Server Anywhere ODBC driver *dbodbc9.dll*.

Separate versions of this DLL are provided for the different chips on which Windows CE is available. The files are in operating-system specific subdirectories of the *ce* directory in your SQL Anywhere directory. For example, the ODBC driver for Windows CE on the ARM chip is in the following location:

```
C:\Program Files\Sybase\SQL Anywhere 9\ce\arm.30
```

☞ For a list of supported versions of Windows CE, see "Windows and NetWare operating systems" [*Introducing SQL Anywhere Studio,* page 98].

❖ **To link an ODBC application (Windows CE)**

1. Add the directory containing the platform-specific import library to the list of library directories.

   The import library is named *dbodbc9.lib* and is stored in an operating-system specific location under the *ce* directory in your SQL Anywhere directory. For example, the import library for Windows CE on the ARM chip is in the following location:

```
C:\Program Files\Sybase\SQL Anywhere 9\ce\arm.30\lib
```

2. Specify the **DRIVER=** parameter in the connection string supplied to the **SQLDriverConnect** function.

```
szConnStrIn = "driver=ospath\dbodbc9.dll;dbf=c:\asademo.db"
```

where *ospath* is the full path to the chip-specific subdirectory of your SQL Anywhere directory on the Windows CE device. For example:

```
\Program Files\Sybase\SQL Anywhere 9\ce\arm.30\lib
```

The sample program (*odbc.c*) uses a File data source (FileDSN connection parameter) called *ASA 9.0 Sample.dsn.* You can create File data sources on your desktop system from the ODBC Driver Manager and copy them to your Windows CE device.

Windows CE and Unicode

Adaptive Server Anywhere uses an encoding known as UTF-8, a multi-byte character encoding which can be used to encode Unicode.

The Adaptive Server Anywhere ODBC driver supports either ASCII (8-bit) strings or Unicode code (wide character) strings. The UNICODE macro controls whether ODBC functions expect ASCII or Unicode strings. If your application must be built with the UNICODE macro defined, but you want to use the ASCII ODBC functions, then the SQL_NOUNICODEMAP macro must also be defined.

The *Samples\ASA\C\odbc.c* sample file illustrates how to use the Unicode ODBC features.

Linking ODBC applications on UNIX

An ODBC Driver Manager is not included with Adaptive Server Anywhere, but there are third party Driver Managers available. This section describes how to build ODBC applications that do not use an ODBC Driver Manager.

ODBC driver

The ODBC driver is a shared object or shared library. Separate versions of the Adaptive Server Anywhere ODBC driver are supplied for single-threaded and multi-threaded applications.

The ODBC drivers are the following files:

Operating system	Threading model	ODBC driver
Solaris/Sparc	Single threaded	*dbodbc9.so* (*dbodbc9.so.1*)
Solaris/Sparc	Multi-threaded	*dbodbc_r.so* (*dbodbc_r.so.1*)

The libraries are installed as symbolic links to the shared library with a version number (in parentheses).

❖ **To link an ODBC application (UNIX)**

1. Link your application directly against the appropriate ODBC driver.

2. When deploying your application, ensure that the appropriate ODBC driver is available in the user's library path.

Data source information   If Adaptive Server Anywhere does not detect the presence of an ODBC Driver Manager, it uses *~/.odbc.ini* for data source information.

Using an ODBC Driver Manager on UNIX

Third-party ODBC Driver Managers for UNIX are available. An ODBC Driver Manager includes the following files:

Operating system	Files
Solaris/Sparc	*libodbc.so* (*libodbc.so.1*)
	libodbcinst.so (*libodbcinst.so.1*)

If your are deploying an application that requires an ODBC Driver Manager and you are not using a third-party Driver Manager, create symbolic links for both the *libodbc* and *libodbcinst* shared libraries to the Adaptive Server Anywhere ODBC driver.

If an ODBC Driver Manager is present, Adaptive Server Anywhere queries the Driver Manager rather than *~/.odbc.ini* for data source information.

Standard ODBC applications do not link directly against the ODBC driver. Instead, ODBC function calls go through the ODBC Driver Manager. On UNIX and Windows CE operating systems, Adaptive Server Anywhere does not include an ODBC Driver Manager. You can still create ODBC applications by linking directly against the Adaptive Server Anywhere ODBC driver, but you can then access only Adaptive Server Anywhere data sources.

ODBC samples

Several ODBC samples are included with Adaptive Server Anywhere. You can find the samples in the *Samples\ASA* subdirectory of your SQL Anywhere directory. By default, this is

```
C:\Program Files\Sybase\SQL Anywhere 9\Samples\ASA
```

The samples in directories starting with *ODBC* illustrate separate and simple ODBC tasks, such as connecting to a database and executing statements. A complete sample ODBC program is supplied as *Samples\ASA\C\odbc.c*. The program performs the same actions as the embedded SQL dynamic cursor example program that is in the same directory.

☞ For a description of the associated embedded SQL program, see "Sample embedded SQL programs" on page 143.

Building the sample ODBC program

The ODBC sample program in *Samples\ASA\C* includes a batch file (shell script for UNIX) that can be used to compile and link the sample application.

❖ **To build the sample ODBC program**

1. Open a command prompt and change directory to the *Samples\ASA\C* subdirectory of your SQL Anywhere directory.

2. Run the *makeall* batch file or shell script

   The format of the command is as follows:

   ```
   makeall api platform compiler
   ```

   The parameters are as follows:

   ♦ *API*   Specify **odbc** to compile the ODBC example rather than an embedded SQL version of the application.

   ♦ *Platform*   Specify **WINNT** to compile for Windows operating systems.

   ♦ *Compiler*   Specify the compiler to use to compile the program. The compiler can be one of the following:

   - **WC**   use Watcom C/C++
   - **MC**   use Microsoft Visual C++
   - **BC**   use Borland C++ Builder

Running the sample ODBC program

The sample program *odbc.c*, when compiled for versions of Windows that support services, runs optionally as a service.

The two files containing the example code for Windows services are the source file *ntsvc.c* and the header file *ntsvc.h*. The code allows the linked executable to be run either as a regular executable or as a Windows service.

❖ **To run the ODBC sample**

1. Start the program:
   ♦ Run the file *Samples\ASA\C\odbcwnt.exe*.

2. Choose a table:
   ♦ Choose one of the tables in the sample database. For example, you may enter **Customer** or **Employee**.

❖ **To run the ODBC sample as a Windows service**

1. Start Sybase Central.

2. In the left pane, select Adaptive Server Anywhere 9.

3. In the right pane, select the Services tab.

4. From the File menu, choose New ➤ Service.

   The Service Creation wizard appears.

5. On the first page, enter a name for the service.

6. On the second page, select Sample program.

7. On the third page, browse to the sample program (*odbcwnt.exe*) from the *Samples\ASA\C* subdirectory of your SQL Anywhere directory.

8. Complete the wizard to install the service.

9. Click Start on the main window to start the service.

When run as a service, the program displays the normal user interface if possible. It also writes the output to the Application Event Log. If it is not possible to start the user interface, the program prints one page of data to the Application Event Log and stops.

ODBC handles

ODBC applications use a small set of **handles** to define basic features such as database connections and SQL statements. A handle is a 32-bit value.

The following handles are used in essentially all ODBC applications.

♦ **Environment** The environment handle provides a global context in which to access data. Every ODBC application must allocate exactly one environment handle upon starting, and must free it at the end.

The following code illustrates how to allocate an environment handle:

```
SQLHENV env;
SQLRETURN rc;
rc = SQLAllocHandle( SQL_HANDLE_ENV, SQL
    _NULL_HANDLE, &env );
```

♦ **Connection** A connection is specified by an ODBC driver and a data source. An application can have several connections associated with its environment. Allocating a connection handle does not establish a connection; a connection handle must be allocated first and then used when the connection is established.

The following code illustrates how to allocate a connection handle:

```
SQLHDBC  dbc;
SQLRETURN rc;
rc = SQLAllocHandle( SQL_HANDLE_DBC, env, &dbc );
```

♦ **Statement** A statement handle provides access to a SQL statement and any information associated with it, such as result sets and parameters. Each connection can have several statements. Statements are used both for cursor operations (fetching data) and for single statement execution (e.g. INSERT, UPDATE, and DELETE).

The following code illustrates how to allocate a statement handle:

```
SQLHSTMT stmt;
SQLRETURN rc;
rc = SQLAllocHandle( SQL_HANDLE_STMT, dbc, &stmt );
```

Allocating ODBC handles

The handle types required for ODBC programs are as follows:

Item	Handle type
Environment	SQLHENV
Connection	SQLHDBC
Statement	SQLHSTMT
Descriptor	SQLHDESC

❖ To use an ODBC handle

1. Call the **SQLAllocHandle** function.

   SQLAllocHandle takes the following parameters:

   ♦ an identifier for the type of item being allocated

   ♦ the handle of the parent item

   ♦ a pointer to the location of the handle to be allocated

   ☞ For a full description, see SQLAllocHandle in the Microsoft *ODBC Programmer's Reference*.

2. Use the handle in subsequent function calls.

3. Free the object using **SQLFreeHandle**.

   SQLFreeHandle takes the following parameters:

   ♦ an identifier for the type of item being freed

   ♦ the handle of the item being freed

   ☞ For a full description, see SQLFreeHandle in the Microsoft ODBC Programmer's Reference.

Example The following code fragment allocates and frees an environment handle:

```
SQLHENV env;
SQLRETURN retcode;
retcode = SQLAllocHandle(
      SQL_HANDLE_ENV,
      SQL_NULL_HANDLE,
      &env );
if( retcode == SQL_SUCCESS
    || retcode == SQL_SUCCESS_WITH_INFO ) {
  // success: application code here
}
SQLFreeHandle( SQL_HANDLE_ENV, env );
```

☞ For more information on return codes and error handling, see "Handling errors" on page 255.

A first ODBC example

The following is a simple ODBC program that connects to the Adaptive Server Anywhere sample database and immediately disconnects.

☞ You can find this sample as
Samples\ASA\ODBCConnect\odbcconnect.cpp in your SQL Anywhere directory.

```
#include <stdio.h>
#include "ntodbc.h"

int main(int argc, char* argv[])
{
  SQLHENV  env;
  SQLHDBC  dbc;
  SQLRETURN retcode;

  retcode = SQLAllocHandle( SQL_HANDLE_ENV,
             SQL_NULL_HANDLE,
             &env );
  if (retcode == SQL_SUCCESS
      || retcode == SQL_SUCCESS_WITH_INFO) {
    printf( "env allocated\n" );
    /* Set the ODBC version environment attribute */
    retcode = SQLSetEnvAttr( env,
                SQL_ATTR_ODBC_VERSION,
                (void*)SQL_OV_ODBC3, 0);
      retcode = SQLAllocHandle( SQL_HANDLE_DBC, env, &dbc );
      if (retcode == SQL_SUCCESS
          || retcode == SQL_SUCCESS_WITH_INFO) {
        printf( "dbc allocated\n" );
        retcode = SQLConnect( dbc,
          (SQLCHAR*) "ASA 9.0 Sample", SQL_NTS,
          (SQLCHAR* ) "DBA", SQL_NTS,
          (SQLCHAR*) "SQL", SQL_NTS );
        if (retcode == SQL_SUCCESS
            || retcode == SQL_SUCCESS_WITH_INFO) {
          printf( "Successfully connected\n" );
        }
        SQLDisconnect( dbc );
      }
      SQLFreeHandle( SQL_HANDLE_DBC, dbc );
    }
    SQLFreeHandle( SQL_HANDLE_ENV, env );
  return 0;
}
```

Connecting to a data source

This section describes how to use ODBC functions to establish a connection to an Adaptive Server Anywhere database.

Choosing an ODBC connection function

ODBC supplies a set of connection functions. Which one you use depends on how you expect your application to be deployed and used:

♦ **SQLConnect**   The simplest connection function.

SQLConnect takes a data source name and optional user ID and password. You may wish to use **SQLConnect** if you hard-code a data source name into your application.

☞ For more information, see SQLConnect in the Microsoft *ODBC Programmer's Reference.*

♦ **SQLDriverConnect**   Connects to a data source using a connection string.

SQLDriverConnect allows the application to use Adaptive Server Anywhere-specific connection information that is external to the data source. Also, you can use **SQLDriverConnect** to request that the Adaptive Server Anywhere driver prompt for connection information.

SQLDriverConnect can also be used to connect without specifying a data source.

☞ For more information, see SQLDriverConnect in the Microsoft *ODBC Programmer's Reference.*

♦ **SQLBrowseConnect**   Connects to a data source using a connection string, like **SQLDriverConnect**.

SQLBrowseConnect allows your application to build its own dialog boxes to prompt for connection information and to browse for data sources used by a particular driver (in this case the Adaptive Server Anywhere driver).

☞ For more information, see SQLBrowseConnect in the Microsoft *ODBC Programmer's Reference.*

The examples in this chapter mainly use **SQLDriverConnect**.

☞ For a complete list of connection parameters that can be used in connection strings, see "Connection parameters" [*ASA Database Administration Guide,* page 176].

Establishing a connection

Your application must establish a connection before it can carry out any database operations.

❖ **To establish an ODBC connection**

1. Allocate an ODBC environment.

   For example:

   ```
   SQLHENV   env;
   SQLRETURN retcode;
   retcode = SQLAllocHandle( SQL_HANDLE_ENV,
     SQL_NULL_HANDLE, &env );
   ```

2. Declare the ODBC version.

   By declaring that the application follows ODBC version 3, SQLSTATE values and some other version-dependent features are set to the proper behavior. For example:

   ```
   retcode = SQLSetEnvAttr( env,
     SQL_ATTR_ODBC_VERSION, (void*)SQL_OV_ODBC3, 0);
   ```

3. If necessary, assemble the data source or connection string.

   Depending on your application, you may have a hard-coded data source or connection string, or you may store it externally for greater flexibility.

4. Allocate an ODBC connection item.

   For example:

   ```
   retcode = SQLAllocHandle( SQL_HANDLE_DBC, env, &dbc );
   ```

5. Set any connection attributes that must be set before connecting.

   Some connection attributes must be set before establishing a connection, while others can be set either before or after. For example:

   ```
   retcode = SQLSetConnectAttr( dbc,
     SQL_AUTOCOMMIT,
     (SQLPOINTER)SQL_AUTOCOMMIT_OFF, 0 );
   ```

   ☞ For more information, see "Setting connection attributes" on page 243.

6. Call the ODBC connection function.

   For example:

```
                       if (retcode == SQL_SUCCESS
                           || retcode == SQL_SUCCESS_WITH_INFO) {
                       printf( "dbc allocated\n" );
                       retcode = SQLConnect( dbc,
                           (SQLCHAR*) "ASA 9.0 Sample", SQL_NTS,
                           (SQLCHAR* ) "DBA", SQL_NTS,
                           (SQLCHAR*) "SQL", SQL_NTS );
                       if (retcode == SQL_SUCCESS
                              || retcode == SQL_SUCCESS_WITH_INFO){
                       // successfully connected.
```

☞ You can find a complete sample as
Samples\ASA\ODBCConnect\odbcconnect.cpp in your SQL Anywhere
directory.

Notes
♦ **SQL_NTS** Every string passed to ODBC has a corresponding length. If
the length is unknown, you can pass SQL_NTS indicating that it is a **Null
Terminated String** whose end is marked by the null character (\0).

♦ **SQLSetConnectAttr** By default, ODBC operates in auto-commit mode.
This mode is turned off by setting SQL_AUTOCOMMIT to false.

☞ For more information, see "Setting connection attributes" on
page 243.

Setting connection attributes

You use the SQLSetConnectAttr function to control details of the
connection. For example, the following statement turns off ODBC
autocommit behavior.

```
retcode = SQLSetConnectAttr( dbc, SQL_AUTOCOMMIT,
    (SQLPOINTER)SQL_AUTOCOMMIT_OFF, 0 );
```

☞ For more information including a list of connection attributes, see
SQLSetConnectAttr in the Microsoft *ODBC Programmer's Reference*.

Many aspects of the connection can be controlled through the connection
parameters. For information, see "Connection parameters" [*ASA Database
Administration Guide,* page 176].

Threads and connections in ODBC applications

You can develop multi-threaded ODBC applications for Adaptive Server
Anywhere. It is recommended that you use a separate connection for each
thread.

You can use a single connection for multiple threads. However, the database
server does not allow more than one active request for any one connection at

a time. If one thread executes a statement that takes a long time, all other threads must wait until the request is complete.

Executing SQL statements

ODBC includes several functions for executing SQL statements:

♦ **Direct execution**   Adaptive Server Anywhere parses the SQL statement, prepares an access plan, and executes the statement. Parsing and access plan preparation are called **preparing** the statement.

♦ **Prepared execution**   The statement preparation is carried out separately from the execution. For statements that are to be executed repeatedly, this avoids repeated preparation and so improves performance.

☞ See "Executing prepared statements" on page 247.

Executing statements directly

The **SQLExecDirect** function prepares and executes a SQL statement. The statement may be optionally include parameters.

The following code fragment illustrates how to execute a statement without parameters. The **SQLExecDirect** function takes a statement handle, a SQL string, and a length or termination indicator, which in this case is a null-terminated string indicator.

☞ The procedure described in this section is straightforward but inflexible. The application cannot take any input from the user to modify the statement. For a more flexible method of constructing statements, see "Executing statements with bound parameters" on page 246.

❖ **To execute a SQL statement in an ODBC application**

1. Allocate a handle for the statement using **SQLAllocHandle**.

   For example, the following statement allocates a handle of type SQL_HANDLE_STMT with name stmt, on a connection with handle dbc:

   ```
   SQLAllocHandle( SQL_HANDLE_STMT, dbc, &stmt );
   ```

2. Call the SQLExecDirect function to execute the statement:

   For example, the following lines declare a statement and execute it. The declaration of deletestmt would usually occur at the beginning of the function:

   ```
   SQLCHAR deletestmt[ STMT_LEN ] =
     "DELETE FROM department WHERE dept_id = 201";
   SQLExecDirect( stmt, deletestmt, SQL_NTS) ;
   ```

☞ For a complete sample with error checking, see *Samples\ASA\ODBCExecute\odbcexecute.cpp*.

Executing statements with bound parameters

This section describes how to construct and execute a SQL statement, using bound parameters to set values for statement parameters at runtime.

❖ **To execute a SQL statement with bound parameters in an ODBC application**

1. Allocate a handle for the statement using **SQLAllocHandle**.

   For example, the following statement allocates a handle of type SQL_HANDLE_STMT with name stmt, on a connection with handle dbc:

   ```
   SQLAllocHandle( SQL_HANDLE_STMT, dbc, &stmt );
   ```

2. Bind parameters for the statement using **SQLBindParameter**.

   For example, the following lines declare variables to hold the values for the department ID, department name, and manager ID, as well as for the statement string itself. They then bind parameters to the first, second, and third parameters of a statement executed using the **stmt** statement handle.

   ```
   #defined DEPT_NAME_LEN 20
   SQLINTEGER cbDeptID = 0,
       cbDeptName = SQL_NTS, cbManagerID = 0;
   SQLCHAR deptname[ DEPT_NAME_LEN ];
   SQLSMALLINT deptID, managerID;
   SQLCHAR insertstmt[ STMT_LEN ] =
     "INSERT INTO department "
     "( dept_id, dept_name, dept_head_id )"
     "VALUES (?, ?, ?)";
   SQLBindParameter( stmt, 1, SQL_PARAM_INPUT,
       SQL_C_SSHORT, SQL_INTEGER, 0, 0,
       &deptID, 0, &cbDeptID);
   SQLBindParameter( stmt, 2, SQL_PARAM_INPUT,
       SQL_C_CHAR, SQL_CHAR, DEPT_NAME_LEN, 0,
       deptname, 0,&cbDeptName);
   SQLBindParameter( stmt, 3, SQL_PARAM_INPUT,
       SQL_C_SSHORT, SQL_INTEGER, 0, 0,
       &managerID, 0, &cbManagerID);
   ```

3. Assign values to the parameters.

   For example, the following lines assign values to the parameters for the fragment of step 2.

   ```
   deptID = 201;
   strcpy( (char * ) deptname, "Sales East" );
   managerID = 902;
   ```

Commonly, these variables would be set in response to user action.

4. Execute the statement using **SQLExecDirect**.

   For example, the following line executes the statement string held in
   insertstmt on the statement handle stmt.

   ```
   SQLExecDirect( stmt, insertstmt, SQL_NTS) ;
   ```

Bind parameters are also used with prepared statements to provide
performance benefits for statements that are executed more than once. For
more information, see "Executing prepared statements" on page 247

☞ The above code fragments to not include error checking. For a complete
sample, including error checking, see
Samples\ASA\ODBCExecute\odbcexecute.cpp.

☞ For more information on **SQLExecDirect**, see SQLExecDirect in the
Microsoft *ODBC Programmer's Reference.*

Executing prepared statements

Prepared statements provide performance advantages for statements that are
used repeatedly. ODBC provides a full set of functions for using prepared
statements.

☞ For an introduction to prepared statements, see "Preparing statements"
on page 14.

❖ **To execute a prepared SQL statement**

1. Prepare the statement using **SQLPrepare**.

   For example, the following code fragment illustrates how to prepare an
   INSERT statement:

   ```
   SQLRETURN   retcode;
   SQLHSTMT    stmt;
   retcode = SQLPrepare( stmt,
               "INSERT INTO department
                ( dept_id, dept_name, dept_head_id )
                VALUES (?, ?, ?,)",
               SQL_NTS);
   ```

   In this example:
   - ♦ **retcode**   Holds a return code that should be tested for success or
     failure of the operation.
   - ♦ **stmt**   Provides a handle to the statement so that it can be referenced
     later.
   - ♦ **?**   The question marks are placeholders for statement parameters.

2. Set statement parameter values using **SQLBindParameter**.

For example, the following function call sets the value of the dept_id variable:

```
SQLBindParameter( stmt,
                  1,
                  SQL_PARAM_INPUT,
                  SQL_C_SSHORT,
                  SQL_INTEGER,
                  0,
                  0,
                  &sDeptID,
                  0,
                  &cbDeptID);
```

In this example:

♦ **stmt** is the statement handle

♦ **1** indicates that this call sets the value of the first placeholder.

♦ **SQL_PARAM_INPUT** indicates that the parameter is an input statement.

♦ **SQL_C_SHORT** indicates the C data type being used in the application.

♦ **SQL_INTEGER** indicates SQL data type being used in the database.

♦ The next two parameters indicate the column precision and the number of decimal digits: both zero for integers.

♦ **&sDeptID** is a pointer to a buffer for the parameter value.

♦ **0** indicates the length of the buffer, in bytes.

♦ **&cbDeptID** is a pointer to a buffer for the length of the parameter value.

3. Bind the other two parameters and assign values to **sDeptId**.

4. Execute the statement:

```
retcode = SQLExecute( stmt);
```

Steps 2 to 4 can be carried out multiple times.

5. Drop the statement.

Dropping the statement frees resources associated with the statement itself. You drop statements using **SQLFreeHandle**.

☞ For a complete sample, including error checking, see *Samples\ASA\ODBCPrepare\odbcprepare.cpp.*

☞ For more information on **SQLPrepare**, see SQLPrepare in the Microsoft *ODBC Programmer's Reference.*

Working with result sets

ODBC applications use cursors to manipulate and update result sets. Adaptive Server Anywhere provides extensive support for different kinds of cursors and cursor operations.

☞ For an introduction to cursors, see "Working with cursors" on page 21.

Choosing a cursor characteristics

ODBC functions that execute statements and manipulate result sets use cursors to carry out their tasks. Applications open a cursor implicitly whenever they execute a **SQLExecute** or **SQLExecDirect** function.

For applications that move through a result set only in a forward direction and do not update the result set, cursor behavior is relatively straightforward. By default, ODBC applications request this behavior. ODBC defines a read-only, forward-only cursor, and Adaptive Server Anywhere provides a cursor optimized for performance in this case.

☞ For a simple example of a forward-only cursor, see "Retrieving data" on page 250.

For applications that need to scroll both forward and backward through a result set, such as many graphical user interface applications, cursor behavior is more complex. What does the application when it returns to a row that has been updated by some other application? ODBC defines a variety of **scrollable cursors** to allow you to build in the behavior that suits your application. Adaptive Server Anywhere provides a full set of cursors to match the ODBC scrollable cursor types.

You set the required ODBC cursor characteristics by calling the **SQLSetStmtAttr** function that defines statement attributes. You must call **SQLSetStmtAttr** before executing a statement that creates a result set.

You can use SQLSetStmtAttr to set many cursor characteristics. The characteristics that determine the cursor type that Adaptive Server Anywhere supplies include the following:

♦ **SQL_ATTR_CURSOR_SCROLLABLE**   Set to SQL_SCROLLABLE for a scrollable cursor and SQL_NONSCROLLABLE for a forward-only cursor. SQL_NONSCROLLABLE is the default.

♦ **SQL_ATTR_CONCURRENCY**   Set to one of the following values:
  • **SQL_CONCUR_READ_ONLY**   Disallow updates. SQL_CONCUR_READ_ONLY is the default.

- **SQL_CONCUR_LOCK** Use the lowest level of locking sufficient to ensure that the row can be updated.

- **SQL_CONCUR_ROWVER** Use optimistic concurrency control, comparing row versions such as SQLBase ROWID or Sybase TIMESTAMP.

- **SQL_CONCUR_VALUES** Use optimistic concurrency control, comparing values.

☞ For more information, see SQLSetStmtAttr in the Microsoft *ODBC Programmer's Reference*.

Example The following fragment requests a read-only, scrollable cursor:

```
SQLAllocHandle( SQL_HANDLE_STMT, dbc, &stmt );
SQLSetStmtAttr( stmt, SQL_ATTR_CURSOR_SCROLLABLE,
    SQL_SCROLLABLE, 0 );
```

Retrieving data

To retrieve rows from a database, you execute a SELECT statement using SQLExecute or SQLExecDirect. This opens a cursor on the statement. You then use **SQLFetch** or **SQLExtendedFetch** to fetch rows through the cursor. When an application free the statement using **SQLFreeHandle** it closes the cursor.

To fetch values from a cursor, your application can use either **SQLBindCol** or **SQLGetData**. If you use **SQLBindCol**, values are automatically retrieved on each fetch. If you use **SQLGetData**, you must call it for each column after each fetch.

SQLGetData is used to fetch values in pieces for columns such as LONG VARCHAR or LONG BINARY. As an alternative, you can set the SQL_MAX_LENGTH statement attribute to a value large enough to hold the entire value for the column. The default value for SQL_ATTR_MAX_LENGTH is 256 kb.

☞ The following code fragment opens a cursor on a query and retrieves data through the cursor. Error checking has been omitted to make the example easier to read. The fragment is taken from a complete sample, which can be found at *Samples\ASA\ODBCSelect\odbcselect.cpp*.

```
SQLINTEGER cbDeptID = 0, cbDeptName = SQL_NTS, cbManagerID = 0;
SQLCHAR deptname[ DEPT_NAME_LEN ];
SQLSMALLINT deptID, managerID;
SQLHENV   env;
SQLHDBC   dbc;
SQLHSTMT stmt;
SQLRETURN retcode;
```

```
SQLAllocHandle( SQL_HANDLE_ENV, SQL_NULL_HANDLE, &env );
SQLSetEnvAttr( env,
            SQL_ATTR_ODBC_VERSION,
            (void*)SQL_OV_ODBC3, 0);
SQLAllocHandle( SQL_HANDLE_DBC, env, &dbc );
SQLConnect( dbc,
            (SQLCHAR*) "ASA 9.0 Sample", SQL_NTS,
            (SQLCHAR*) "DBA", SQL_NTS,
            (SQLCHAR*) "SQL", SQL_NTS );
SQLAllocHandle( SQL_HANDLE_STMT, dbc, &stmt );
SQLBindCol( stmt, 1,
            SQL_C_SSHORT, &deptID, 0, &cbDeptID);
SQLBindCol( stmt, 2,
            SQL_C_CHAR, deptname,
            sizeof(deptname), &cbDeptName);
SQLBindCol( stmt, 3,
            SQL_C_SSHORT, &managerID, 0, &cbManagerID);

SQLExecDirect( stmt, (SQLCHAR * )
"SELECT dept_id, dept_name, dept_head_id FROM DEPARTMENT "
            "ORDER BY dept_id", SQL_NTS );
while( ( retcode = SQLFetch( stmt ) ) != SQL_NO_DATA ){
   printf( "%d %20s %d\n", deptID, deptname, managerID );
}
SQLFreeHandle( SQL_HANDLE_STMT, stmt );
SQLDisconnect( dbc );
SQLFreeHandle( SQL_HANDLE_DBC, dbc );
SQLFreeHandle( SQL_HANDLE_ENV, env );
```

The number of row positions you can fetch in a cursor is governed by the size of an integer. You can fetch rows numbered up to number 2147483646, which is one less than the value that can be held in an integer. When using negative numbers (rows from the end) you can fetch down to one more than the largest negative value that can be held in an integer.

Updating and deleting rows through a cursor

The Microsoft ODBC Programmer's Reference suggests that you use SELECT... FOR UPDATE to indicate that a query is updateable using positioned operations. You do not need to use the FOR UPDATE clause in Adaptive Server Anywhere: SELECT statements are automatically updateable as long as the following conditions are met:

♦ The underlying query supports updates.

That is to say, as long as a data modification statement on the columns in the result is meaningful, then positioned data modification statements can be carried out on the cursor.

The ANSI_UPDATE_CONSTRAINTS database option limits the type of queries that are updateable.

251

☞ For more information, see "ANSI_UPDATE_CONSTRAINTS option [compatibility]" [*ASA Database Administration Guide,* page 635].

♦ The cursor type supports updates.

If you are using a read-only cursor, you cannot update the result set.

ODBC provides two alternatives for carrying out positioned updates and deletes:

♦ Use the **SQLSetPos** function.

Depending on the parameters supplied (SQL_POSITION, SQL_REFRESH, SQL_UPDATE, SQL_DELETE) **SQLSetPos** sets the cursor position and allows an application to refresh data, or update, or delete data in the result set.

This is the method to use with Adaptive Server Anywhere.

♦ Send positioned UPDATE and DELETE statements using **SQLExecute**. This method should not be used with Adaptive Server Anywhere.

Using bookmarks

ODBC provides **bookmarks**, which are values used to identify rows in a cursor. Adaptive Server Anywhere supports bookmarks for all kinds of cursors except dynamic cursors.

Before ODBC 3.0, a database could specify only whether it supported bookmarks or not: there was no interface to provide this information for each cursor type. There was no way for a database server to indicate for what kind of cursor bookmarks were supported. For ODBC 2 applications, Adaptive Server Anywhere returns that it does support bookmarks. There is therefore nothing to prevent you from trying to use bookmarks with dynamic cursors; however, you should not use this combination.

Calling stored procedures

This section describes how to create and call stored procedures and process the results from an ODBC application.

☞ For a full description of stored procedures and triggers, see "Using Procedures, Triggers, and Batches" [*ASA SQL User's Guide,* page 655].

Procedures and result sets

There are two types of procedures: those that return result sets and those that do not. You can use **SQLNumResultCols** to tell the difference: the number of result columns is zero if the procedure does not return a result set. If there is a result set, you can fetch the values using **SQLFetch** or **SQLExtendedFetch** just like any other cursor.

Parameters to procedures should be passed using parameter markers (question marks). Use **SQLBindParameter** to assign a storage area for each parameter marker, whether it is an INPUT, OUTPUT, or INOUT parameter.

To handle multiple result sets, ODBC must describe the currently executing cursor, not the procedure-defined result set. Therefore, ODBC does not always describe column names as defined in the RESULT clause of the stored procedure definition. To avoid this problem, you can use column aliases in your procedure result set cursor.

Example

This example creates and calls a procedure that does not return a result set. The procedure takes one INOUT parameter, and increments its value. In the example, the variable **num_col** will have the value zero, since the procedure does not return a result set. Error checking has been omitted to make the example easier to read.

```
HDBC dbc;
HSTMT stmt;
long i;
SWORD num_col;

/* Create a procedure */
SQLAllocStmt( dbc, &stmt );
SQLExecDirect( stmt,
      "CREATE PROCEDURE Increment( INOUT a INT )" \
      " BEGIN" \
         " SET a = a + 1" \
      " END", SQL_NTS );

/* Call the procedure to increment 'i' */
i = 1;
SQLBindParameter( stmt, 1, SQL_C_LONG, SQL_INTEGER, 0,
            0, &i, NULL );
SQLExecDirect( stmt, "CALL Increment( ? )",
            SQL_NTS );
SQLNumResultCols( stmt, &num_col );
do_something( i );
```

This example calls a procedure that returns a result set. In the example, the variable **num_col** will have the value 2 since the procedure returns a result set with two columns. Again, error checking has been omitted to make the example easier to read.

```
HDBC dbc;
HSTMT stmt;
SWORD num_col;
RETCODE retcode;
char emp_id[ 10 ];
char emp_lname[ 20 ];

/* Create the procedure */
SQLExecDirect( stmt,
    "CREATE PROCEDURE employees()" \
    " RESULT( emp_id CHAR(10), emp_lname CHAR(20))"\
    " BEGIN" \
    " SELECT emp_id, emp_lname FROM employee" \
    " END", SQL_NTS );

/* Call the procedure - print the results */
SQLExecDirect( stmt, "CALL employees()", SQL_NTS );
SQLNumResultCols( stmt, &num_col );
SQLBindCol( stmt, 1, SQL_C_CHAR, &emp_id,
            sizeof(emp_id), NULL );
SQLBindCol( stmt, 2, SQL_C_CHAR, &emp_lname,
            sizeof(emp_lname), NULL );

for( ;; ) {
   retcode = SQLFetch( stmt );
   if( retcode == SQL_NO_DATA_FOUND ) {
      retcode = SQLMoreResults( stmt );
   if( retcode == SQL_NO_DATA_FOUND ) break;
}    else {
      do_something( emp_id, emp_lname );
   }
}
```

Handling errors

Errors in ODBC are reported using the return value from each of the ODBC function calls and either the **SQLError** function or the **SQLGetDiagRec** function. The **SQLError** function was used in ODBC versions up to, but not including, version 3. As of version 3 the **SQLError** function has been deprecated and replaced by the **SQLGetDiagRec** function.

Every ODBC function returns a **SQLRETURN**, which is one of the following status codes:

Status code	Description
SQL_SUCCESS	No error.
SQL_SUCCESS_WITH_-INFO	The function completed, but a call to **SQLError** will indicate a warning.
	The most common case for this status is that a value being returned is too long for the buffer provided by the application.
SQL_ERROR	The function did not complete because of an error. Call **SQLError** to get more information on the problem.
SQL_INVALID_HANDLE	An invalid environment, connection, or statement handle was passed as a parameter.
	This often happens if a handle is used after it has been freed, or if the handle is the null pointer.
SQL_NO_DATA_FOUND	There is no information available.
	The most common use for this status is when fetching from a cursor; it indicates that there are no more rows in the cursor.
SQL_NEED_DATA	Data is needed for a parameter.
	This is an advanced feature described in the ODBC SDK documentation under **SQLParamData** and **SQLPutData**.

Every environment, connection, and statement handle can have one or more errors or warnings associated with it. Each call to **SQLError** or **SQLGetDiagRec** returns the information for one error and removes the information for that error. If you do not call **SQLError** or **SQLGetDiagRec**

to remove all errors, the errors are removed on the next function call that passes the same handle as a parameter.

Each call to **SQLError** passes three handles for an environment, connection, and statement. The first call uses SQL_NULL_HSTMT to get the error associated with a connection. Similarly, a call with both SQL_NULL_DBC and SQL_NULL_HSTMT get any error associated with the environment handle.

Each call to **SQLGetDiagRec** can pass either an environment, connection or statement handle. The first call passes in a handle of type SQL_HANDLE_DBC to get the error associated with a connection. The second call passes in a handle of type SQL_HANDLE_STMT to get the error associated with the statement that was just executed.

SQLError and **SQLGetDiagRec** return SQL_SUCCESS if there is an error to report (*not* SQL_ERROR), and SQL_NO_DATA_FOUND if there are no more errors to report.

Example 1   The following code fragment uses **SQLError** and return codes:

```
/* Declare required variables */
SQLHDBC dbc;
SQLHSTMT stmt;
SQLRETURN retcode;
UCHAR errmsg[100];
/* code omitted here */
retcode = SQLAllocHandle(SQL_HANDLE_STMT, dbc, &stmt );
if( retcode == SQL_ERROR ){
   SQLError( env, dbc, SQL_NULL_HSTMT, NULL, NULL,
         errmsg, sizeof(errmsg), NULL );
   /* Assume that print_error is defined */
   print_error( "Allocation failed", errmsg );
   return;
}

/* Delete items for order 2015 */
retcode = SQLExecDirect( stmt,
      "delete from sales_order_items where id=2015",
      SQL_NTS );
if( retcode == SQL_ERROR ) {
   SQLError( env, dbc, stmt, NULL, NULL,
         errmsg, sizeof(errmsg), NULL );
   /* Assume that print_error is defined */
   print_error( "Failed to delete items", errmsg );
   return;
}
```

Example 2   The following code fragment uses **SQLGetDiagRec** and return codes:

```
/* Declare required variables */
SQLHDBC dbc;
SQLHSTMT stmt;
SQLRETURN retcode;
SQLSMALLINT errmsglen;
SQLINTEGER errnative;
UCHAR errmsg[255];
UCHAR errstate[5];
/* code omitted here */
retcode = SQLAllocHandle(SQL_HANDLE_STMT, dbc, &stmt );
if( retcode == SQL_ERROR ){
   SQLGetDiagRec(SQL_HANDLE_DBC, dbc, 1, errstate,
       &errnative, errmsg, sizeof(errmsg), &errmsglen);
    /* Assume that print_error is defined */
   print_error( "Allocation failed", errstate,
errnative, errmsg );
   return;

}
/* Delete items for order 2015 */
retcode = SQLExecDirect( stmt,
      "delete from sales_order_items where id=2015",
      SQL_NTS );
if( retcode == SQL_ERROR ) {
   SQLGetDiagRec(SQL_HANDLE_STMT, stmt, recnum,
errstate,
      &errnative, errmsg, sizeof(errmsg), &errmsglen);
   /* Assume that print_error is defined */
   print_error("Failed to delete items", errstate,
errnative, errmsg );
   return;
}
```

The Database Tools Interface

About this chapter

This chapter describes how to use the database tools library that is provided with Adaptive Server Anywhere to add database management features to C or C++ applications.

Contents

Introduction to the database tools interface

Sybase Adaptive Server Anywhere includes Sybase Central and a set of utilities for managing databases. These database management utilities carry out tasks such as backing up databases, creating databases, translating transaction logs to SQL, and so on.

Supported platforms

All the database management utilities use a shared library called the database tools library. It is supplied for Windows operating systems and for UNIX. The name of this library is *dbtool9.dll* for Windows, and *libdbtool9.so* (non-threaded) or *libdbtool9_r.so* (threaded) for UNIX.

> **Non-threaded library for UNIX deprecated**
> The non-threaded library for UNIX is deprecated: it is fully supported in the current software but will not be supported in the next major release of SQL Anywhere Studio.

You can develop your own database management utilities or incorporate database management features into your applications by calling the database tools library. This chapter describes the interface to the database tools library. In this chapter, we assume you are familiar with how to call DLLs from the development environment you are using.

The database tools library has functions, or entry points, for each of the database management utilities. In addition, functions must be called before use of other database tools functions and when you have finished using other database tools functions.

Windows CE

The *dbtool9.dll* library is supplied for Windows CE, but includes only entry points for DBToolsInit, DBToolsFini, DBRemoteSQL, and DBSynchronizeLog. Other tools are not provided for Windows CE.

The dbtools.h header file

The dbtools header file included with Adaptive Server Anywhere lists the entry points to the DBTools library and also the structures used to pass information to and from the library. The *dbtools.h* file is installed into the *h* subdirectory under your installation directory. You should consult the *dbtools.h* file for the latest information about the entry points and structure members.

The *dbtools.h* header file includes two other files:

♦ **sqlca.h** This is included for resolution of various macros, not for the SQLCA itself.

♦ **dllapi.h** Defines preprocessor macros for operating-system dependent and language-dependent macros.

Also, the *sqldef.h* header file includes error return values.

Using the database tools interface

This section provides an overview of how to develop applications that use the DBTools interface for managing databases.

Using the import libraries

In order to use the DBTools functions, you must link your application against a DBTools **import library** which contains the required function definitions.

Supported platforms

Import libraries are compiler-specific and are supplied for Windows operating systems with the exception of Windows CE. Import libraries for the DBTools interface are provided with Adaptive Server Anywhere, and can be found in the *lib* subdirectory of each operating system's directory, under your installation directory. The provided DBTools import libraries are as follows:

Compiler	Library
Watcom	*win32\dbtlstw.lib*
Microsoft	*win32\dbtlstM.lib*
Borland	*win32\dbtlstB.lib*

Starting and finishing the DBTools library

Before using any other DBTools functions, you must call DBToolsInit. When you are finished using the DBTools DLL, you must call DBToolsFini.

The primary purpose of the DBToolsInit and DBToolsFini functions is to allow the DBTools DLL to load the Adaptive Server Anywhere language DLL. The language DLL contains localized versions of all error messages and prompts that DBTools uses internally. If DBToolsFini is not called, the reference count of the language DLL is not decremented and it will not be unloaded, so be careful to ensure there is a matched pair of DBToolsInit/DBToolsFini calls.

The following code fragment illustrates how to initialize and clean up DBTools:

```
// Declarations
a_dbtools_info   info;
short            ret;

//Initialize the a_dbtools_info structure
memset( &info, 0, sizeof( a_dbtools_info) );
info.errorrtn = (MSG_CALLBACK)MyErrorRtn;

// initialize DBTools
ret = DBToolsInit( &info );
if( ret != EXIT_OKAY ) {
   // DLL initialization failed
   ...
}
// call some DBTools routines . . .
...
// cleanup the DBTools dll
DBToolsFini( &info );
```

Calling the DBTools functions

All the tools are run by first filling out a structure, and then calling a function (or **entry point**) in the DBTools DLL. Each entry point takes a pointer to a single structure as argument.

The following example shows how to use the DBBackup function on a Windows operating system.

```
// Initialize the structure
a_backup_db backup_info;
memset( &backup_info, 0, sizeof( backup_info ) );

// Fill out the structure
backup_info.version = DB_TOOLS_VERSION_NUMBER;
backup_info.output_dir = "C:\\BACKUP";
backup_info.connectparms ="uid=DBA;pwd=SQL;dbf=asademo.db";

backup_info.confirmrtn = (MSG_CALLBACK) ConfirmRtn ;
backup_info.errorrtn = (MSG_CALLBACK) ErrorRtn ;
backup_info.msgrtn = (MSG_CALLBACK) MessageRtn ;
backup_info.statusrtn = (MSG_CALLBACK) StatusRtn ;
backup_info.backup_database = TRUE;

// start the backup
DBBackup( &backup_info );
```

☞ For information about the members of the DBTools structures, see

Software component return codes

All database tools are provided as entry points in a DLL. These entry points use the following return codes:

Code	Explanation
0	Success
1	General failure
2	Invalid file format
3	File not found, unable to open
4	Out of memory
5	Terminated by the user
6	Failed communications
7	Missing a required database name
8	Client/server protocol mismatch
9	Unable to connect to the database server
10	Database server not running
11	Database server not found
254	Reached stop time
255	Invalid parameters on the command-line

Using callback functions

Several elements in DBTools structures are of type MSG_CALLBACK. These are pointers to callback functions.

Uses of callback functions

Callback functions allow DBTools functions to return control of operation to the user's calling application. The DBTools library uses callback functions to handle messages sent to the user by the DBTools functions for four purposes:

♦ **Confirmation** Called when an action needs to be confirmed by the user. For example, if the backup directory does not exist, the tools DLL asks if it needs to be created.

♦ **Error message** Called to handle a message when an error occurs, such

as when an operation is out of disk space.

♦ **Information message**   Called for the tools to display some message to the user (such as the name of the current table being backed up).

♦ **Status information**   Called for the tools to display the status of an operation (such as the percentage done when unloading a table).

Assigning a callback function to a structure

You can directly assign a callback routine to the structure. The following statement is an example using a backup structure:

```
backup_info.errorrtn = (MSG_CALLBACK) MyFunction
```

MSG_CALLBACK is defined in the *dllapi.h* header file supplied with Adaptive Server Anywhere. Tools routines can call back to the Calling application with messages that should appear in the appropriate user interface, whether that be a windowing environment, standard output on a character-based system, or other user interface.

Confirmation callback function example

The following example confirmation routine asks the user to answer YES or NO to a prompt and returns the user's selection:

```
extern short _callback ConfirmRtn(
    char * question )
{
   int ret;
   if( question != NULL ) {
      ret = MessageBox( HwndParent, question,
      "Confirm", MB_ICONEXCLAMTION|MB_YESNO );
   }
   return( 0 );
}
```

Error callback function example

The following is an example of an error message handling routine, which displays the error message in a message box.

```
extern short _callback ErrorRtn(
    char * errorstr )
{
   if( errorstr != NULL ) {
      ret = MessageBox( HwndParent, errorstr,
      "Backup Error", MB_ICONSTOP|MB_OK );
   }
   return( 0 );
}
```

Message callback function example

A common implementation of a message callback function outputs the message to the screen:

```
extern short _callback MessageRtn(
    char * errorstr )
{
    if( messagestr != NULL ) {
    OutputMessageToWindow( messagestr );
    }
    return( 0 );
}
```

Status callback function
example

A status callback routine is called when the tools needs to display the status of an operation (like the percentage done unloading a table). Again, a common implementation would just output the message to the screen:

```
extern short _callback StatusRtn(
    char * statusstr )
{
    if( statusstr == NULL ) {
        return FALSE;
    }
    OutputMessageToWindow( statustr );
    return TRUE;
}
```

Version numbers and compatibility

Each structure has a member that indicates the version number. You should use this version member to hold the version of the DBTools library that your application was developed against. The current version of the DBTools library is included as the constant in the *dbtools.h* header file.

❖ **To assign the current version number to a structure**

1. Assign the version constant to the version member of the structure before calling the DBTools function. The following line assigns the current version to a backup structure:

   ```
   backup_info.version = DB_TOOLS_VERSION_NUMBER;
   ```

Compatibility

The version number allows your application to continue working against newer versions of the DBTools library. The DBTools functions use the version number supplied by your application to allow the application to work, even if new members have been added to the DBTools structure.

Applications will not work against older versions of the DBTools library.

Using bit fields

Many of the DBTools structures use bit fields to hold Boolean information in a compact manner. For example, the backup structure includes the following

bit fields:

```
a_bit_field      backup_database   : 1;
a_bit_field      backup_logfile   : 1;
a_bit_field      backup_writefile: 1;
a_bit_field      no_confirm   : 1;
a_bit_field      quiet      : 1;
a_bit_field      rename_log   : 1;
a_bit_field      truncate_log : 1;
a_bit_field      rename_local_log: 1;
a_bit_field  server_backup   : 1;
```

Each bit field is one bit long, indicated by the 1 to the right of the colon in the structure declaration. The specific data type used depends on the value assigned to **a_bit_field**, which is set at the top of *dbtools.h*, and is operating system-dependent.

You assign an integer value of 0 or 1 to a bit field to pass Boolean information in the structure.

A DBTools example

You can find this sample and instructions for compiling it in the *Samples\ASA\DBTools* subdirectory of your SQL Anywhere directory. The sample program itself is *Samples\ASA\DBTools\main.cpp*. The sample illustrates how to use the DBTools library to carry out a backup of a database.

```
#define WINNT

#include <stdio.h>
#include "windows.h"
#include "string.h"
#include "dbtools.h"
extern short _callback ConfirmCallBack( char * str )
{
   if( MessageBox( NULL, str, "Backup",
       MB_YESNO|MB_ICONQUESTION ) == IDYES )
   {
      return 1;
   }
   return 0;
}
```

```
extern short _callback MessageCallBack( char * str )
{
   if( str != NULL )
   {
      fprintf( stdout, "%s", str );
      fprintf( stdout, "\n" );
      fflush( stdout );
   }
   return 0;
}
extern short _callback StatusCallBack( char * str )
{
   if( str != NULL )
   {
      fprintf( stdout, "%s", str );
      fprintf( stdout, "\n" );
      fflush( stdout );
   }
   return 0;
}
extern short _callback ErrorCallBack( char * str )
{
   if( str != NULL )
   {
      fprintf( stdout, "%s", str );
      fprintf( stdout, "\n" );
      fflush( stdout );
   }
   return 0;
}
typedef void (CALLBACK *DBTOOLSPROC)( void * );

// Main entry point into the program.
int main( int argc, char * argv[] )
{
   a_dbtools_info      dbt_info;
   a_backup_db       backup_info;
   char        dir_name[ _MAX_PATH + 1];
   char        connect[ 256 ];
   HINSTANCE      hinst;
   DBTOOLSPROC       dbbackup;
   DBTOOLSPROC       dbtoolsinit;
   DBTOOLSPROC       dbtoolsfini;

// Always initialize to 0 so new versions
   // of the structure will be compatible.
   memset( &backup_info, 0, sizeof( a_backup_db ) );
   backup_info.version = DB_TOOLS_VERSION_NUMBER;
   backup_info.quiet = 0;
   backup_info.no_confirm = 0;
   backup_info.confirmrtn = (MSG_CALLBACK)ConfirmCallBack;
   backup_info.errorrtn = (MSG_CALLBACK)ErrorCallBack;
   backup_info.msgrtn = (MSG_CALLBACK)MessageCallBack;
   backup_info.statusrtn = (MSG_CALLBACK)StatusCallBack;
```

```
if( argc > 1 )
    {
        strncpy( dir_name, argv[1], _MAX_PATH );
    }
    else
    {
        // DBTools does not expect (or like) a trailing slash
        strcpy( dir_name, "c:\\temp" );
    }
    backup_info.output_dir = dir_name;
if( argc > 2 )
    {
        strncpy( connect, argv[2], 255 );
    }
    else
    {
        strcpy( connect, "DSN=ASA 9.0 Sample" );
    }
    backup_info.connectparms = connect;
    backup_info.startline = "";
    backup_info.quiet = 0;
    backup_info.no_confirm = 0;
    backup_info.backup_database = 1;
    backup_info.backup_logfile = 1;
    backup_info.rename_log = 0;
    backup_info.truncate_log = 0;

hinst = LoadLibrary( "dbtool9.dll" );
    if( hinst == NULL )
    {
        // Failed
        return 0;
    }
    dbt_info.errorrtn = (MSG_CALLBACK)ErrorCallBack;
    dbbackup = (DBTOOLSPROC) GetProcAddress( (HMODULE)hinst,
        "_DBBackup@4" );
    dbtoolsinit = (DBTOOLSPROC) GetProcAddress( (HMODULE)hinst,
        "_DBToolsInit@4" );
    dbtoolsfini = (DBTOOLSPROC) GetProcAddress( (HMODULE)hinst,
        "_DBToolsFini@4" );
    (*dbtoolsinit)( &dbt_info );
    (*dbbackup)( &backup_info );
    (*dbtoolsfini)( &dbt_info );
    FreeLibrary( hinst );
    return 0;
}
```

DBTools functions

This section describes the functions available in the DBTools library. The functions are listed alphabetically.

DBBackup function

Function	Database backup function. This function is used by the *dbbackup* command-line utility.
Prototype	short **DBBackup (** const a_backup_db * *backup-db* **);**
Parameters	

Parameter	Description
backup-db	Pointer to "a_backup_db structure" on page 280

Return value	A return code, as listed in "Software component return codes" on page 263.
Usage	The DBBackup function manages all database backup tasks.
	☞ For descriptions of these tasks, see "The Backup utility" [*ASA Database Administration Guide,* page 498].
See also	"a_backup_db structure" on page 280

DBChangeLogName function

Function	Changes the name of the transaction log file. This function is used by the *dblog* command-line utility.
Prototype	short **DBChangeLogName (** const a_change_log * *change-log* **);**
Parameters	

Parameter	Description
change-log	Pointer to "a_change_log structure" on page 282

Return value	A return code, as listed in "Software component return codes" on page 263.
Usage	The -t option of the *dblog* command-line utility changes the name of the transaction log. DBChangeLogName provides a programmatic interface to this function.
	☞ For descriptions of the *dblog* utility, see "The Transaction Log utility" [*ASA Database Administration Guide,* page 580].
See also	"a_change_log structure" on page 282

DBChangeWriteFile function

Function	Changes a write file to refer to another database file. This function is used by the *dbwrite* command-line utility when the -d option is applied.
Prototype	short **DBChangeWriteFile (** const a_writefile * *writefile* **);**
Parameters	

Parameter	Description
writefile	Pointer to "a_writefile structure" on page 322

Return value	A return code, as listed in "Software component return codes" on page 263.
Usage	☞ For information about the Write File utility and its features, see "The Write File utility (deprecated)" [*ASA Database Administration Guide,* page 609].
See also	"DBCreateWriteFile function" on page 271
	"DBStatusWriteFile function" on page 274
	"a_writefile structure" on page 322

DBCollate function

Function	Extracts a collation sequence from a database.
Prototype	short **DBCollate (** const a_db_collation * *db-collation* **);**
Parameters	

Parameter	Description
db-collation	Pointer to "a_db_collation structure" on page 289

Return value	A return code, as listed in "Software component return codes" on page 263.
Usage	☞ For information about the collation utility and its features, see "The Collation utility" [*ASA Database Administration Guide,* page 503]
See also	"a_db_collation structure" on page 289

DBCompress function

Function	Compresses a database file. This function is used by the *dbshrink* command-line utility.
Prototype	short **DBCompress (** const a_compress_db * *compress-db* **);**
Parameters	

Parameter	Description
compress-db	Pointer to "a_compress_db structure" on page 284

Return value   A return code, as listed in "Software component return codes" on page 263.

Usage   ☞ For information about the Compression utility and its features, see "The Compression utility (deprecated)" [*ASA Database Administration Guide,* page 507].

See also   "a_compress_db structure" on page 284

DBCreate function

Function   Creates a database. This function is used by the *dbinit* command-line utility.

Prototype   short **DBCreate (** const a_create_db * *create-db* **);**

Parameters

Parameter	Description
create-db	Pointer to "a_create_db structure" on page 286

Return value   A return code, as listed in "Software component return codes" on page 263.

Usage   ☞ For information about the initialization utility, see "The Initialization utility" [*ASA Database Administration Guide,* page 530].

See also   "a_create_db structure" on page 286

DBCreateWriteFile function

Function   Creates a write file. This function is used by the *dbwrite* command-line utility when the -c option is applied.

Prototype   short **DBCreateWriteFile (** const a_writefile * *writefile* **);**

Parameters

Parameter	Description
writefile	Pointer to "a_writefile structure" on page 322

Return value   A return code, as listed in "Software component return codes" on page 263.

Usage   ☞ For information about the Write File utility and its features, see "The Write File utility (deprecated)" [*ASA Database Administration Guide,* page 609].

See also   "DBChangeWriteFile function" on page 270

                 "DBStatusWriteFile function" on page 274

DBErase function

Function	Erases a database file and/or transaction log file. This function is used by the *dberase* command-line utility.
Prototype	short **DBErase (** const an_erase_db * *erase-db* **);**
Parameters	

Parameter	Description
erase-db	Pointer to "an_erase_db structure" on page 295

Return value	A return code, as listed in "Software component return codes" on page 263.
Usage	☞ For information about the Erase utility and its features, see "The Erase utility" [*ASA Database Administration Guide,* page 521].
See also	"an_erase_db structure" on page 295

DBExpand function

Function	Uncompresses a database file. This function is used by the *dbexpand* command-line utility.
Prototype	short **DBExpand (** const an_expand_db * *expand-db* **);**
Parameters	

Parameter	Description
expand_db	Pointer to "an_expand_db structure" on page 296

Return value	A return code, as listed in "Software component return codes" on page 263.
Usage	☞ For information about the Uncompression utility and its features, see "The Uncompression utility (deprecated)" [*ASA Database Administration Guide,* page 585].
See also	"an_expand_db structure" on page 296

DBInfo function

Function	Returns information about a database file. This function is used by the *dbinfo* command-line utility.
Prototype	short **DBInfo (** const a_db_info * *db-info* **);**
Parameters	

Parameter	Description
db-info	Pointer to "a_db_info structure" on page 291

Return value   A return code, as listed in "Software component return codes" on page 263.

Usage   ☞ For information about the Information utility and its features, see "The Information utility" [*ASA Database Administration Guide,* page 528].

See also   "DBInfoDump function" on page 273

"DBInfoFree function" on page 273

"a_db_info structure" on page 291

DBInfoDump function

Function   Returns information about a database file. This function is used by the *dbinfo* command-line utility when the -u option is used.

Prototype   short **DBInfoDump (** const a_db_info * *db-info* **);**

Parameters

Parameter	Description
db-info	Pointer to "a_db_info structure" on page 291

Return value   A return code, as listed in "Software component return codes" on page 263.

Usage   ☞ For information about the Information utility and its features, see "The Information utility" [*ASA Database Administration Guide,* page 528].

See also   "DBInfo function" on page 272

"DBInfoFree function" on page 273

"a_db_info structure" on page 291

DBInfoFree function

Function   Called to free resources after the DBInfoDump function is called.

Prototype   short **DBInfoFree (** const a_db_info * *db-info* **);**

Parameters

Parameter	Description
db-info	Pointer to "a_db_info structure" on page 291

Return value   A return code, as listed in "Software component return codes" on page 263.

See also	

DBLicense function

| Function | Called to modify or report the licensing information of the database server. |
| Prototype | short **DBLicense (** const a_db_lic_info * *db-lic-info* **);** |

Parameters

Parameter	Description
db-lic-info	Pointer to

Return value	A return code, as listed in .
Usage	
See also	

DBStatusWriteFile function

| Function | Gets the status of a write file. This function is used by the *dbwrite* command-line utility when the -s option is applied. |
| Prototype | short **DBStatusWriteFile (** const a_writefile * *writefile* **);** |

Parameters

Parameter	Description
writefile	Pointer to

Return value	A return code, as listed in .
Usage	
See also	

DBSynchronizeLog function

Function Synchronize a database with a MobiLink synchronization server.

Prototype short **DBSynchronizeLog(** const a _sync_db * *sync-db* **);**

Parameters

Parameter	Description
sync-db	Pointer to "a_sync_db structure" on page 298

Return value A return code, as listed in "Software component return codes" on page 263.

Usage ☞ For information about the features you can access, see "Initiating synchronization" [*MobiLink Clients,* page 78].

See also "DBTools Interface for dbmlsync" [*MobiLink Clients,* page 267]

DBToolsFini function

Function Decrements the counter and frees resources when an application is finished with the DBTools library.

Prototype short **DBToolsFini (** const a_dbtools_info * *dbtools-info* **);**

Parameters

Parameter	Description
dbtools-info	Pointer to "a_dbtools_info structure" on page 294

Return value A return code, as listed in "Software component return codes" on page 263.

Usage The DBToolsFini function must be called at the end of any application that uses the DBTools interface. Failure to do so can lead to lost memory resources.

See also "DBToolsInit function" on page 275

"a_dbtools_info structure" on page 294

DBToolsInit function

Function Prepares the DBTools library for use.

Prototype short **DBToolsInit(** const a_dbtools_info * *dbtools-info* **);**

Parameters

Parameter	Description
dbtools-info	Pointer to "a_dbtools_info structure" on page 294

Return value
A return code, as listed in "Software component return codes" on page 263.

Usage
The primary purpose of the DBToolsInit function is to load the Adaptive Server Anywhere language DLL. The language DLL contains localized versions of error messages and prompts that DBTools uses internally.

The DBToolsInit function must be called at the start of any application that uses the DBTools interface, before any other DBTools functions.

Example
♦ The following code sample illustrates how to initialize and clean up DBTools:

```
a_dbtools_info   info;
short            ret;

memset( &info, 0, sizeof( a_dbtools_info) );
info.errorrtn = (MSG_CALLBACK)MakeProcInstance(
        (FARPROC)MyErrorRtn, hInst );

// initialize DBTools
ret = DBToolsInit( &info );
if( ret != EXIT_OKAY ) {
      // DLL initialization failed
      ...
   }
   // call some DBTools routines . . .
   ...
   // cleanup the DBTools dll
   DBToolsFini( &info );
```

See also
"DBToolsFini function" on page 275

"a_dbtools_info structure" on page 294

DBToolsVersion function

Function
Returns the version number of the DBTools library.

Prototype
short **DBToolsVersion (** void **);**

Return value
A short integer indicating the version number of the DBTools library.

Usage
Use the DBToolsVersion function to check that the DBTools library is not older than one against which your application is developed. While applications can run against newer versions of DBTools, they cannot run against older versions.

See also
"Version numbers and compatibility" on page 265

DBTranslateLog function

Function	Translates a transaction log file to SQL. This function is used by the *dbtran* command-line utility.
Prototype	short **DBTranslateLog (** const a_translate_log * *translate-log* **);**
Parameters	

Parameter	Description
translate-log	Pointer to "a_translate_log structure" on page 308

Return value	A return code, as listed in "Software component return codes" on page 263.
Usage	☞ For information about the log translation utility, see "The Log Translation utility" [*ASA Database Administration Guide,* page 556].
See also	"a_translate_log structure" on page 308

DBTruncateLog function

Function	Truncates a transaction log file. This function is used by the *dbbackup* command-line utility.
Prototype	short **DBTruncateLog (** const a_truncate_log * *truncate-log* **);**
Parameters	

Parameter	Description
truncate-log	Pointer to "a_truncate_log structure" on page 313

Return value	A return code, as listed in "Software component return codes" on page 263.
Usage	☞ For information about the backup utility, see "The Backup utility" [*ASA Database Administration Guide,* page 498]
See also	"a_truncate_log structure" on page 313

DBUnload function

Function	Unloads a database. This function is used by the *dbunload* command-line utility and also by the *dbxtract* utility for SQL Remote.
Prototype	short **DBUnload (** const an_unload_db * *unload-db* **);**
Parameters	

Parameter	Description
unload-db	Pointer to "an_unload_db structure" on page 314

Return value   A return code, as listed in "Software component return codes" on page 263.

Usage   ☞ For information about the Unload utility, see "The Unload utility" [*ASA Database Administration Guide,* page 588].

See also   "an_unload_db structure" on page 314

DBUpgrade function

Function   Upgrades a database file. This function is used by the *dbupgrade* command-line utility.

Prototype   short **DBUpgrade (** const an_upgrade_db * *upgrade-db* **);**

Parameters

Parameter	Description
upgrade-db	Pointer to "an_upgrade_db structure" on page 319

Return value   A return code, as listed in "Software component return codes" on page 263.

Usage   ☞ For information about the upgrade utility, see "The Upgrade utility" [*ASA Database Administration Guide,* page 599].

See also   "an_upgrade_db structure" on page 319

DBValidate function

Function   Validates all or part of a database. This function is used by the *dbvalid* command-line utility.

Prototype   short **DBValidate (** const a_validate_db * *validate-db* **);**

Parameters

Parameter	Description
validate-db	Pointer to "a_validate_db structure" on page 321

Return value   A return code, as listed in "Software component return codes" on page 263.

Usage   ☞ For information about the upgrade utility, see "The Validation utility" [*ASA Database Administration Guide,* page 604].

> ***Caution***
> *Validating a table or an entire database should be performed while no connections are making changes to the database; otherwise, spurious errors may be reported indicating some form of database corruption even though no corruption actually exists.*

See also   "a_validate_db structure" on page 321

DBTools structures

This section lists the structures that are used to exchange information with the DBTools library. The structures are listed alphabetically.

Many of the structure elements correspond to command-line options on the corresponding utility. For example, several structures have a member named quiet, which can take on values of 0 or 1. This member corresponds to the quiet operation (-q) command-line option used by many of the utilities.

a_backup_db structure

Function
Holds the information needed to carry out backup tasks using the DBTools library.

Syntax
```
typedef struct a_backup_db {
unsigned short version;
const char * output_dir;
const char * connectparms;
const char *  startline;
MSG_CALLBACK confirmrtn;
MSG_CALLBACK errorrtn;
MSG_CALLBACK msgrtn;
MSG_CALLBACK statusrtn;
a_bit_field backup_database : 1;
a_bit_field backup_logfile : 1;
a_bit_field backup_writefile : 1;
a_bit_field no_confirm : 1;
a_bit_field quiet : 1;
a_bit_field rename_log : 1;
a_bit_field truncate_log : 1;
a_bit_field rename_local_log: 1;
const char * hotlog_filename;
char backup_interrupted;
a_bit_field server_backup : 1;
} a_backup_db;
```

Members

Member	Description
Version	DBTools version number
output_dir	Path to the output directory. For example: `"c:\backup"`

Member	Description

connectparms	Parameters needed to connect to the database. They take the form of connection strings, such as the following:
	
	`"UID=DBA;PWD=SQL;DBF=c:\asa\`
	`   asademo.db"`
	
	The database server would be started by the connection string START parameter. The following is an example.
	
	`"START=d:\asa90\win32\dbeng9.exe"`
	
	A full example connection string including the START parameter:
	
	`"userid=dba;password=sql;dbf=d:\`
	`   asa90\asademo.db;start=d:\asa90\`
	`   win32\dbeng9.exe"`
	
	For the full range of connection string options, see "Connection parameters" [*ASA Database Administration Guide,* page 176]

startline	Obsolete: The startline field is no longer used. The database server would be started by the connection string START parameter. Refer to connectparams for details.
confirmrtn	Callback routine for confirming an action
errorrtn	Callback routine for handling an error message
msgrtn	Callback routine for handling an information message
statusrtn	Callback routine for handling a status message
backup_database	Backup the database file (1) or not (0)
backup_logfile	Backup the transaction log file (1) or not (0)
backup_writefile	Backup the database write file (1) or not (0), if a write file is being used
no_confirm	Operate with (0) or without (1) confirmation
quiet	Operate without printing messages (1), or print messages (0)
rename_log	Rename the transaction log
truncate_log	Delete the transaction log

Member	Description
rename_local_log	Rename the local backup of the transaction log
hotlog_filename	File name for the live backup file
backup_interrupted	Indicates that the operation was interrupted
server_backup	

See also

a_change_log structure

Function   Holds the information needed to carry out *dblog* tasks using the DBTools library.

Syntax
```
typedef struct a_change_log {
unsigned short version;
const char * dbname;
const char * logname;
MSG_CALLBACK errorrtn;
MSG_CALLBACK msgrtn;
a_bit_field query_only : 1;
a_bit_field quiet : 1;
a_bit_field mirrorname_present : 1;
a_bit_field change_mirrorname : 1;
a_bit_field change_logname : 1;
a_bit_field ignore_ltm_trunc : 1;
a_bit_field ignore_remote_trunc : 1;
a_bit_field set_generation_number : 1;
a_bit_field ignore_dbsync_trunc : 1;
const char * mirrorname;
unsigned short generation_number;
char * zap_current_offset;
char * zap_starting_offset;
char * encryption_key;
} a_change_log;
```

Members

Member	Description
version	DBTools version number
dbname	Database file name

Member	Description
logname	The name of the transaction log. If set to NULL, there is no log
errorrtn	Callback routine for handling an error message
msgrtn	Callback routine for handling an information message
query_only	If 1, just display the name of the transaction log. If 0, permit changing of the log name
quiet	Operate without printing messages (1), or print messages (0)
mirrorname_present	Set to 1. Indicates that the version of DBTools is recent enough to support the mirrorname field
change_mirrorname	If 1, permit changing of the log mirror name
change_logname	If 1, permit changing of the transaction log name
ignore_ltm_trunc	When using the Log Transfer Manager, performs the same function as the dbcc settrunc('ltm', 'gen_id', n) Replication Server function: For information on dbcc, see your Replication Server documentation
ignore_remote_trunc	For SQL Remote. Resets the offset kept for the purposes of the DELETE_OLD_LOGS option, allowing transaction logs to be deleted when they are no longer needed
set_generation_-number	When using the Log Transfer Manager, used after a backup is restored to set the generation number
ignore_dbsync_trunc	When using dbmlsync, resets the offset kept for the purposes of the DELETE_OLD_LOGS option, allowing transaction logs to be deleted when they are no longer needed
mirrorname	The new name of the transaction log mirror file
generation_number	The new generation number. Used together with set_generation_number
key_file	A file holding the encryption key

Member	Description
zap_current_offset	Change the current offset to the specified value. This is for use only in resetting a transaction log after an unload and reload to match *dbremote* or *dbmlsync* settings.
zap_starting_offset	Change the starting offset to the specified value. This is for use only in resetting a transaction log after an unload and reload to match *dbremote* or *dbmlsync* settings.
encryption_key	The encryption key for the database file.

See also   "DBChangeLogName function" on page 269

For more information on callback functions, see "Using callback functions" on page 263.

a_compress_db structure

Function   Holds the information needed to carry out database compression tasks using the DBTools library.

Syntax
```
typedef struct a_compress_db {
unsigned short version;
const char * dbname;
const char * compress_name;
MSG_CALLBACK errorrtn;
MSG_CALLBACK msgrtn;
MSG_CALLBACK statusrtn;
a_bit_field display_free_pages : 1;
a_bit_field quiet : 1;
a_bit_field record_unchanged : 1;
a_compress_stats *  stats;
MSG_CALLBACK confirmrtn;
a_bit_field noconfirm : 1;
const char *  encryption_key;
} a_compress_db;
```

Members

Member	Description
version	DBTools version number
dbname	The file name of the database to compress

Member	Description
compress_name	The file name of the compressed database
errorrtn	Callback routine for handling an error message
msgrtn	Callback routine for handling an information message
statusrtn	Callback routine for handling a status message
display_free_pages	Display the free page information.
quiet	Operate without printing messages (1), or print messages (0)
record_unchanged	Set to 1. Indicates that the a_compress_stats structure is recent enough to have an **unchanged** member
a_compress_stats	Pointer to a structure of type a_compress_stats. This is filled in if the member is not NULL and display_free_pages is not zero
confirmrtn	Callback routine for confirming an action
noconfirm	Operate with (0) or without (1) confirmation
encryption_key	The encryption key for the database file.

See also   "DBCompress function" on page 270

"a_compress_stats structure" on page 285

For more information on callback functions, see "Using callback functions" on page 263.

a_compress_stats structure

Function   Holds information describing compressed database file statistics.

Syntax
```
typedef struct a_compress_stats {
a_stats_line tables;
a_stats_line indices;
a_stats_line other;
a_stats_line free;
a_stats_line total;
a_sql_int32 free_pages;
a_sql_int32 unchanged;
} a_compress_stats;
```

Members

Member	Description
tables	Holds compression information regarding tables
indices	Holds compression information regarding indexes
other	Holds other compression information
free	Holds information regarding free space
total	Holds overall compression information
free_pages	Holds information regarding free pages
unchanged	The number of pages that the compression algorithm was unable to shrink

See also

a_create_db structure

Function
Holds the information needed to create a database using the DBTools library.

Syntax

```
typedef struct a_create_db {
unsigned short version;
const char * dbname;
const char * logname;
const char * startline;
unsigned short page_size;
const char * default_collation;
MSG_CALLBACK errorrtn;
MSG_CALLBACK msgrtn;
short database_version;
char verbose;
a_bit_field blank_pad : 2;
a_bit_field respect_case : 1;
a_bit_field encrypt : 1;
a_bit_field debug : 1;
a_bit_field  dbo_avail : 1;
a_bit_field mirrorname_present : 1;
a_bit_field avoid_view_collisions : 1;
short collation_id;
const char *   dbo_username;
const char * mirrorname;
a_bit_field java_classes : 1;
a_bit_field jconnect : 1;
const char * data_store_type;
const char *  encryption_key;
const char *  encryption_algorithm;
const char *  jdk_version;
a_bit_field respect_password_case : 1;
a_bit_field  checksum : 1;
} a_create_db;
```

Members

Member	Description
version	DBTools version number
dbname	Database file name
logname	New transaction log name

287

Member	Description
startline	The command-line used to start the database server. The following is an example start line: `"c:\asa\win32\dbeng9.exe"` The default start line is used if this member is NULL The following is the default START parameter: `"dbeng9 -gp <PAGE_SIZE> -C 10M"` (The -c 10M is added only when the page size is >=2048)
page_size	The page size of the database
default_collation	The collation for the database
errorrtn	Callback routine for handling an error message
msgrtn	Callback routine for handling an information message
database_version	The version number of the database
verbose	Run in verbose mode
blank_pad	Treat blanks as significant in string comparisons and hold index information to reflect this
respect_case	Make string comparisons case sensitive and hold index information to reflect this
encrypt	Encrypt the database
debug	Reserved
dbo_avail	Set to 1. The dbo user is available in this database
mirrorname_present	Set to 1. Indicates that the version of DBTools is recent enough to support the mirrorname field
avoid_view_-collisions	Omit the generation of Watcom SQL compatibility views SYS.SYSCOLUMNS and SYS.-SYSINDEXES
collation_id	Collation identifier
dbo_username	Obsolete
mirrorname	Transaction log mirror name
encryption_dllname	The DLL used to encrypt the database.

Member	Description
java_classes	Create a Java-enabled database.
jconnect	Include system procedures needed for jConnect
data_store_type	Reserved. Use NULL.
encryption_key	The encryption key for the database file.
encryption_algorithm	The encryption algorithm (**AES**).
jdk_version	One of the values for the *dbinit* -jdk option.
respect_password_-case	
checksum	

See also   "DBCreate function" on page 271

For more information on callback functions, see "Using callback functions" on page 263.

a_db_collation structure

Function   Holds the information needed to extract a collation sequence from a database using the DBTools library.

Syntax
```
typedef struct a_db_collation {
unsigned short version;
const char * connectparms;
const char * startline;
const char * collation_label;
const char * filename;
MSG_CALLBACK confirmrtn;
MSG_CALLBACK errorrtn;
MSG_CALLBACK msgrtn;
a_bit_field include_empty : 1;
a_bit_field hex_for_extended : 1;
a_bit_field replace : 1;
a_bit_field quiet : 1;
const char * input_filename;
} a_db_collation;
```

Members

Member	Description
version	DBTools version number

Member	Description
connectparms	Parameters needed to connect to the database. They take the form of connection strings, such as the following: `"UID=DBA;PWD=SQL;DBF=c:\asa\` `    asademo.db"` The database server would be started by the connection string START parameter. The following is an example. `"START=d:\asa90\win32\dbeng9.exe"` A full example connection string including the START parameter: `"userid=dba;password=sql;dbf=d:\` `    asa90\asademo.db;start=d:\asa90\` `    win32\dbeng9.exe"` For the full range of connection string options, see "Connection parameters" [*ASA Database Administration Guide,* page 176]
startline	Obsolete: The startline field is no longer used. The database server would be started by the connection string START parameter. Refer to connectparams for details.
confirmrtn	Callback routine for confirming an action
errorrtn	Callback routine for handling an error message
msgrtn	Callback routine for handling an information message
include_empty	Write empty mappings for gaps in the collations sequence
hex_for_extended	Use two-digit hexadecimal numbers to represent high-value characters
replace	Operate without confirming actions
quiet	Operate without messages
input_filename	Input collation definition

See also

"DBCollate function" on page 270

For more information on callback functions, see "Using callback functions" on page 263.

a_db_info structure

Function   Holds the information needed to return *dbinfo* information using the
                  DBTools library.

Syntax   typedef struct a_db_info {
                  unsigned short   version;
                  MSG_CALLBACK   errorrtn;
                  const char *   dbname;
                  unsigned short   dbbufsize;
                  char *   dbnamebuffer;
                  unsigned short   logbufsize;
                  char *   lognamebuffer;
                  unsigned short   wrtbufsize;
                  char *   wrtnamebuffer;
                  a_bit_field   quiet : 1;
                  a_bit_field   mirrorname_present : 1;
                  a_sysinfo   sysinfo;
                  a_sql_uint32   free_pages;
                  a_bit_field   compressed : 1;
                  const char *   connectparms;
                  const char *   startline;
                  MSG_CALLBACK   msgrtn;
                  MSG_CALLBACK   statusrtn;
                  a_bit_field   page_usage : 1;
                  a_table_info *   totals;
                  a_sql_uint32   file_size;
                  a_sql_uint32   unused_pages;
                  a_sql_uint32   other_pages;
                  unsigned short   mirrorbufsize;
                  char *   mirrornamebuffer;
                  char *   collationnamebuffer;
                  unsigned short   collationnamebufsize;
                  char *   classesversionbuffer;
                  unsigned short   classesversionbufsize;
                  a_bit_field   checksum : 1;
                  } a_db_info;

Members

Member	Description
version	DBTools version number
errortrn	Callback routine for handling an error message
dbname	Database file name

Member	Description
dbbufsize	The length of the dbnamebuffer member
dbnamebuffer	Database file name
logbufsize	The length of the lognamebuffer member
lognamebuffer	Transaction log file name
wrtbufsize	The length of the wrtnamebuffer member
wrtnamebuffer	The write file name
quiet	Operate without confirming messages
mirrorname_present	Set to 1. Indicates that the version of DBTools is recent enough to support the mirrorname field
sysinfo	Pointer to a_sysinfo structure
free_pages	Number of free pages
compressed	1 if compressed, otherwise 0

connectparms	Parameters needed to connect to the database. They take the form of connection strings, such as the following:
	
	`"UID=DBA;PWD=SQL;DBF=c:\asa\`
	`    asademo.db"`
	
	The database server would be started by the connection string START parameter. The following is an example.
	
	`"START=d:\asa90\win32\dbeng9.exe"`
	
	A full example connection string including the START parameter:
	
	`"userid=dba;password=sql;dbf=d:\`
	`    asa90\asademo.db;start=d:\asa90\`
	`    win32\dbeng9.exe"`
	
	For the full range of connection string options, see "Connection parameters" [*ASA Database Administration Guide,* page 176]

startline	Obsolete: The startline field is no longer used. The database server would be started by the connection string START parameter. Refer to connectparams for details.
msgrtn	Callback routine for handling an information message

Member	Description
statusrtn	Callback routine for handling a status message
page_usage	1 to report page usage statistics, otherwise 0
totals	Pointer to a_table_info structure
file_size	Size of database file
unused_pages	Number of unused pages
other_pages	Number of pages that are neither table nor index pages
mirrorbufsize	The length of the mirrornamebuffer member
mirrornamebuffer	The transaction log mirror name
collationnamebuffer	The database collation name and label (the maximum size is 128+1)
collationnamebuf-size	The length of the collationnamebuffer member
classesversionbuffer	The JDK version of the installed Java classes, such as 1.1.3, 1.1.8, 1.3, or an empty string if Java classes are not installed in the database (the maximum size is 10+1)
classesversionbuf-size	The length of the classesversionbuffer member
checksum	Database page checksums enabled if 1, disabled if 0

See also   "DBInfo function" on page 272

For more information on callback functions, see "Using callback functions" on page 263.

a_dblic_info structure

Function   Holds information containing licensing information. You must use this information only in a manner consistent with your license agreement.

293

Syntax	typedef struct a_dblic_info { unsigned short version; char * exename; char * username; char * compname; char * platform_str; a_sql_int32 nodecount; a_sql_int32 conncount; a_license_type type; MSG_CALLBACK errorrtn; MSG_CALLBACK msgrtn; a_bit_field quiet : 1; a_bit_field query_only : 1; } a_dblic_info;

Members

Member	Description
version	DBTools version number
exename	Executable name
username	User name for licensing
compname	Company name for licensing
platform_str	Operating system: WinNT or NLM or UNIX
nodecount	Number of nodes licensed.
conncount	Must be 1000000L
type	See *lictype.h* for values
errorrtn	Callback routine for handling an error message
msgrtn	Callback routine for handling an information message
quiet	Operate without printing messages (1), or print messages (0)
query_only	If 1, just display the license information. If 0, permit changing the information

a_dbtools_info structure

Function	Holds the information needed to start and finish working with the DBTools library.

Syntax	typedef struct a_dbtools_info { MSG_CALLBACK errorrtn; } a_dbtools_info;

Members

Member	Description
errorrtn	Callback routine for handling an error message

See also

For more information on callback functions, see .

an_erase_db structure

Function	Holds information needed to erase a database using the DBTools library.
Syntax	typedef struct an_erase_db { unsigned short version; const char * dbname; MSG_CALLBACK confirmrtn; MSG_CALLBACK errorrtn; MSG_CALLBACK msgrtn; a_bit_field quiet : 1; a_bit_field erase : 1; const char * encryption_key; } an_erase_db;

Members

Member	Description
version	DBTools version number
dbname	Database file name to erase
confirmrtn	Callback routine for confirming an action
errorrtn	Callback routine for handling an error message
msgrtn	Callback routine for handling an information message
quiet	Operate without printing messages (1), or print messages (0)
erase	Erase without confirmation (1) or with confirmation (0)

Member	Description
encryption_-key	The encryption key for the database file.

an_expand_db structure

Function   Holds information needed for database expansion using the DBTools library.

Syntax

```
typedef struct an_expand_db {
unsigned short version;
const char * compress_name;
const char * dbname;
MSG_CALLBACK errorrtn;
MSG_CALLBACK msgrtn;
MSG_CALLBACK statusrtn;
a_bit_field quiet : 1;
MSG_CALLBACK confirmrtn;
a_bit_field noconfirm : 1;
const char *  unused_field;
const char * encryption_key;
} an_expand_db;
```

Members

Member	Description
version	DBTools version number
compress_name	Name of compressed database file
dbname	Database file name
errorrtn	Callback routine for handling an error message
msgrtn	Callback routine for handling an information message
statusrtn	Callback routine for handling a status message
quiet	Operate without printing messages (1), or print messages (0)
confirmrtn	Callback routine for confirming an action
noconfirm	Operate with (0) or without (1) confirmation

Member	Description
key_file	A file holding the encryption key
encryption_key	The encryption key for the database file.

See also   "DBExpand function" on page 272

a_name structure

Function   Holds a linked list of names. This is used by other structures requiring lists of names.

Syntax   typedef struct a_name {
struct a_name * next;
char name[1];
} a_name, * p_name;

Members

Member	Description
next	Pointer to the next a_name structure in the list
name	The name
p_name	Pointer to the previous a_name structure

See also   "a_translate_log structure" on page 308

"a_validate_db structure" on page 321

"an_unload_db structure" on page 314

a_stats_line structure

Function   Holds information needed for database compression and expansion using the DBTools library.

Syntax   typedef struct a_stats_line {
a_sql_int32 pages;
a_sql_int32 bytes;
a_sql_int32 compressed_bytes;
} a_stats_line;

Members

Member	Description
pages	Number of pages
bytes	Number of bytes for uncompressed database
compressed_bytes	Number of bytes for compressed database

See also

a_sync_db structure

Function
Holds information needed for the *dbmlsync* utility using the DBTools library.

Syntax
```
typedef struct a_sync_db {
unsigned short   version;
char * connectparms;
char * publication;
const char * offline_dir;
char * extended_options;
char * script_full_path;
const char * include_scan_range;
const char * raw_file;
MSG_CALLBACK   confirmrtn;
MSG_CALLBACK   errorrtn;
MSG_CALLBACK   msgrtn;
MSG_CALLBACK   logrtn;

a_sql_uint32   debug_dump_size;
a_sql_uint32   dl_insert_width;
a_bit_field    verbose       : 1;
a_bit_field    debug         : 1;
a_bit_field    debug_dump_hex    : 1;
a_bit_field    debug_dump_char   : 1;
a_bit_field    debug_page_offsets : 1;
a_bit_field    use_hex_offsets   : 1;
a_bit_field    use_relative_offsets : 1;
a_bit_field    output_to_file    : 1;
a_bit_field    output_to_mobile_link : 1;
a_bit_field    dl_use_put      : 1;
a_bit_field    dl_use_upsert     : 1;
a_bit_field    kill_other_connections  : 1;
a_bit_field    retry_remote_behind  : 1;
a_bit_field    ignore_debug_interrupt  : 1;
```

```
SET_WINDOW_TITLE_CALLBACK set_window_title_rtn;
char *     default_window_title;
MSG_QUEUE_CALLBACK   msgqueuertn;
MSG_CALLBACK   progress_msg_rtn;
SET_PROGRESS_CALLBACK progress_index_rtn;
char **     argv;
char **     ce_argv;

a_bit_field     connectparms_allocated  : 1;
a_bit_field     entered_dialog     : 1;
a_bit_field     used_dialog_allocation  : 1;
a_bit_field     ignore_scheduling   : 1;
a_bit_field     ignore_hook_errors  : 1;
a_bit_field     changing_pwd     : 1;
a_bit_field     prompt_again     : 1;
a_bit_field     retry_remote_ahead   : 1;
a_bit_field     rename_log     : 1;
a_bit_field     hide_conn_str     : 1;
a_bit_field     hide_ml_pwd     : 1;
a_sql_uint32   dlg_launch_focus;
char * mlpassword;
char * new_mlpassword;
char * verify_mlpassword;

a_sql_uint32 pub_name_cnt;
char **     pub_name_list;
USAGE_CALLBACK   usage_rtn;
a_sql_uint32     log_size;
a_sql_uint32     hovering_frequency;
a_bit_short     ignore_hovering     : 1;
a_bit_short     verbose_upload     : 1;
a_bit_short     verbose_upload_data   : 1;
a_bit_short     verbose_download   : 1;
a_bit_short     verbose_download_data   : 1;
a_bit_short     autoclose     : 1;
a_bit_short     ping     : 1;
a_bit_short     _unused     : 9;
char * encryption_key;
a_syncpub * upload_defs;
char * log_file_name;
char * user_name;
```

```
a_bit_short    verbose_minimum    : 1;
a_bit_short    verbose_hook       : 1;
a_bit_short    verbose_row_data   : 1;
a_bit_short    verbose_row_cnts   : 1;
a_bit_short    verbose_option_info    : 1;
a_bit_short    strictly_ignore_trigger_ops : 1;
a_bit_short    _unused2     : 10;
a_sql_uint32   est_upld_row_cnt;
STATUS_CALLBACK    status_rtn;
MSG_CALLBACK    warningrtn;
char **    ce_reproc_argv;

a_bit_short    upload_only     : 1;
a_bit_short    download_only    : 1;
a_bit_short    allow_schema_change    : 1;
a_bit_short    dnld_gen_num    : 1;
a_bit_short    _unused3    :12;
const char *    apply_dnld_file;
const char *    create_dnld_file;
char *    sync_params;
const char *    dnld_file_extra;

COMServer *    com_server;
a_bit_short    trans_upload    : 1;
a_bit_short    continue_download    : 1;
a_sql_uint32   dnld_read_size;
a_sql_uint32   dnld_fail_len;
a_sql_uint32   upld_fail_len;
} a_sync_db;
```

Members

Member	Description
version	DBTools version number

Member	Description
connectparms	Parameters needed to connect to the database. They take the form of connection strings, such as the following:
	`"UID=DBA;PWD=SQL;DBF=c:\asa\` `    asademo.db"`
	The database server would be started by the connection string START parameter. The following is an example.
	`"START=d:\asa90\win32\dbeng9.exe"`
	A full example connection string including the START parameter:

	`"userid=dba;password=sql;dbf=d:\`
	`    asa90\asademo.db;start=d:\asa90\`
	`    win32\dbeng9.exe"`

	For the full range of connection string options, see "Connection parameters" [*ASA Database Administration Guide,* page 176]
publication	Deprecated; use NULL
offline_dir	Log directory, as specified on the command line after the switches
extended_options	Extended options, as specified with -e
script_full_path	Deprecated; use Null
include_scan_-range	Reserved; use NULL
raw_file	Reserved; use NULL
confirmrtn	Reserved; use NULL
errorrtn	Function to display error messages
msgrtn	Function to write messages to the user interface and, optionally, to the log file
logrtn	Function to write messages only to the log file
debug_dump_size	Reserved; use 0
dl_insert_width	Reserved; use 0
verbose	Deprecated; use 0
debug	Deprecated; use 0
debug_dump_hex	Deprecated; use 0

Member	Description
debug_dump_char	Deprecated; use 0
debug_page_-offsets	Deprecated; use 0
use_hex_offsets	Deprecated; use 0
use_relative_-offsets	Deprecated; use 0
output_to_file	Deprecated; use 0
output_to_-mobile_link	Deprecated; use 1
dl_use_put	Deprecated; use 0
dl_use_upsert	Deprecated; use 0
kill_other_-connections	TRUE if -d option is specified
retry_remote_-behind	TRUE if -r or -rb is specified
ignore_debug_-interrupt	Reserved—use 0
set_window_-title_rtn	Function to call to change the title of the dbmlsync window (Windows NT/2000/XP only)
default_window_-title	Name of the program to display in the window caption (for example, DBMLSync)

Member	Description
msgqueuertn	Function called by DBMLSync when it wants to sleep. The parameter specifies the desired sleep period in milliseconds. The function should return the following, as defined in dllapi.h .
	♦ MSGQ_SLEEP_THROUGH indicates that the routine slept for the requested number of milliseconds. In most cases this is the value you should return.
	♦ MSGQ_SHUTDOWN_REQUESTED indicates that you would like the synchronization to terminate as soon as possible.
	♦ MSGQ_SYNC_REQUESTED indicates that the routine slept for less than the requested number of milliseconds and that the next synchronization should begin immediately if a synchronization is not currently in progress.
progress_msg_rtn	Function to change the text in the status window, above the progress bar
progress_index_-rtn	Function to update the state of the progress bar
argv	argv array for this run; the last element of the array must be NULL
ce_argv	Reserved; use NULL
connectparms_-allocated	Reserved; use 0
entered_dialog	Reserved; use 0
used_dialog_-allocation	Reserved; use 0
ignore_scheduling	TRUE if -is was specified
ignore_hook_-errors	TRUE if -eh was specified
changing_pwd	TRUE if -mn was specified
prompt_again	Reserved—use 0
retry_remote_-ahead	TRUE if -ra was specified

Member	Description
rename_log	TRUE if -x was specified, in which case the log file is renamed and restarted
hide_conn_str	TRUE unless -vc was specified
hide_ml_pwd	TRUE unless -vp was specified
delay_ml_disconn	TRUE if -x was specified
dlg_launch_focus	Reserved; use 0
mlpassword	MobiLink password specified with -mp; NULL otherwise
new_mlpassword	New MobiLink password specified with -mn; NULL otherwise
verify_-mlpassword	Reserved; use NULL
pub_name_cnt	Deprecated; use 0
pub_name_list	Deprecated; use NULL
usage_rtn	Reserved; use NULL
log_size	Log size in bytes, as specified with -x; otherwise 0
hovering_-frequency	Hovering frequency in seconds; as set with -pp
ignore_hovering	True if -p was specified
verbose_upload	True if -vu was specified
verbose_upload_-data	Reserved; use 0
verbose_-download	Reserved; use 0
verbose_-download_data	Reserved; use 0
autoclose	TRUE if -k was specified
ping	TRUE if -pi was specified
_unused	Reserved; use 0
encryption_key	Database key, as specified with -ek

Member	Description
upload_defs	Linked list of publications to be uploaded together—see a_syncpub
log_file_name	output log file name specified with -o or -ot
user_name	MobiLink user name, specifed with -u
verbose_-minimum	TRUE if -v was specified
verbose_hook	TRUE if -vs was specified
verbose_row_data	TRUE if -vr was specified
verbose_row_cnts	TRUE if -vn was specified
verbose_option_-info	TRUE if -vo was specified
strictly_ignore_-trigger_ops	Reserved; use 0
_unused2	Reserved; use 0
est_upld_row_cnt	Estimated number of rows to upload, specified with -urc
status_rtn	Reserved; use NULL
warningrtn	Function to display warning messages
ce_reproc_argv	Reserved, use NULL
upload_only	True if -uo was specified
download_only	TRUE if -ds was specified
allow_schema_-change	TRUE if -sc was specified
dnld_gen_num	TRUE if -bg was specified
_unused3	Reserved; use 0
apply_dnld_file	File specified with -ba; otherwise NULL
create_dnld_file	File specified with -bc; otherwise NULL
sync_params	User authentication parameters – specified with -ap
dnld_file_extra	String specified with -be
com_server	Reserved, use NULL

Member	Description
trans_upload	TRUE if -tu was specified
continue_- download	TRUE if -dc is specified
dnld_read_size	Value specified by -drs switch
dnld_fail_len	Reserved, use 0
upld_fail_len	Reserved, use 0

Some members correspond to features accessible from the *dbmlsync* command-line utility. Unused members should be assigned the value 0, FALSE, or NULL, depending on datatype.

See the *dbtools.h* header file for additional comments.

☞ For more information, see "MobiLink synchronization client" [*MobiLink Clients,* page 96].

See also

a_syncpub structure

Function
Holds information needed for the *dbmlsync* utility.

Syntax
```
typedef struct a_syncpub {
    struct a_syncpub * next;
    char * pub_name;
    char * ext_opt;
    a_bit_field alloced_by_dbsync: 1;
} a_syncpub;
```

Members

Member	Description
a_syncpub	pointer to the next node in the list, NULL for the last node
pub_name	publication name(s) specified for this −n option. This is the exact string following -n on the command line.
ext_opt	extended options specified using the −eu option
alloced_by_dbsync	Reserved; use FALSE

See also   "DBTools Interface for dbmlsync" [*MobiLink Clients,* page 267]

a_sysinfo structure

Function   Holds information needed for *dbinfo* and *dbunload* utilities using the DBTools library.

```
typedef struct a_sysinfo {
a_bit_field valid_data : 1;
a_bit_field blank_padding : 1;
a_bit_field case_sensitivity : 1;
a_bit_field encryption : 1;
a_bit_field  pwd_case_sensitivity: 1;
char default_collation[11];
unsigned short page_size;
 } a_sysinfo;
```

Members

Member	Description
valid_date	Bit-field indicating whether the following values are set
blank_padding	1 if blank padding is used in this database, 0 otherwise
case_sensitivity	1 if the database is case-sensitive, 0 otherwise
encryption	1 if the database is encrypted, 0 otherwise
pwd_case_- sensitivity	
default_- collation	The collation sequence for the database
page_size	The page size for the database

See also   "a_db_info structure" on page 291

a_table_info structure

Function   Holds information about a table needed as part of the a_db_info structure.

Syntax   typedef struct a_table_info {
                          struct a_table_info * next;
                          a_sql_uint32 table_id;
                          a_sql_uint32 table_pages;
                          a_sql_uint32 index_pages;
                          a_sql_uint32 table_used;
                          a_sql_uint32 index_used;
                          char * table_name;
                          a_sql_uint32 table_used_pct;
                          a_sql_uint32 index_used_pct;
                          } a_table_info;

Members

Member	Description
next	Next table in the list
table_id	ID number for this table
table_pages	Number of table pages
index_pages	Number of index pages
table_used	Number of bytes used in table pages
index_used	Number of bytes used in index pages
table_name	Name of the table
table_used_pct	Table space utilization as a percentage
index_used_pct	Index space utilization as a percentage

See also   "a_db_info structure" on page 291

a_translate_log structure

Function   Holds information needed for transaction log translation using the DBTools
                          library.

Syntax   typedef struct a_translate_log {
                          unsigned short   version;
                          const char *   logname;
                          const char *   sqlname;
                          p_name   userlist;
                          MSG_CALLBACK confirmrtn;
                          MSG_CALLBACK errorrtn;
                          MSG_CALLBACK msgrtn;
                          char   userlisttype;

```
                a_bit_field     remove_rollback : 1;
                a_bit_field     ansi_sql   : 1;
                a_bit_field     since_checkpoint: 1;
                a_bit_field     omit_comments  : 1;
                a_bit_field     replace      : 1;
                a_bit_field     debug      : 1;
                a_bit_field     include_trigger_trans : 1;
                a_bit_field     comment_trigger_trans : 1;
                a_sql_uint32    since_time;
                const char *    connectparms;
                MSG_CALLBACK logrtn;
                const char *   reserved_1;
                const char *   reserved_2;
                a_sql_uint32      debug_dump_size;

                a_bit_field     debug_sql_remote      : 1;
                a_bit_field     debug_dump_hex        : 1;
                a_bit_field     debug_dump_char       : 1;
                a_bit_field     debug_page_offsets    : 1;
                a_bit_field     reserved_3          : 1;
                a_bit_field     use_hex_offsets       : 1;
                a_bit_field     use_relative_offsets   : 1;
                a_bit_field     include_audit        : 1;
                a_bit_field     chronological_order   : 1;
                a_bit_field     force_recovery        : 1;
                a_bit_field     include_subsets       : 1;
                a_bit_field     force_chaining        : 1;
                a_sql_uint32     recovery_ops;
                a_sql_uint32     recovery_bytes;

                const char *   include_source_sets;
                const char *   include_destination_sets;
                const char *   include_scan_range;
                const char *   repserver_users;
                const char *   include_tables;
                const char *   include_publications;
                const char *   queueparms;
                a_bit_field      generate_reciprocals  :1;
                a_bit_field      match_mode          :1;
                const char *   match_pos;
                MSG_CALLBACK   statusrtn;
                const char *   encryption_key;
                a_bit_field      show_undo           :1;
                a_bit_field     quiet :1;
                const char *   logs_dir;
                } a_translate_log;
```

Members

Member	Description
version	DBTools version number
logname	The name of the transaction log. If set to NULL, there is no log.
sqlname	
userlist	
confirmrtn	Callback routine for confirming an action
errorrtn	Callback routine for handling an error message
msgrtn	Callback routine for handling an information message
userlisttype	
remove_rollback	
ansi_sql	
since_checkpoint	
omit_comments	
replace	Operate without confirming actions
debug	Reserved
include_trigger_trans	
comment_trigger_trans	
since_time	

Member	Description

connectparms	Parameters needed to connect to the database. They take the form of connection strings, such as the following:
	
	```"UID=DBA;PWD=SQL;DBF=c:\asa\ asademo.db"```
	
	The database server would be started by the connection string START parameter. The following is an example.
	
	```"START=d:\asa90\win32\dbeng9.exe"```
	
	A full example connection string including the START parameter:
	
	```"userid=dba;password=sql;dbf=d:\ asa90\asademo.db;start=d:\asa90\ win32\dbeng9.exe"```
	
	For the full range of connection string options, see "Connection parameters" [*ASA Database Administration Guide,* page 176]

logrtn	Function to write messages only to the log file
reserved_1	Reserved, use NULL
reserved_2	Reserved, use NULL
debug_dump_size	Reserved, use zero
debug_sql_remote	Reserved, use false
debug_dump_hex	Reserved, use false
debug_dump_char	Reserved, use false
debug_page_offsets	Reserved, use false
reserved_3	Reserved, use false
use_hex_offsets	Reserved, use false
use_relative_offsets	Reserved, use false
include_audit	Reserved, use false
chronological_order	Reserved, use false

Member	Description
force_recovery	Reserved, use false
include_subsets	Reserved, use false
force_chaining	Reserved, use false
recovery_ops	Rreserved, use zero
recovery_bytes	Reserved, use zero
include_source_sets	Reserved, use NULL
include_destination_sets	Reserved, use NULL
include_scan_range	Reserved, use NULL
repserver_users	Reserved, use NULL
include_tables	Reserved, use NULL
include_publications	Reserved, use NULL
queueparms	Reserved for sstran only
generate_reciprocals	Reserved, use false
match_mode	Reserved, use false
match_pos	Reserved, use NULL
statusrtn	Callback routine for handling a status message
encryption_key	-ek sets string
show_undo	reserved, use false
quiet	Operate without printing messages (1), or print messages (0)
logs_dir	

The members correspond to features accessible from the *dbtran* command-line utility.

See the *dbtools.h* header file for additional comments.

See also

For more information on callback functions, see "Using callback functions" on page 263.

a_truncate_log structure

Function   Holds information needed for transaction log truncation using the DBTools library.

Syntax

```
typedef struct a_truncate_log {
unsigned short version;
const char * connectparms;
const char * startline;
MSG_CALLBACK errorrtn;
MSG_CALLBACK msgrtn;
a_bit_field quiet : 1;
a_bit_field  server_backup : 1;
char truncate_interrupted;
} a_truncate_log;
```

Members

Member	Description
version	DBTools version number.
connectparms	Parameters needed to connect to the database. They take the form of connection strings, such as the following:
	`"UID=DBA;PWD=SQL;DBF=c:\asa\` `    asademo.db"`
	The database server would be started by the connection string START parameter. The following is an example.
	`"START=d:\asa90\win32\dbeng9.exe"`
	A full example connection string including the START parameter:

	`"userid=dba;password=sql;dbf=d:\`
	`    asa90\asademo.db;start=d:\asa90\`
	`    win32\dbeng9.exe"`

	For the full range of connection string options, see "Connection parameters" [*ASA Database Administration Guide,* page 176]

Member	Description
startline	Obsolete: The startline field is no longer used. The database server would be started by the connection string START parameter. Refer to connectparams for details.
errorrtn	Callback routine for handling an error message
msgrtn	Callback routine for handling an information message
quiet	Operate without printing messages (1), or print messages (0)
server_backup	
truncate_-interrupted	Indicates that the operation was interrupted

See also

For more information on callback functions, see .

an_unload_db structure

Function   Holds information needed to unload a database using the DBTools library or extract a remote database for SQL Remote. Those fields used by the *dbxtract* SQL Remote extraction utility are indicated.

Syntax
```
typedef struct an_unload_db {
unsigned short version;
const char * connectparms;
const char * startline;
const char * temp_dir;
const char * reload_filename;
MSG_CALLBACK errorrtn;
MSG_CALLBACK msgrtn;
MSG_CALLBACK statusrtn;
MSG_CALLBACK confirmrtn;
char unload_type;
char verbose;
```

```
                      a_bit_field unordered : 1;
                      a_bit_field no_confirm : 1;
                      a_bit_field use_internal_unload : 1;
                      a_bit_field dbo_avail : 1;
                      a_bit_field extract : 1;
                      a_bit_field table_list_provided : 1;
                      a_bit_field exclude_tables : 1;
                      a_bit_field more_flag_bits_present : 1;
                      a_sysinfo sysinfo;
                      const char * remote_dir;
                      const char * dbo_username;
                      const char * subscriber_username;
                      const char * publisher_address_type;
                      const char * publisher_address;
                      unsigned short isolation_level;

                      a_bit_field start_subscriptions : 1;
                      a_bit_field exclude_foreign_keys : 1;
                      a_bit_field exclude_procedures : 1;
                      a_bit_field exclude_triggers : 1;
                      a_bit_field exclude_views : 1;
                      a_bit_field isolation_set : 1;
                      a_bit_field include_where_subscribe : 1;
                      a_bit_field debug : 1;
                      p_name table_list;
                      a_bit_short escape_char_present : 1;
                      a_bit_short view_iterations_present : 1;
                      a_bit_short use_internal_reload : 1;

                      unsigned short view_iterations;
                      char escape_char;
                      char * reload_connectparms;
                      char * reload_db_filename;
                      a_bit_field output_connections:1;
                      char unload_interrupted;
                      a_bit_field replace_db:1;
                      const char * locale;
                      const char * site_name;
                      const char * template_name;
                      a_bit_field preserve_ids:1;
                      a_bit_field exclude_hooks:1;
                      char * reload_db_logname;
                      const char * encryption_key;
                      const char * encryption_algorithm;
                      a_bit_field syntax_version_7:1;
                      a_bit_field remove_java:1;
                      unsigned short   reload_page_size;
                      } an_unload_db;
```

Members

Members	Description
version	DBTools version number.

connectparms	Parameters needed to connect to the database. They take the form of connection strings, such as the following:
	
	`"UID=DBA;PWD=SQL;DBF=c:\asa\`
	`   asademo.db"`
	
	The database server would be started by the connection string START parameter. The following is an example.
	
	`"START=d:\asa90\win32\dbeng9.exe"`
	
	A full example connection string including the START parameter:
	
	`"userid=dba;password=sql;dbf=d:\`
	`   asa90\asademo.db;start=d:\asa90\`
	`   win32\dbeng9.exe"`
	
	For the full range of connection string options, see "Connection parameters" [*ASA Database Administration Guide,* page 176]

startline	Obsolete: The startline field is no longer used. The database server would be started by the connection string START parameter. Refer to connectparams for details.
temp_dir	directory for unloading data files
reload_filename	dbunload -r option, something like "reload.sql"
errorrtn	optional user callback
msgrtn	optional user callback
statusrtn	optional user callback
confirmrtn	optional user callback
unload_type	UNLOAD_xxx (shown above)

Members	Description
verbose	VB_xxx (shown above)
unordered	dbunload -u sets TRUE
no_confirm	dbunload -y sets TRUE
use_internal_unload	dbunload -i? sets TRUE
dbo_avail	Obsolete
extract	TRUE if dbxtract, otherwise FALSE
table_list_provided	dbunload -e <list> or -i sets TRUE
exclude_tables	dbunload -e sets TRUE dbunload -i (undocumented) sets FALSE
more_flag_bits_present	usually set TRUE
sysinfo	(internal use)
remote_dir	(like temp_dir) but for internal un-loads // on server side
dbo_username	The dbo_username functionality has been dropped as of 6.0 // you can leave it set to NULL
subscriber_username	argument to dbxtract
publisher_address_type	(unused)
publisher_address	(unused)
isolation_level	dbxtract -l sets value
start_subscriptions	dbxtract TRUE by default, -b sets FALSE
exclude_foreign_keys	dbxtract -xf sets TRUE
exclude_procedures	dbxtract -xp sets TRUE
exclude_triggers	dbxtract -xt sets TRUE
exclude_views	dbxtract -xv sets TRUE
isolation_set	dbxtract -l sets TRUE
include_where_subscribe	dbxtract -f sets TRUE/mlxtract sets TRUE
debug	(internal use)

Members	Description
table_list	selective table list
escape_char_present	-p sets TRUE and // escape_char must be set
view_iterations_present	(unused)
use_internal_reload	usually set TRUE // -ix/-xx sets FALSE // -ii/-xi sets TRUE
view_iterations	(unused)
escape_char	used when escape_char_present is TRUE
reload_connectparms	userid,password,database for reload database
reload_db_filename	filename of reload database to create
output_connections	(internal use)
unload_interrupted	(internal use)
replace_db	dbunload -ar sets TRUE
locale	(internal use) locale (language and charset)
site_name	for dbxtract: specify a site name
template_name	for dbxtract: specify a template name
preserve_ids	dbunload sets TRUE/-m sets FALSE
exclude_hooks	dbxtract -hx sets TRUE
reload_db_logname	log filename for the reload database
encryption_key	-ek sets string
encryption_algorithm	-ea sets, "aes" or "aes_fips"
syntax_version_7	mlxtract -s7 sets TRUE
remove_java	-jr sets TRUE
reload_page_size	dbunload -ap sets value

The members correspond to features accessible from the *dbunload* and *dbxtract*, and *mlxtract* command-line utilities.

See the *dbtools.h* header file for additional comments.

See also   "DBUnload function" on page 277

an_upgrade_db structure

Function   Holds information needed to upgrade a database using the DBTools library.

Syntax

```
typedef struct an_upgrade_db {
unsigned short version;
const char * connectparms;
const char * startline;
MSG_CALLBACK errorrtn;
MSG_CALLBACK msgrtn;
MSG_CALLBACK statusrtn;
a_bit_field quiet : 1;
a_bit_field dbo_avail : 1;
const char *   dbo_username;
a_bit_field java_classes  : 1;
a_bit_field jconnect : 1;
a_bit_field remove_java : 1;
a_bit_field java_switch_specified : 1;
const char * jdk_version;
} an_upgrade_db;
```

Members

Member	Description
version	DBTools version number.

Member	Description

connectparms	Parameters needed to connect to the database. They take the form of connection strings, such as the following:
	
	`"UID=DBA;PWD=SQL;DBF=c:\asa\`
	`   asademo.db"`
	
	The database server would be started by the connection string START parameter. The following is an example.
	
	`"START=d:\asa90\win32\dbeng9.exe"`
	
	A full example connection string including the START parameter:
	
	`"userid=dba;password=sql;dbf=d:\`
	`   asa90\asademo.db;start=d:\asa90\`
	`   win32\dbeng9.exe"`
	
	For the full range of connection string options, see "Connection parameters" [*ASA Database Administration Guide, page 176*]

startline	Obsolete: The startline field is no longer used. The database server would be started by the connection string START parameter. Refer to connectparams for details.
errorrtn	Callback routine for handling an error message
msgrtn	Callback routine for handling an information message
statusrtn	Callback routine for handling a status message
quiet	Operate without printing messages (1), or print messages (0)
dbo_avail	Obsolete
dbo_username	The dbo_username functionality has been dropped as of 6.0 – leave it set to NULL
java_classes	Upgrade the database to be Java-enabled
jconnect	Upgrade the database to include jConnect procedures
remove_java	Upgrade the database, removing the Java features
java_switch_-specified	
jdk_version	One of the values for the *dbinit* – jdk option.

See also   "DBUpgrade function" on page 278

For more information on callback functions, see .

a_validate_db structure

Function   Holds information needed for database validation using the DBTools library.

Syntax

```
typedef struct a_validate_db {
unsigned short version;
const char * connectparms;
const char * startline;
p_name tables;
MSG_CALLBACK errorrtn;
MSG_CALLBACK msgrtn;
MSG_CALLBACK statusrtn;
a_bit_field quiet : 1;
a_bit_field index : 1;
a_validate_type type;
} a_validate_db;
```

Members

Member	Description
version	DBTools version number.
connectparms	Parameters needed to connect to the database. They take the form of connection strings, such as the following: `"UID=DBA;PWD=SQL;DBF=c:\asa\` `asademo.db"` The database server would be started by the connection string START parameter. The following is an example. `"START=d:\asa90\win32\dbeng9.exe"` A full example connection string including the START parameter: `"userid=dba;password=sql;dbf=d:\` `asa90\asademo.db;start=d:\asa90\` `win32\dbeng9.exe"` For the full range of connection string options, see "Connection parameters" [*ASA Database Administration Guide, page 176*]
startline	Obsolete: The startline field is no longer used. The database server would be started by the connection string START parameter. Refer to connectparams for details.
tables	Pointer to a linked list of table names

Member	Description
errorrtn	Callback routine for handling an error message
msgrtn	Callback routine for handling an information message
statusrtn	Callback routine for handling a status message
quiet	Operate without printing messages (1), or print messages (0)
index	Validate indexes
type	See "a_validate_type enumeration" on page 325

See also   "DBValidate function" on page 278

"a_name structure" on page 297

☞ For more information on callback functions, see "Using callback functions" on page 263.

a_writefile structure

Function   Holds information needed for database write file management using the DBTools library.

Syntax
```
typedef struct a_writefile {
unsigned short version;
const char * writename;
const char * wlogname;
const char * dbname;
const char * forcename;
MSG_CALLBACK confirmrtn;
MSG_CALLBACK errorrtn;
MSG_CALLBACK msgrtn;
char action;
a_bit_field quiet   : 1;
a_bit_field erase   : 1;
a_bit_field force   : 1;
a_bit_field mirrorname_present : 1;
const char *  wlogmirrorname;
a_bit_field make_log_and_mirror_names: 1;
const char * encryption_key;
} a_writefile;
```

Members

Member	Description
version	DBTools version number

Member	Description
writename	Write file name
wlogname	Used only when creating write files
dbname	Used when changing and creating write files
forcename	Forced file name reference
confirmrtn	Callback routine for confirming an action. Only used when creating a write file
errorrtn	Callback routine for handling an error message
msgrtn	Callback routine for handling an information message
action	Reserved for use by Sybase
quiet	Operate without printing messages (1), or print messages (0)
erase	Used for creating write files only. Erase without confirmation (1) or with confirmation (0)
force	If 1, force the write file to point to a named file
mirrorname_present	Used when creating only. Set to 1. Indicates that the version of DBTools is recent enough to support the mirrorname field
wlogmirrorname	Name of the transaction log mirror
make_log_and_-mirror_names	If TRUE, use the values in wlogname and wlogmirrorname to determine filenames.
encryption_key	The encryption key for the database file.

See also   "DBChangeWriteFile function" on page 270

"DBCreateWriteFile function" on page 271

"DBStatusWriteFile function" on page 274

For more information on callback functions, see "Using callback functions" on page 263.

DBTools enumeration types

This section lists the enumeration types that are used by the DBTools library. The enumerations are listed alphabetically.

Verbosity enumeration

Function   Specifies the volume of output.

Syntax
```
enum {
VB_QUIET,
VB_NORMAL,
VB_VERBOSE
};
```

Parameters

Value	Description
VB_QUIET	No output
VB_NORMAL	Normal amount of output
VB_VERBOSE	Verbose output, useful for debugging

See also   "a_create_db structure" on page 286

"an_unload_db structure" on page 314

Blank padding enumeration

Function   Used in the "a_create_db structure" on page 286, to specify the value of blank_pad.

Syntax
```
enum {
NO_BLANK_PADDING,
BLANK_PADDING
};
```

Parameters

Value	Description
NO_BLANK_PADDING	Does not use blank padding
BLANK_PADDING	Uses blank padding

See also   "a_create_db structure" on page 286

dbtran_userlist_type enumeration

Function | The type of a user list, as used by an "a_translate_log structure" on page 308.

Syntax | typedef enum dbtran_userlist_type {
DBTRAN_INCLUDE_ALL,
DBTRAN_INCLUDE_SOME,
DBTRAN_EXCLUDE_SOME
} dbtran_userlist_type;

Parameters

Value	Description
DBTRAN_INCLUDE_ALL	Include operations from all users
DBTRAN_INCLUDE_SOME	Include operations only from the users listed in the supplied user list
DBTRAN_EXCLUDE_-SOME	Exclude operations from the users listed in the supplied user list

See also | "a_translate_log structure" on page 308

dbunload type enumeration

Function | The type of unload being performed, as used by the "an_unload_db structure" on page 314.

Syntax | enum {
UNLOAD_ALL,
UNLOAD_DATA_ONLY,
UNLOAD_NO_DATA
};

Parameters

Value	Description
UNLOAD_ALL	Unload both data and schema
UNLOAD_DATA_ONLY	Unload data. Do not unload schema
UNLOAD_NO_DATA	Unload schema only

See also | "an_unload_db structure" on page 314

a_validate_type enumeration

Function | The type of validation being performed, as used by the "a_validate_db

Syntax

```
typedef enum {
VALIDATE_NORMAL = 0,
VALIDATE_DATA,
VALIDATE_INDEX,
VALIDATE_EXPRESS,
VALIDATE_FULL
} a_validate_type;
```

Parameters

Value	Description
VALIDATE_- NORMAL	Validate with the default check only.
VALIDATE_DATA	Validate with data check in addition to the default check.
VALIDATE_INDEX	Validate with index check in addition to the default check.
VALIDATE_EXPRESS	Validate with express check in addition to the default and data checks.
VALIDATE_FULL	Validate with both data and index check in addition toe the default check.

See also

"Validating a database using the dbvalid command-line utility" [*ASA Database Administration Guide,* page 605]

"VALIDATE TABLE statement" [*ASA SQL Reference,* page 662]

CHAPTER 9

The OLE DB and ADO Programming Interfaces

About this chapter
This chapter describes how to use the OLE DB interface to Adaptive Server Anywhere.

Many applications that use the OLE DB interface do so through the Microsoft ActiveX Data Objects (ADO) programming model, rather than directly. This chapter also describes ADO programming with Adaptive Server Anywhere.

Contents

Introduction to OLE DB

OLE DB is a data access model from Microsoft. It uses the Component Object Model (COM) interfaces and, unlike ODBC, OLE DB does not assume that the data source uses a SQL query processor.

Adaptive Server Anywhere includes an **OLE DB provider** named **ASAProv**. This provider is available for current Windows and Windows CE platforms.

You can also access Adaptive Server Anywhere using the Microsoft OLE DB Provider for ODBC (MSDASQL), together with the Adaptive Server Anywhere ODBC driver.

Using the Adaptive Server Anywhere OLE DB provider brings several benefits:

♦ Some features, such as updating through a cursor, are not available using the OLE DB/ODBC bridge.

♦ If you use the Adaptive Server Anywhere OLE DB provider, ODBC is not required in your deployment.

♦ MSDASQL allows OLE DB clients to work with any ODBC driver but does not guarantee that you can use the full range of functionality of each ODBC driver. Using the Adaptive Server Anywhere provider, you can get full access to Adaptive Server Anywhere features from OLE DB programming environments.

Supported platforms

The Adaptive Server Anywhere OLE DB provider is designed to work with OLE DB 2.5 and later. For Windows CE and its successors, the OLE DB provider is designed for ADOCE 3.0 and later.

ADOCE is the Microsoft ADO for Windows CE SDK and provides database functionality for applications developed with the Windows CE Toolkits for Visual Basic 5.0 and Visual Basic 6.0.

☞ For a list of supported platforms, see "Operating system versions" [*Introducing SQL Anywhere Studio,* page 111].

Distributed transactions

The OLE DB driver can be used as a resource manager in a distributed transaction environment.

☞ For more information, see "Three-Tier Computing and Distributed Transactions" on page 507.

ADO programming with Adaptive Server Anywhere

ADO (ActiveX Data Objects) is a data access object model exposed through an Automation interface, which allows client applications to discover the methods and properties of objects at runtime without any prior knowledge of the object. Automation allows scripting languages like Visual Basic to use a standard data access object model. ADO uses OLE DB to provide data access.

Using the Adaptive Server Anywhere OLE DB provider, you get full access to Adaptive Server Anywhere features from an ADO programming environment.

This section describes how to carry out basic tasks while using ADO from Visual Basic. It is not a complete guide to programming using ADO.

Code samples from this section can be found in the following files:

Development tool	Sample
Microsoft Visual Basic 6.0	*Samples\ASA\VBSampler\vbsampler.vbp*
Microsoft eMbedded Visual Basic 3.0	*Samples\ASA\ADOCE\OLEDB_PocketPC.ebp*

☞ For information on programming in ADO, see your development tool documentation.

Connecting to a database with the Connection object

This section describes a simple Visual Basic routine that connects to a database.

Sample code   You can try this routine by placing a command button named **Command1** on a form, and pasting the routine into its **Click** event. Run the program and click the button to connect and then disconnect.

```
Private Sub cmdTestConnection_Click()
    ' Declare variables
    Dim myConn As New ADODB.Connection
    Dim myCommand As New ADODB.Command
    Dim cAffected As Long

    On Error GoTo HandleError

    ' Establish the connection
    myConn.Provider = "ASAProv"
    myConn.ConnectionString = _
        "Data Source=ASA 9.0 Sample"
    myConn.Open
    MsgBox "Connection succeeded"
    myConn.Close
    Exit Sub

HandleError:
    MsgBox "Connection failed"
    Exit Sub
End Sub
```

Notes

The sample carries out the following tasks:

♦ It declares the variables used in the routine.

♦ It establishes a connection, using the Adaptive Server Anywhere OLE DB provider, to the sample database.

♦ It uses a Command object to execute a simple statement, which displays a message on the database server window.

♦ It closes the connection.

When the **ASAProv** provider is installed, it registers itself. This registration process includes making registry entries in the COM section of the registry, so that ADO can locate the DLL when the **ASAProv** provider is called. If you change the location of your DLL, you must reregister it.

❖ **To register the OLE DB provider**

1. Open a command prompt.

2. Change to the directory where the OLE DB provider is installed.

3. Enter the following command to register the provider:

```
regsvr32 dboledb9.dll
```

☞ For more information about connecting to a database using OLE DB, see "Connecting to a database using OLE DB" [*ASA Database Administration Guide,* page 62].

330

Executing statements with the Command object

This section describes a simple routine that sends a simple SQL statement to the database.

Sample code

You can try this routine by placing a command button named **Command2** on a form, and pasting the routine into its **Click** event. Run the program and click the button to connect, display a message on the database server window, and then disconnect.

```
Private Sub cmdUpdate_Click()
    ' Declare variables
    Dim myConn As New ADODB.Connection
    Dim myCommand As New ADODB.Command
    Dim cAffected As Long
    ' Establish the connection
    myConn.Provider = "ASAProv"
    myConn.ConnectionString = _
       "Data Source=ASA 9.0 Sample"
    myConn.Open

    'Execute a command
    myCommand.CommandText = _
    "update customer set fname='Liz' where id=102"
    Set myCommand.ActiveConnection = myConn
    myCommand.Execute cAffected
    MsgBox CStr(cAffected) +
    " rows affected.", vbInformation

    myConn.Close
End Sub
```

Notes

After establishing a connection, the example code creates a Command object, sets its **CommandText** property to an update statement, and sets its **ActiveConnection** property to the current connection. It then executes the update statement and displays the number of rows affected by the update in a message box.

In this example, the update is sent to the database and committed as soon as it is executed.

☞ For information on using transactions within ADO, see "Using transactions" on page 334.

You can also carry out updates through a cursor.

☞ For more information, see "Updating data through a cursor" on page 333.

Querying the database with the Recordset object

The ADO **Recordset** object represents the result set of a query. You can use it to view data from a database.

Sample code
You can try this routine by placing a command button named **cmdQuery** on a form and pasting the routine into its **Click** event. Run the program and click the button to connect, display a message on the database server window, execute a query and display the first few rows in message boxes, and then disconnect.

```
Private Sub cmdQuery_Click()
' Declare variables
    Dim myConn As New ADODB.Connection
    Dim myCommand As New ADODB.Command
    Dim myRS As New ADODB.Recordset

    On Error GoTo ErrorHandler:

    ' Establish the connection
    myConn.Provider = "ASAProv"
    myConn.ConnectionString = _
       "Data Source=ASA 9.0 Sample"
    myConn.CursorLocation = adUseServer
    myConn.Mode = adModeReadWrite
    myConn.IsolationLevel = adXactCursorStability
    myConn.Open

    'Execute a query
    Set myRS = New Recordset
    myRS.CacheSize = 50
    myRS.Source = "Select * from customer"
    myRS.ActiveConnection = myConn
    myRS.CursorType = adOpenKeyset
    myRS.LockType = adLockOptimistic
    myRS.Open

    'Scroll through the first few results
    myRS.MoveFirst
    For i = 1 To 5
      MsgBox myRS.Fields("company_name"), vbInformation
      myRS.MoveNext
    Next

    myRS.Close
    myConn.Close
    Exit Sub
ErrorHandler:
    MsgBox Error(Err)
    Exit Sub
End Sub
```

Notes The **Recordset** object in this example holds the results from a query on the Customer table. The **For** loop scrolls through the first several rows and displays the company_name value for each row.

This is a simple example of using a cursor from ADO.

☞ For more advanced examples of using a cursor from ADO, see "Working with Recordset object" on page 333.

Working with Recordset object

When working with Adaptive Server Anywhere, the ADO **Recordset** represents a cursor. You can choose the type of cursor by declaring a **CursorType** property of the **Recordset** object before you open the **Recordset**. The choice of cursor type controls the actions you can take on the **Recordset** and has performance implications.

Cursor types The set of cursor types supported by Adaptive Server Anywhere is described in "Cursor properties" on page 26. ADO has its own naming convention for cursor types.

The available cursor types, the corresponding cursor type constants, and the Adaptive Server Anywhere types they are equivalent to, are as follows:

ADO cursor type	ADO constant	Adaptive Server Anywhere type
Dynamic cursor	adOpenDynamic	Dynamic scroll cursor
Keyset cursor	adOpenKeyset	Scroll cursor
Static cursor	adOpenStatic	Insensitive cursor
Forward only	adOpenForwardOnly	No-scroll cursor

☞ For information on choosing a cursor type that is suitable for your application, see "Choosing cursor types" on page 26.

Sample code The following code sets the cursor type for an ADO **Recordset** object:

```
Dim myRS As New ADODB.Recordset
myRS.CursorType = adOpenDynamic
```

Updating data through a cursor

The Adaptive Server Anywhere OLE DB provider lets you update a result set through a cursor. This capability is not available through the MSDASQL provider.

Updating record sets You can update the database through a record set.

```
Private Sub Command6_Click()
    Dim myConn As New ADODB.Connection
    Dim myRS As New ADODB.Recordset
    Dim SQLString As String
    ' Connect
    myConn.Provider = "ASAProv"
    myConn.ConnectionString = _
        "Data Source=ASA 9.0 Sample"
    myConn.Open
    myConn.BeginTrans
    SQLString = "Select * from customer"
    myRS.Open SQLString, _
      myConn, adOpenDynamic, adLockBatchOptimistic

    If myRS.BOF And myRS.EOF Then
        MsgBox "Recordset is empty!", _
        16, "Empty Recordset"
    Else
        MsgBox "Cursor type: " + _
        CStr(myRS.CursorType), vbInformation
        myRS.MoveFirst
        For i = 1 To 3
            MsgBox "Row: " + CStr(myRS.Fields("id")), _
             vbInformation
            If i = 2 Then
                myRS.Update "City", "Toronto"
                myRS.UpdateBatch
            End If
            myRS.MoveNext
        Next i
'        myRS.MovePrevious
        myRS.Close
    End If
    myConn.CommitTrans
    myConn.Close
End Sub
```

Notes   If you use the adLockBatchOptimistic setting on the recordset, the
myRS.Update method does not make any changes to the database itself.
Instead, it updates a local copy of the **Recordset**.

The **myRS.UpdateBatch** method makes the update to the database server,
but does not commit it, because it is inside a transaction. If an **UpdateBatch**
method was invoked outside a transaction, the change would be committed.

The **myConn.CommitTrans** method commits the changes. The **Recordset**
object has been closed by this time, so there is no issue of whether the local
copy of the data is changed or not.

Using transactions

By default, any change you make to the database using ADO is committed

as soon as it is executed. This includes explicit updates, as well as the **UpdateBatch** method on a **Recordset**. However, the previous section illustrated that you can use the **BeginTrans** and **RollbackTrans** or **CommitTrans** methods on the **Connection** object to use transactions.

Transaction isolation level is set as a property of the Connection object. The IsolationLevel property can take on one of the following values:

ADO isolation level	Constant	ASA level
Unspecified	adXactUnspecified	Not applicable. Set to 0
Chaos	adXactChaos	Unsupported. Set to 0
Browse	adXactBrowse	0
Read uncommitted	adXactReadUncommit-ted	0
Cursor stability	adXactCursorStability	1
Read committed	adXactReadCommitted	1
Repeatable read	adXactRepeatableRead	2
Isolated	adXactIsolated	3
Serializable	adXactSerializable	3

☞ For more information on isolation levels, see "Isolation levels and consistency" [*ASA SQL User's Guide,* page 106].

Supported OLE DB interfaces

The OLE DB API consists of a set of interfaces. The following table describes the support for each interface in the Adaptive Server Anywhere OLE DB driver.

Interface	Purpose	Limitations
IAccessor	Define bindings between client memory and data store values.	DBACCESSOR_-PASSBYREF not supported. DBACCESSOR_-OPTIMIZED not supported.
IAlterIndex IAlterTable	Alter tables, indexes, and columns.	Not supported.
IChapteredRowset	A chaptered rowset allows rows of a rowset to be accessed in separate chapters.	Not supported. Adaptive Server Anywhere does not support chaptered rowsets.
IColumnsInfo	Get simple information about the columns of a rowset.	Not on CE.
IColumnsRowset	Get information about optional metadata columns in a rowset, and get a rowset of column metadata.	Not on CE.
ICommand	Execute SQL commands.	Does not support calling. IcommandProperties: GetProperties with DBPROPSET_-PROPERTIESINERROR to find properties that could not have been set.

Interface	Purpose	Limitations
ICommandPersist	Persist the state of a command object (but not any active rowsets). These persistent command objects can subsequently be enumerated using the PROCEDURES or VIEWS rowset.	Not on CE.
ICommandPrepare	Prepare commands.	Not on CE.
ICommandProperties	Set Rowset properties for rowsets created by a command. Most commonly used to specify the interfaces the rowset should support.	Supported.
ICommandText	Set the SQL command text for ICommand.	Only the DBGUID_-DEFAULT SQL dialect is supported.
IcommandWithParameters	Set or get parameter information for a command.	No support for parameters stored as vectors of scalar values.
		No support for BLOB parameters.
		Not on CE.
IConvertType		Supported.
		Limited on CE.
IDBAsynchNotify	Asynchronous processing.	Not supported.
IDBAsyncStatus	Notify client of events in the asynchronous processing of data source initialization, populating rowsets, and so on.	
IDBCreateCommand	Create commands from a session.	Supported.
IDBCreateSession	Create a session from a data source object.	Supported.

Interface	Purpose	Limitations
IDBDataSourceAdmin	Create/destroy/modify data source objects, which are COM objects used by clients. This interface is not used to manage data stores (databases).	Not supported.
IDBInfo	Find information about keywords unique to this provider (that is, to find non-standard SQL keywords). Also, find information about literals, special characters used in text matching queries, and other literal information.	Not on CE.
IDBInitialize	Initialize data source objects and enumerators.	Not on CE.
IDBProperties	Manage properties on a data source object or enumerator.	Not on CE.
IDBSchemaRowset	Get information about system tables, in a standard form (a rowset).	Not on CE.
IErrorInfo IErrorLookup IErrorRecords	ActiveX error object support.	Not on CE.
IGetDataSource	Returns an interface pointer to the session's data source object.	Supported.
IIndexDefinition	Create or drop indexes in the data store.	Not supported.
IMultipleResults	Retrieve multiple results (rowsets or row counts) from a command.	Supported.

Interface	Purpose	Limitations
IOpenRowset	Non-SQL way to access a database table by its name.	Supported. Opening a table by its name is supported, not by a GUID.
IParentRowset	Access chaptered/hierarchical rowsets.	Not supported.
IRowset	Access rowsets.	Supported.
IRowsetChange	Allow changes to rowset data, reflected back to the data store. InsertRow/SetData for blobs not yet implemented.	Not on CE.
IRowsetChapter-Member	Access chaptered/hierarchical rowsets.	Not supported.
IRowsetCurrentIndex	Dynamically change the index for a rowset.	Not supported.
IRowsetFind	Find a row within a rowset matching a specified value.	Not supported.
IRowsetIdentity	Compare row handles.	Not supported.
IRowsetIndex	Access database indexes.	Not supported.
IRowsetInfo	Find information about a rowset properties or to find the object that created the rowset.	Not on CE.
IRowsetLocate	Position on rows of a rowset, using bookmarks.	Not on CE.
IRowsetNotify	Provides a COM callback interface for rowset events.	Supported.
IRowsetRefresh	Get the latest value of data that is visible to a transaction.	Not supported.
IRowsetResynch	Old OLEDB 1.x interface, superseded by IRowsetRefresh.	Not supported.

Interface	Purpose	Limitations
IRowsetScroll	Scroll through rowset to fetch row data.	Not supported.
IRowsetUpdate	Delay changes to rowset data until Update is called.	Supported. Not on CE.
IRowsetView	Use views on an existing rowset.	Not supported.
ISequentialStream	Retrieve a blob column.	Supported for reading only. No support for SetData with this interface. Not on CE.
ISessionProperties	Get session property information.	Supported.
ISourcesRowset	Get a rowset of data source objects and enumerators.	Not on CE.
ISQLErrorInfo ISupportErrorInfo	ActiveX error object support.	Optional on CE.
ITableDefinition ITableDefinitionWithConstraints	Create, drop, and alter tables with constraints.	Not on CE.
ITransaction	Commit or abort transactions.	Not all the flags are supported. Not on CE.
ITransactionJoin	Support distributed transactions.	Not all the flags are supported. Not on CE.
ITransactionLocal	Handle transactions on a session. Not all the flags are supported.	Not on CE.
ITransactionOptions	Get or set options on a transaction.	Not on CE.

Interface	Purpose	Limitations
IViewChapter	Work with views on an existing rowset, specifically to apply post-processing filters/sorting on rows.	Not supported.
IViewFilter	Restrict contents of a rowset to rows matching a set of conditions.	Not supported.
IViewRowset	Restrict contents of a rowset to rows matching a set of conditions, when opening a rowset.	Not supported.
IViewSort	Apply sort order to a view.	Not supported.

CHAPTER 10

Introduction to the Adaptive Server Anywhere .NET Data Provider

About this chapter
This chapter introduces you to the Adaptive Server Anywhere .NET data provider.

Contents

Adaptive Server Anywhere .NET data provider features

If you are using Visual Studio .NET on Windows NT/2000/XP, the following data providers are supported for accessing Adaptive Server Anywhere:

♦ **iAnywhere.Data.AsaClient**   uses the Adaptive Server Anywhere .NET data provider described in this book.

♦ **System.Data.Oledb**   is a general-purpose data provider for OLE DB data sources. It is part of the Microsoft .NET Framework. You can use System.Data.Oledb together with the Adaptive Server Anywhere OLE DB driver to access Adaptive Server Anywhere databases.

♦ **System.Data.Odbc**   is a general-purpose data provider for ODBC data sources. It is part of the Microsoft .NET Framework. You can use System.Data.Odbc together with the Adaptive Server Anywhere ODBC driver to access Adaptive Server Anywhere databases.

On Windows CE, only the Adaptive Server Anywhere .NET data provider is supported.

There are some key benefits to using the Adaptive Server Anywhere .NET data provider:

♦ The Adaptive Server Anywhere .NET data provider is faster than the OLE DB provider.

♦ In the .NET environment, the Adaptive Server Anywhere .NET data provider provides native access to Adaptive Server Anywhere. Unlike the other supported providers, it communicates directly with Adaptive Server Anywhere and does not require bridge technology.

Running the sample projects

There are three sample projects included with the Adaptive Server Anywhere .NET data provider. They are:

♦ **SimpleCE**   A Compact Framework Windows CE sample that demonstrates a simple list box that is filled with the names from the employee table when you click the Connect button.

♦ **SimpleWin32**   A Windows sample that demonstrates a simple list box that is filled with the names from the employee table when you click the Connect button.

♦ **TableViewer**   A Windows program that allows you to enter and execute SQL statements.

☞ For tutorials explaining the win32 and Table Viewer samples, see "Using the Adaptive Server Anywhere .NET Data Provider Sample Applications" on page 347.

Note   If your SQL Anywhere installation directory is not the default (*C:\Program Files\Sybase\SQL Anywhere 9*), you may receive an error referencing the data provider DLL when you load the sample projects. If this happens, add a new reference to *iAnywhere.Data.AsaClient.dll.*

☞ For instructions on adding a reference to the DLL, see "Adding a reference to the data provider DLL in your project" on page 358.

Using the Adaptive Server Anywhere .NET Data Provider Sample Applications

About this chapter

This chapter explains how to use the Simple and Table Viewer sample projects included with the Adaptive Server Anywhere .NET data provider.

If your SQL Anywhere installation directory is not the default (*C:\Program Files\Sybase\SQL Anywhere 9*), you may receive an error referencing the data provider DLL when you load the sample projects. If this happens, add a new reference to *iAnywhere.Data.AsaClient.dll*.

☞ For instructions on adding a reference to the DLL, see "Adding a reference to the data provider DLL in your project" on page 358.

Contents

Tutorial: Using the Simple code sample

This tutorial is based on the Simple project that is included with the .NET data provider.

The complete application can be found in your SQL Anywhere installation directory at *Samples\ASA\ADO.NET\SimpleWin32\Simple.csproj*.

The Simple project illustrates the following features:

♦ connecting to a database

♦ executing a query using the AsaCommand object

♦ using the AsaDataReader object

♦ basic error handling

☞ For more information about how the sample works, see "Understanding the Simple sample project" on page 349.

❖ **To run the Simple code sample in Visual Studio .NET**

1. Start Visual Studio .NET.

2. Choose File ➤ Open ➤ Project.

3. Browse to *Samples\ASA\ADO.NET\SimpleWin32* in your SQL Anywhere installation directory and open the *Simple.csproj* project.

4. When you use the Adaptive Server Anywhere .NET data provider in a project, you must add a reference to the data provider DLL. This has already been done in the Simple code sample. You can view the reference to the data provider DLL in the following location:

   ♦ In the Solution Explorer window, open the References folder.

   ♦ You should see iAnywhere.Data.AsaClient in the list.

   ☞ For instructions about adding a reference to the data provider DLL, see "Adding a reference to the data provider DLL in your project" on page 358.

5. You must also add a using directive to your source code to reference the data provider classes. This has already been done in the Simple code sample. To view the using directive:

   ♦ Open the source code for the project.

      • In the Solution Explorer window, select *Form1.cs*.

      • Choose View ➤ Code.

♦ In the `using` directives in the top section, you should see the following line:

```
using iAnywhere.Data.AsaClient;
```

This line is required for C# projects. If you are using Visual Basic .NET, you need to add a different line to your source code.

☞ For more information, see "Referencing the data provider classes in your source code" on page 359.

6. To run the Simple sample, choose Debug ➤ Start Without Debugging or press Ctrl+F5.

The AsaSample dialog appears.

♦ In the AsaSample dialog, click Connect.

The application connects to the Adaptive Server Anywhere sample database and puts the last name of each employee in the dialog, as follows:

7. Click the X in the upper right corner of the screen to terminate the application and disconnect from the sample database. This also shuts down the database server.

You have now run the application. The next section describes the application code.

Understanding the Simple sample project

This section illustrates some key features of the Adaptive Server Anywhere .NET data provider by walking through some of the code from the Simple code sample. The Simple code sample uses the Adaptive Server Anywhere sample database, *asademo.db*, which is held in your SQL Anywhere installation directory.

☞ For information about the sample database, including the tables in the database and the relationships between them, see "About the sample database" [*Introducing SQL Anywhere Studio,* page 198].

In this section, the code is described a few lines at a time. Not all code from the sample is included here. To see the whole code, open the sample project at *Samples\ASA\ADO.NET\SimpleWin32\Simple.csproj.*

Declaring controls   The following code declares a button named btnConnect and a ListBox named listEmployees.

```
private System.Windows.Forms.Button btnConnect;
private System.Windows.Forms.ListBox listEmployees;
```

Connecting to the database   The btnConnect_Click method declares and initializes a connection object (new AsaConnection).

```
private void btnConnect_Click(object sender,
   System.EventArgs e)
   AsaConnection conn = new AsaConnection(
    "Data Source=ASA 9.0 Sample;UID=DBA;PWD=SQL" );
```

The AsaConnection object uses the connection string to connect to the sample database.

```
conn.Open();
```

☞ For more information about the AsaConnection object, see "AsaConnection class" on page 403.

Executing a query   The following code uses the Command object (AsaCommand) to define and execute a SQL statement (SELECT emp_lname FROM employee). Then, it returns the DataReader object (AsaDataReader).

```
AsaCommand cmd = new AsaCommand(
     "select emp_lname from employee", conn );
AsaDataReader reader = cmd.ExecuteReader();
```

☞ For more information about the Command object, see "AsaCommand class" on page 393.

Displaying the results   The following code loops through the rows held in the AsaDataReader object and adds them to the ListBox control. The DataReader uses GetString(0) to get the first value from the row.

Each time the Read method is called, the DataReader gets another row back from the result set. A new item is added to the ListBox for each row that is read.

```
listEmployees.BeginUpdate();
while( reader.Read() ) {
     listEmployees.Items.Add( reader.GetString( 0 ) );
}
listEmployees.EndUpdate();
```

☞ For more information about the AsaDataReader object, see "AsaDataReader class" on page 418.

Finishing off   The following code at the end of the method closes the reader and connection objects.

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

Error handling   Any errors that occur during execution and that originate with Adaptive Server Anywhere .NET data provider objects are handled by displaying them in a message box. The following code catches the error and displays its message:

```
catch( AsaException ex ) {
    MessageBox.Show( ex.Errors[0].Message );
}
```

☞ For more information about the AsaException object, see "AsaException class" on page 437.

Tutorial: Using the Table Viewer code sample

This tutorial is based on the Table Viewer project that is included with the Adaptive Server Anywhere .NET data provider.

The complete application can be found in your SQL Anywhere installation directory at *Samples\ASA\ado.net\TableViewer\TableViewer.csproj.*

The Table Viewer project is more complex than the Simple project. It illustrates the following features:

♦ connecting to a database

♦ working with the AsaDataAdapter object

♦ more advanced error handling and result checking

☞ For more information about how the sample works, see "Understanding the Table Viewer sample project" on page 354.

❖ **To run the Table Viewer code sample in Visual Studio .NET**

1. Start Visual Studio .NET.

2. Choose File ➤ Open ➤ Project.

3. Browse to *SamplesASA\ado.net\TableViewer* in your SQL Anywhere installation directory and open the *TableViewer.csproj* project.

4. If you want to use the Adaptive Server Anywhere .NET data provider in a project, you must add a reference to the data provider DLL. This has already been done in the Table Viewer code sample. You can view the reference to the data provider DLL in the following location:

   ♦ In the Solution Explorer window, Open the References folder.

   ♦ You should see iAnywhere.Data.AsaClient in the list.
   ☞ For instructions about adding a reference to the data provider DLL, see "Adding a reference to the data provider DLL in your project" on page 358.

5. You must also add a using directive to your source code to reference the data provider classes. This has already been done in the Table Viewer code sample. To view the using directive:

   ♦ Open the source code for the project.

      • In the Solution Explorer window, select *TableViewer.cs.*

      • Choose View ➤ Code.

♦ In the `using` directives in the top section, you should see the following line:

```
using iAnywhere.Data.AsaClient;
```

This line is required for C# projects. If you are using Visual Basic .NET, you need to add an `Imports` line to your source code.

☞ For more information, see "Referencing the data provider classes in your source code" on page 359.

6. Choose Debug ➤ Start Without Debugging to run the Table Viewer project.

The Table Viewer dialog appears.

♦ In the Table Viewer dialog, click Connect.

The application connects to the Adaptive Server Anywhere sample database.

♦ In the Table Viewer dialog, click Execute.

The application retrieves the data from the employee table in the sample database and puts the query results in the Results DataList, as follows:

Table Viewer						_ □ ×
Connect string:						Connect
Data Source=ASA 8.0 Sample						Execute
SQL Statement:						
SELECT * FROM Employee						

Results						
	emp_id	manager_id	emp_fname	emp_lname	dept_id	street	city ▲
▶	102	501	Fran	Whitney	100	9 East Washi	Corr
	105	501	Matthew	Cobb	100	7 Pleasant St	Grim
	129	902	Philip	Chin	200	539 Pond Str	Oak
	148	1293	Julie	Jordan	300	1244 Great Pl	Woc
	160	501	Robert	Breault	100	358 Cherry St	Miltc
	184	1576	Melissa	Espinoza	400	1121 Apple T	Iroqu

If you attempt to execute the query without first connecting to a database, a message appears instructing you to connect to a database.

♦ You can also execute other SQL statements from this application: enter a SQL statement in the SQL Statement pane and then click Execute.

7. Click the X in the upper right corner of the screen to terminate the application and disconnect from the sample database. This also shuts down the database server.

You have now run the application. The next section describes the application code.

Understanding the Table Viewer sample project

This section illustrates some key features of the Adaptive Server Anywhere .NET data provider by walking through some of the code from the Table Viewer code sample. The Table Viewer project uses the Adaptive Server Anywhere sample database, *asademo.db*, which is held in your SQL Anywhere installation directory.

☞ For information about the sample database, including the tables in the database and the relationships between them, see "About the sample database" [*Introducing SQL Anywhere Studio,* page 198].

In this section the code is described a few lines at a time. Not all code from the sample is included here. To see the whole code, open the sample project at *Samples\ASA\ado.net\Tableviewer\TableViewer.csproj.*

Declaring controls   The following code declares a TextBox labeled Connection String, a button named btnConnect, a TextBox labeled txtSQLStatement, a button named btnExecute, and a DataGrid labeled dgResults.

```
Private System.Windows.Forms.Label label1;
private System.Windows.Forms.TextBox txtConnectString;
private System.Windows.Forms.Label label2;
private System.Windows.Forms.Button btnConnect;
private System.Windows.Forms.TextBox txtSQLStatement;
private System.Windows.Forms.Button btnExecute;
private System.Windows.Forms.DataGrid dgResults;
```

Declaring a global variable   The AsaConnection function is used to declare a global variable. This connection is used for the initial connection to the database, as well as when you click Execute to retrieve the result set from the database.

```
private AsaConnection     _conn;
```

☞ For more information about the AsaConnection function, see "AsaConnection constructors" on page 403.

Connecting to the database   The following code provides a default value for the connection string that appears in the Connection String field by default.

```
this.txtConnectString.Text =
  "Data Source=ASA 9.0 Sample";
```

The Connection object later uses the connection string (`"Data Source=ASA 9.0 Sample"`) to connect to the sample database.

```
_conn = new AsaConnection( txtConnectString.Text );
_conn.Open();
```

☞ For more information about the Connection object, see "AsaConnection class" on page 403.

Defining a query The following code defines the default query that appears in the SQL Statement field.

```
this.txtSQLStatement.Text = "SELECT * FROM employee";
```

Displaying the results Before the results are fetched, the application checks whether the Connection object has been initialized. If it has, it ensures that the connection state is open.

```
if( _conn == null || _conn.State !=
  ConnectionState.Open ) {
  MessageBox.Show( "Connect to a database first",
   "Not connected" );
  return;
```

Once you are connected to the database, the following code creates a new DataSet and uses the DataAdapter object (AsaDataAdapter) to execute a SQL statement (SELECT * FROM employee), and fill the DataSet. The last two lines bind the DataSet to the grid on the screen.

```
DataSet ds =new DataSet ();
AsaDataAdapter da=new AsaDataAdapter(
    txtSQLStatement.Text, _conn);
da.Fill(ds, "Results")
dgResults.DataSource = ds.Tables["Results"];
```

Because a global variable is used to declare the connection, the connection that was opened earlier is reused to execute the SQL statement.

☞ For more information about the DataAdapter object, see "AsaDataAdapter class" on page 409.

Error handling If there is an error when the application attempts to connect to the database, the following code catches the error and displays its message:

```
try {
    _conn = new AsaConnection( txtConnectString.Text );
    _conn.Open();
  } catch( AsaException ex ) {
    MessageBox.Show( ex.Errors[0].Source + " : "
     + ex.Errors[0].Message + " (" +
     ex.Errors[0].NativeError.ToString() + ")",
        "Failed to connect" );
}
```

Developing Applications with the .NET Data Provider

About this chapter
This chapter describes how to develop and deploy applications with the Adaptive Server Anywhere .NET data provider.

Contents

Using the .NET provider in a Visual Studio .NET project

Once you have installed the Adaptive Server Anywhere .NET data provider, you must make two changes to your Visual Studio .NET project to be able to use it:

♦ add a reference to the Adaptive Server Anywhere .NET data provider DLL

♦ add a line to your source code to reference the Adaptive Server Anywhere .NET data provider classes

☞ For information about installing and registering the Adaptive Server Anywhere .NET data provider, see "Deploying the Adaptive Server Anywhere .NET data provider" on page 389.

Adding a reference to the data provider DLL in your project

Adding a reference tells Visual Studio .NET which DLL to include to find the code for the Adaptive Server Anywhere .NET data provider.

❖ **To add a reference to the Adaptive Server Anywhere .NET data provider in a Visual Studio .NET project**

1. Start Visual Studio .NET and open your project.

2. In the Solution Explorer window, right-click the References folder and choose Add Reference from the popup menu.

    The Add Reference dialog appears.

3. On the .NET tab, click Browse to locate *iAnywhere.Data.AsaClient.dll*. (The default location is *\Program Files\Sybase\SQL Anywhere 9\win32*). Select the DLL and click Open.

    Note that there is a separate version of the DLL for each of Windows and Windows CE.

    ☞ For a complete list of installed DLLs, see "Adaptive Server Anywhere .NET data provider required files" on page 389.

4. You can verify that the DLL is added to your project. Open the Add Reference dialog and then click the .NET tab. *iAnywhere.Data.AsaClient.dll* appears in the Selected Components list. Click OK to close the dialog.

    The DLL is added to the References folder in the Solution Explorer window of your project.

Referencing the data provider classes in your source code

In order to use the Adaptive Server Anywhere .NET data provider, you must also add a line to your source code to reference the data provider. You must add a different line for C# than for Visual Basic .NET.

❖ To reference the data provider classes in your code

1. Start Visual Studio .NET and open your project.

2. If you are using C#, add the following line to the list of `using` directives at the beginning of your project:

   ```
   using iAnywhere.Data.AsaClient;
   ```

3. If you are using Visual Basic .NET, add the following line at the beginning of your project before the line `Public Class Form1`:

   ```
   Imports iAnywhere.Data.AsaClient
   ```

   This line is not strictly required. However, it allows you to use short forms for the Adaptive Server Anywhere classes. Without it, you can still use

   ```
   iAnywhere.Data.AsaClient.AsaConnection
    conn = new iAnywhere.Data.AsaClient.AsaConnection()
   ```

   instead of

   ```
   AsaConnection    conn = new AsaConnection()
   ```

   in your code.

Connecting to a database

Before you can carry out any operations on the data, your application must connect to the database. This section describes how to write code to connect to an Adaptive Server Anywhere database.

☞ For more information, see "AsaConnection class" on page 403 and "ConnectionString property" on page 404.

❖ **To connect to an Adaptive Server Anywhere database**

1. Allocate an AsaConnection object.

   The following code creates an AsaConnection object named conn:

   ```
   AsaConnection conn = new AsaConnection(
   ```

   You can have more than one connection to a database from your application. Some applications use a single connection to an Adaptive Server Anywhere database, and keep the connection open all the time. To do this, you can declare a global variable for the connection:

   ```
   private AsaConnection      _conn;
   ```

   ☞ For more information, see the sample code in *Samples\ASA\ado.net\TableViewer\TableViewer.csproj* and "Understanding the Table Viewer sample project" on page 354.

2. Specify the connection string used to connect to the database.

   For example:

   ```
   "Data Source=ASA 9.0 Sample;UID=DBA;PWD=SQL" );
   ```

   ☞ For a complete list of connection parameters, see "Connection parameters" [*ASA Database Administration Guide,* page 176].

   Instead of supplying a connection string, you could prompt users for their user ID and password if you wish.

3. Open a connection to the database.

   The following code attempts to connect to a database. It autostarts the database server if necessary.

   ```
   conn.Open();
   ```

4. Catch connection errors.

   Your application should be designed to catch any errors that occur when attempting to connect to the database. The following code demonstrates how to catch an error and display its message:

```
try {
    _conn = new AsaConnection( txtConnectString.Text );
    _conn.Open();
} catch( AsaException ex ) {
    MessageBox.Show( ex.Errors[0].Source + " : "
      + ex.Errors[0].Message + " (" +
      ex.Errors[0].NativeError.ToString() + ")",
        "Failed to connect" );
```

Alternately, you can use the ConnectionString property to set the
connection string, rather than passing the connection string when the
AsaConnection object is created:

```
AsaConnection    _conn;
    _conn = new AsaConnection();
    _conn.ConnectionString =
      "Data Source=ASA 9.0 Sample;UID=DBA;PWD=SQL";
    _conn.Open();
```

5. Close the connection to the database. Connections to the database stay
    open until they are explicitly closed using the conn.Close() method.

Visual Basic .NET
connection example

The following Visual Basic .NET code opens a connection to the Adaptive
Server Anywhere sample database:

```
Private Sub Button1_Click(ByVal sender As _
    System.Object, ByVal e As System.EventArgs) _
    Handles Button1.Click
    ' Declare the connection object
    Dim myConn As New _
      iAnywhere.Data.AsaClient.AsaConnection()
    myConn.ConnectionString = _
      "Data Source=ASA 9.0 Sample;UID=DBA;PWD=SQL"
    myConn.Open()
    myConn.Close()
End Sub
```

Connection pooling

The Adaptive Server Anywhere .NET provider supports connection pooling.
Connection pooling allows your application to reuse existing connections
from a pool by saving the connection handle to a pool so it can be reused,
rather than repeatedly creating a new connection to the database. Connection
pooling is turned on by default.

The pool size is set in your connection string using the POOLING option.
You can also specify the minimum and maximum pool sizes. For example,

```
"Data Source=ASA 9.0 Sample;UID=DBA;PWD=SQL;POOLING=TRUE;Max
        Pool Size=50;Min Pool Size=5"
```

361

When your application first attempts to connect to the database, it checks the pool for an existing connection that uses the same connection parameters you have specified. If a matching connection is found, that connection is used. Otherwise, a new connection is used. When you disconnect, the connection is returned to the pool so that it can be reused.

☞ For more information about connection pooling, see "ConnectionString property" on page 404.

Checking the connection state

Once your application has established a connection to the database, you can check the connection state to ensure that the connection is open before you fetch data from the database to update it. If a connection is lost or is busy, or if another command is being processed, you can return an appropriate message to the user.

The AsaConnection class has a state property that checks the state of the connection. Possible state values are Open and Closed.

The following code checks whether the Connection object has been initialized, and if it has, it ensures that the connection is open. A message is returned to the user if the connection is not open.

```
if( _conn == null || _conn.State !=
  ConnectionState.Open ) {
  MessageBox.Show( "Connect to a database first",
   "Not connected" );
  return;
```

☞ For more information, see "State property" on page 407.

Accessing and manipulating data

With the Adaptive Server Anywhere .NET data provider, there are two ways you can access data: using the AsaCommand object or using the AsaDataAdapter object.

♦ **AsaCommand object**   The AsaCommand object is the recommended way of accessing and manipulating data in .NET.

The AsaCommand object allows you to execute SQL statements that retrieve or modify data directly from the database. Using the AsaCommand object, you can issue SQL statements and call stored procedures directly against the database.

Within an AsaCommand object, the AsaDataReader is used to return read-only result sets from a query or stored procedure. The AsaDataReader returns only one row at a time, but this does not degrade performance because the Adaptive Server Anywhere client-side libraries use prefetch buffering to prefetch several rows at a time.

Using the AsaCommand object allows you to group your changes into transactions rather than operating in autocommit mode. When you use the AsaTransaction object, locks are placed on the rows so that other users cannot modify them.

☞ For more information, see "AsaCommand class" on page 393 and "AsaDataReader class" on page 418.

♦ **AsaDataAdapter object**   The AsaDataAdapter object retrieves the entire result set into a DataSet. A DataSet is a disconnected store for data that is retrieved from a database. You can then edit the data in the DataSet and when you are finished, the AsaDataAdapter object updates the database with the changes made to the DataSet. When you use the AsaDataAdapter, there is no way to prevent other users from modifying the rows in your DataSet. You need to include logic within your application to resolve any conflicts that may occur.

☞ For more information about conflicts, see "Resolving conflicts when using the AsaDataAdapter" on page 372.

☞ For more information about the AsaDataAdapter object, see "AsaDataAdapter class" on page 409.

There is no performance impact from using the AsaDataReader within an AsaCommand object to fetch rows from the database rather than the AsaDataAdapter object.

Using the AsaCommand object to retrieve and manipulate data

The following sections describe how to retrieve data and how to insert, update, or delete rows using the AsaDataReader.

Getting data using the AsaCommand object

The AsaCommand object allows you to issue a SQL statement or call a stored procedure against an Adaptive Server Anywhere database. You can issue the following types of commands to retrieve data from the database:

♦ **ExecuteReader**   Used to issue a command that returns a result set. This method uses a forward-only, read-only cursor. You can loop quickly through the rows of the result set only in one direction.

♦ **ExecuteScalar**   Used to issue a command that returns a single value. This can be the first column in the first row of the result set, or a SQL statement that returns an aggregate value such as COUNT or AVG. This method uses a forward-only, read-only cursor.

When using the AsaCommand object you can use the AsaDataReader to retrieve a result set that is based on a join. However, you can only make changes (inserts, updates, or deletes) to data that is from a single table. You cannot update result sets that are based on joins.

The following instructions use the Simple code sample included with the .NET data provider.

❖ **To issue a command that returns a complete result set**

1. Declare and initialize a Connection object.

   ```
   AsaConnection conn = new AsaConnection(
       "Data Source=ASA 9.0 Sample;UID=DBA;PWD=SQL" );
   ```

2. Open the connection.

   ```
   try {
       conn.Open();
   ```

3. Add a Command object to define and execute a SQL statement.

```
AsaCommand cmd = new AsaCommand(
     "select emp_lname from employee", conn );
```

If you are calling a stored procedure, you must specify the parameters for the stored procedure.

☞ For more information, see "Using stored procedures" on page 384 and "AsaParameter class" on page 441.

4. Call the ExecuteReader method to return the DataReader object.

```
AsaDataReader reader = cmd.ExecuteReader();
```

5. Display the results.

```
listEmployees.BeginUpdate();
while( reader.Read() ) {
     listEmployees.Items.Add( reader.GetString( 0 ) );
}
listEmployees.EndUpdate();
```

6. Close the DataReader and Connection objects.

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

❖ **To issue a command that returns only one value**

1. Declare and initialize an AsaConnection object.

```
AsaConnection conn = new AsaConnection(
     "Data Source=ASA 9.0 Sample" );
```

2. Open the connection.

```
conn.Open();
```

3. Add an AsaCommand object to define and execute a SQL statement.

```
AsaCommand cmd = new AsaCommand(
     "select count(*) from employee where sex = 'M'",
     conn );
```

If you are calling a stored procedure, you must specify the parameters for the stored procedure.

☞ For more information, see "Using stored procedures" on page 384.

4. Call the ExecuteScalar method to return the object containing the value.

```
int count = (int) cmd.ExecuteScalar();
```

5. Close the AsaConnection object.

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

When using the AsaDataReader, there are several Get methods available that you can use to return the results in desired the data type.

☞ For more information, see "AsaDataReader class" on page 418.

Visual Basic .NET DataReader example

The following Visual Basic .NET code opens a connection to the Adaptive Server Anywhere sample database and uses the DataReader to return the last name of the first five employees in the result set:

```
Dim myConn As New .AsaConnection()
Dim myCmd As _
  New .AsaCommand _
  ("select emp_lname from employee", myConn)
Dim myReader As AsaDataReader
Dim counter As Integer
myConn.ConnectionString = _
  "Data Source=ASA 9.0 Sample;UID=DBA;PWD=SQL"
myConn.Open()
myReader = myCmd.ExecuteReader()
counter = 0
Do While (myReader.Read())
  MsgBox(myReader.GetString(0))
  counter = counter + 1
  If counter >= 5 Then Exit Do
Loop
myConn.Close()
```

Inserting, updating, and deleting rows using the AsaCommand object

In order to perform an insert, update, or delete with the AsaCommand object, you use the ExecuteNonQuery function. The ExecuteNonQuery function issues a command (SQL statement or stored procedure) that does not return a result set.

☞ For more information, see "ExecuteNonQuery method" on page 396.

You can only make changes (inserts, updates, or deletes) to data that is from a single table. You cannot update result sets that are based on joins. You must be connected to a database to use the AsaCommand object.

☞ For information about obtaining primary key values for autoincrement primary keys, see "Obtaining primary key values" on page 378.

If you want to set the isolation level for a command, you must use the AsaCommand object as part of an AsaTransaction object. When you modify data without an AsaTransaction object, the .NET provider operates in autocommit mode and any changes that you make are applied immediately.

☞ For more information, see "Transaction processing" on page 386.

❖ **To issue a command that inserts a row**

1. Declare and initialize an AsaConnection object.

```
AsaConnection   conn = new AsaConnection(
    c_connStr );
```

2. Open the connection.

```
conn.Open();
```

3. Add an AsaCommand object to define and execute an INSERT statement.

   You can use an INSERT, UPDATE, or DELETE statement with the ExecuteNonQuery method.

```
AsaCommand      insertCmd = new AsaCommand(
    "INSERT INTO department( dept_id, dept_name )
    VALUES( ?, ? )", conn );
```

   If you are calling a stored procedure, you must specify the parameters for the stored procedure.

   ☞ For more information, see "Using stored procedures" on page 384 and "AsaParameter class" on page 441.

4. Set the parameters for the AsaCommand object.

   The following code defines parameters for the dept_id and dept_name columns respectively.

```
AsaParameter parm = new AsaParameter();
parm.AsaDbType = AsaDbType.Integer;
insertCmd.Parameters.Add( parm );
parm = new AsaParameter();
parm.AsaDbType = AsaDbType.Char;
insertCmd.Parameters.Add( parm );
```

5. Insert the new values and call the ExecuteNonQuery method to apply the changes to the database.

```
insertCmd.Parameters[0].Value = 600;
insertCmd.Parameters[1].Value = "Eastern Sales";
int recordsAffected = insertCmd.ExecuteNonQuery();
insertCmd.Parameters[0].Value = 700;
insertCmd.Parameters[1].Value = "Western Sales";
int recordsAffected = insertCmd.ExecuteNonQuery();
```

6. Display the results and bind them to the grid on the screen.

```
AsaCommand        selectCmd = new AsaCommand(
    "SELECT * FROM department", conn );
AsaDataReader   dr = selectCmd.ExecuteReader();
dataGrid.DataSource = dr;
```

7. Close the AsaDataReader and AsaConnection objects.

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

❖ **To issue a command that updates a row**

1. Declare and initialize an AsaConnection object.

```
AsaConnection   conn = new AsaConnection(
    c_connStr );
```

2. Open the connection.

```
conn.Open();
```

3. Add an AsaCommand object to define and execute an UPDATE statement.

You can use an INSERT, UPDATE, or DELETE statement with the ExecuteNonQuery method.

```
AsaCommand        updateCmd = new AsaCommand(
    "UPDATE department SET dept_name = 'Engineering'
    WHERE dept_id=100", conn );
```

If you are calling a stored procedure, you must specify the parameters for the stored procedure.

☞ For more information, see "Using stored procedures" on page 384 and "AsaParameter class" on page 441.

4. Call the ExecuteNonQuery method to apply the changes to the database.

```
int recordsAffected = updateCmd.ExecuteNonQuery();
```

5. Display the results and bind them to the grid on the screen.

```
AsaCommand        selectCmd = new AsaCommand(
    "SELECT * FROM department", conn );
AsaDataReader   dr = selectCmd.ExecuteReader();
dataGrid.DataSource = dr;
```

6. Close the AsaDataReader and AsaConnection objects.

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

❖ **To issue a command that deletes a row**

1. Declare and initialize an AsaConnection object.

   ```
   AsaConnection   conn = new AsaConnection(
       c_connStr );
   ```

2. Open the connection.

   ```
   conn.Open();
   ```

3. Create an AsaCommand object to define and execute a DELETE statement.

   You can use an INSERT, UPDATE, or DELETE statement with the ExecuteNonQuery method.

   ```
   AsaCommand      deleteCmd = new AsaCommand(
       "DELETE FROM department WHERE ( dept_id > 500 )", conn
           );
   ```

   If you are calling a stored procedure, you must specify the parameters for the stored procedure.

   ☞ For more information, see "Using stored procedures" on page 384 and "AsaParameter class" on page 441.

4. Call the ExecuteNonQuery method to apply the changes to the database.

   ```
   int recordsAffected = deleteCmd.ExecuteNonQuery();
   ```

5. Close the AsaConnection object.

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

Obtaining DataReader schema information

You can obtain schema information about columns in the result set.

If you are using the AsaDataReader, you can use the GetSchemaTable method to obtain information about the result set. The GetSchemaTable method returns the standard .NET DataTable object, which provides information about all the columns in the result set, including column properties.

☞ For more information about the GetSchemaTable method, see "GetSchemaTable method" on page 426.

❖ **To obtain information about a result set using the GetSchemaTable method**

1. Declare and initialize a connection object.

```
AsaConnection conn = new AsaConnection(
    c_connStr );
```

2. Open the connection.

```
conn.Open();
```

3. Create an AsaCommand object with the SELECT statement you want to use. The schema is returned for the result set of this query.

```
AsaCommand cmd = new AsaCommand(
    "SELECT * FROM employee", conn );
```

4. Create an AsaDataReader object and execute the Command object you created.

```
AsaDataReader  dr = cmd.ExecuteReader();
```

5. Fill the DataTable with the schema from the data source.

```
DataTable       schema = dr.GetSchemaTable();
```

6. Close the AsaDataReader and AsaConnection objects.

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

7. Bind the DataTable to the grid on the screen.

```
dataGrid.DataSource = schema;
```

Using the AsaDataAdapter object to access and manipulate data

The following sections describe how to retrieve data and how to insert, update, or delete rows using the AsaDataAdapter.

Getting data using the AsaDataAdapter object

The AsaDataAdapter allows you to view the entire result set by using the Fill method to fill a DataSet with the results from a query by binding the DataSet to the display grid.

Using the AsaDataAdapter, you can pass any string (SQL statement or stored procedure) that returns a result set. When you use the AsaDataAdapter, all the rows are fetched in one operation using a

forward-only, read-only cursor. Once all the rows in the result set have been read, the cursor is closed. The AsaDataAdapter allows you to make changes to the DataSet. Once your changes are complete, you must reconnect to the database to apply the changes.

You can use the AsaDataAdapter object to retrieve a result set that is based on a join. However, you can only make changes (inserts, updates, or deletes) to data that is from a single table. You cannot update result sets that are based on joins.

Caution

Any changes you make to the DataSet are made while you are disconnected. This means that your application does not have locks on these rows in the database. Your application must be designed to resolve any conflicts that may occur when changes from the DataSet are applied to the database in the event that another user changes the data you are modifying before your changes are applied to the database.

☞ For more information about the AsaDataAdapter, see "AsaDataAdapter class" on page 409.

AsaDataAdapter
example

The following example shows how to fill a DataSet using the AsaDataAdapter.

❖ **To retrieve data using the AsaDataAdapter object**

1. Connect to the database.

2. Create a new DataSet. In this case, the DataSet is called Results.

   ```
   DataSet ds =new DataSet ();
   ```

3. Create a new AsaDataAdapter object to execute a SQL statement and fill the DataSet.

   ```
   AsaDataAdapter da=new AsaDataAdapter(
       txtSQLStatement.Text, _conn);
   da.Fill(ds, "Results")
   ```

4. Bind the DataSet to the grid on the screen.

   ```
   dgResults.DataSource = ds.Tables["Results"]
   ```

Inserting, updating, and deleting rows using the AsaDataAdapter object

The AsaDataAdapter retrieves the result set into a DataSet. A DataSet is a collection of tables and the relationships and constraints between those

tables. The DataSet is built into the .NET Framework, and is independent of the data provider used to connect to your database.

When you use the AsaDataAdapter, you must be connected to the database to fill the DataSet and to update the database with changes made to the DataSet. However, once the DataSet is filled, you can modify the DataSet while disconnected from the database.

If you do not want to apply your changes to the database right away, you can write the DataSet, including the data and/or the schema, to an XML file using the WriteXML method. Then, you apply the changes at a later time by loading a DataSet with the ReadXML method.

☞ For more information, see the .NET Framework documentation for WriteXML and ReadXML.

When you call the Update method to apply changes from the DataSet to the database, the AsaDataAdapter analyzes the changes that have been made and then invokes the appropriate commands, INSERT, UPDATE, or DELETE, as necessary. When you use the DataSet, you can only make changes (inserts, updates, or deletes) to data that is from a single table. You cannot update result sets that are based on joins. If another user has a lock on the row you are trying to update, an exception is thrown.

> **Caution**
> Any changes you make to the DataSet are made while you are disconnected. This means that your application does not have locks on these rows in the database. Your application must be designed to resolve any conflicts that may occur when changes from the DataSet are applied to the database in the event that another user changes the data you are modifying before your changes are applied to the database.

Resolving conflicts when using the AsaDataAdapter

When you use the AsaDataAdapter, no locks are placed on the rows in the database. This means there is the potential for conflicts to arise when you apply changes from the DataSet to the database. Your application should include logic to resolve or log conflicts that arise.

Some of the conflicts that your application logic should address include:

♦ **Unique primary keys** If two users insert new rows into a table, each row must have a unique primary key. For tables with autoincrement primary keys, the values in the DataSet may become out of sync with the values in the data source.

☞ For information about obtaining primary key values for autoincrement primary keys, see "Obtaining primary key values" on page 378.

♦ **Updates made to the same value**   If two users modify the same value, your application should include logic to determine which value is correct.

♦ **Schema changes**   If a user modifies the schema of a table you have updated in the DataSet, the update will fail when you apply the changes to the database.

♦ **Data concurrency**   Concurrent applications should see a consistent set of data. The AsaDataAdapter does not place a lock on rows that it fetches, so another user can update a value in the database once you have retrieved the DataSet and are working offline.

Many of these potential problems can be avoided by using the AsaCommand, AsaDataReader, and AsaTransaction objects to apply changes to the database. The AsaTransaction object is recommended because it allows you to set the isolation level for the transaction and it places locks on the rows so that other users cannot modify them.

☞ For more information about using transactions to apply your changes to the database, see "Inserting, updating, and deleting rows using the AsaCommand object" on page 366.

To simplify the process of conflict resolution, you can design your insert, update, or delete statement to be a stored procedure call. By including INSERT, UPDATE, and DELETE statements in stored procedures, you can catch the error if the operation fails. In addition to the statement, you can add error handling logic to the stored procedure so that if the operation fails the appropriate action is taken, such as recording the error to a log file, or trying the operation again.

❖ **To insert rows into a table using the AsaDataAdapter**

1. Declare and initialize an AsaConnection object.

   ```
   AsaConnection   conn = new AsaConnection(
       c_connStr );
   ```

2. Open the connection.

   ```
   conn.Open();
   ```

3. Create a new AsaDataAdapter object.

   ```
   AsaDataAdapter adapter = new AsaDataAdapter();
   adapter.MissingMappingAction =
       MissingMappingAction.Passthrough;
   adapter.MissingSchemaAction =
       MissingSchemaAction.Add;
   ```

4. Create the necessary AsaCommand objects and define any necessary parameters.

   The following code creates a SELECT and an INSERT command and defines the parameters for the INSERT command.

```
adapter.SelectCommand = new AsaCommand(
    "SELECT * FROM department", conn );
adapter.InsertCommand = new AsaCommand(
    "INSERT INTO department( dept_id, dept_name )
    VALUES( ?, ? )", conn );
adapter.InsertCommand.UpdatedRowSource =
    UpdateRowSource.None;
AsaParameter parm = new AsaParameter();
parm.AsaDbType = AsaDbType.Integer;
parm.SourceColumn = "dept_id";
parm.SourceVersion = DataRowVersion.Current;
adapter.InsertCommand.Parameters.Add(
    parm );
parm = new AsaParameter();
parm.AsaDbType = AsaDbType.Char;
parm.SourceColumn = "dept_name";
parm.SourceVersion = DataRowVersion.Current;
adapter.InsertCommand.Parameters.Add( parm );
```

5. Fill the DataTable with the results of the SELECT statement.

```
DataTable dataTable = new DataTable( "department" );
int rowCount = adapter.Fill( dataTable );
```

6. Insert the new rows into the DataTable and apply the changes to the database.

```
DataRow row1 = dataTable.NewRow();
row1[0] = 600;
row1[1] = "Eastern Sales";
dataTable.Rows.Add( row1 );
DataRow row2 = dataTable.NewRow();
row2[0] = 700;
row2[1] = "Western Sales";
dataTable.Rows.Add( row2 );
recordsAffected = adapter.Update( dataTable );
```

7. Display the results of the updates.

```
dataTable.Clear();
rowCount = adapter.Fill( dataTable );
dataGrid.DataSource = dataTable;
```

8. Close the connection.

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

❖ **To update rows using the AsaDataAdapter object**

1. Declare and initialize an AsaConnection object.

```
AsaConnection conn = new AsaConnection( c_connStr );
```

2. Open the connection.

```
conn.Open();
```

3. Create a new AsaDataAdapter object.

```
AsaDataAdapter adapter = new AsaDataAdapter();
adapter.MissingMappingAction =
    MissingMappingAction.Passthrough;
adapter.MissingSchemaAction =
    MissingSchemaAction.Add;
```

4. Create an AsaCommand object and define its parameters.

The following code creates a SELECT and an UPDATE command and defines the parameters for the UPDATE command.

```
adapter.SelectCommand = new AsaCommand(
    "SELECT * FROM department WHERE dept_id > 500",
    conn );
adapter.UpdateCommand = new AsaCommand(
    "UPDATE department SET dept_name = ?
    WHERE dept_id = ?", conn );
adapter.UpdateCommand.UpdatedRowSource =
    UpdateRowSource.None;
AsaParameter parm = new AsaParameter();
parm.AsaDbType = AsaDbType.Char;
parm.SourceColumn = "dept_name";
parm.SourceVersion = DataRowVersion.Current;
adapter.UpdateCommand.Parameters.Add( parm );
parm = new AsaParameter();
parm.AsaDbType = AsaDbType.Integer;
parm.SourceColumn = "dept_id";
parm.SourceVersion = DataRowVersion.Original;
adapter.UpdateCommand.Parameters.Add( parm );
```

5. Fill the DataTable with the results of the SELECT statement.

```
DataTable dataTable = new DataTable( "department" );
int rowCount = adapter.Fill( dataTable );
```

6. Update the DataTable with the updated values for the rows and apply the changes to the database.

```
foreach ( DataRow row in dataTable.Rows )
{
row[1] = ( string ) row[1] + "_Updated";
}
recordsAffected = adapter.Update( dataTable );
```

7. Bind the results to the grid on the screen.

```
dataTable.Clear();
adapter.SelectCommand.CommandText =
    "SELECT * FROM department";
rowCount = adapter.Fill( dataTable );
dataGrid.DataSource = dataTable;
```

8. Close the connection.

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

❖ **To delete rows from a table using the AsaDataAdapter object**

1. Declare and initialize an AsaConnection object.

```
AsaConnection conn = new AsaConnection( c_connStr );
```

2. Open the connection.

```
conn.Open();
```

3. Create an AsaDataAdapter object.

```
AsaDataAdapter adapter = new AsaDataAdapter();
adapter.MissingMappingAction =
    MissingMappingAction.Passthrough;
adapter.MissingSchemaAction =
    MissingSchemaAction.AddWithKey;
```

4. Create the required AsaCommand objects and define any necessary parameters.

   The following code creates a SELECT and a DELETE command and defines the parameters for the DELETE command.

```
adapter.SelectCommand = new AsaCommand(
    "SELECT * FROM department WHERE dept_id > 500",
    conn );
adapter.DeleteCommand = new AsaCommand(
    "DELETE FROM department WHERE dept_id = ?",
    conn );
adapter.DeleteCommand.UpdatedRowSource =
    UpdateRowSource.None;
AsaParameter parm = new AsaParameter();
parm.AsaDbType = AsaDbType.Integer;
parm.SourceColumn = "dept_id";
parm.SourceVersion = DataRowVersion.Original;
adapter.DeleteCommand.Parameters.Add( parm );
```

5. Fill the DataTable with the results of the SELECT statement.

```
DataTable dataTable = new DataTable( "department" );
int rowCount = adapter.Fill( dataTable );
```

6. Modify the DataTable and apply the changes to the database.

```
for each ( DataRow in dataTable.Rows )
{
    row.Delete();
}
recordsAffected = adapter.Update( dataTable )
```

7. Bind the results to the grid on the screen.

```
dataTable.Clear();
rowCount = adapter.Fill( dataTable );
dataGrid.DataSource = dataTable;
```

8. Close the connection.

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

Obtaining AsaDataAdapter schema information

When using the AsaDataAdapter, you can use the FillSchema method to obtain schema information about the result set in the DataSet. The FillSchema method returns the standard .NET DataTable object, which provides the names of all the columns in the result set.

☞ For more information about the FillSchema method, see .

❖ **To obtain DataSet schema information using the FillSchema method**

1. Declare and initialize an AsaConnection object.

```
AsaConnection   conn = new AsaConnection(
    c_connStr );
```

2. Open the connection.

```
conn.Open();
```

3. Create an AsaDataAdapter with the SELECT statement you want to use. The schema is returned for the result set of this query.

```
AsaDataAdapter  adapter = new AsaDataAdapter(
    "SELECT * FROM employee", conn );
```

4. Create a new DataTable object, in this case called Table, to fill with the schema.

```
DataTable       dataTable = new DataTable(
    "Table" );
```

5. Fill the DataTable with the schema from the data source.

```
adapter.FillSchema( dataTable, SchemaType.Source );
```

6. Close the AsaConnection object.

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

7. Bind the DataSet to the grid on the screen.

```
dataGrid.DataSource = dataTable;
```

Obtaining primary key values

If the table you are updating has an autoincremented primary key, uses UUIDs, or if the primary key comes from a primary key pool, you can use a stored procedure to obtain values generated by the data source.

When using the AsaDataAdapter, this technique can be used to fill the columns in the DataSet with the primary key values generated by the data source. If you want to use this technique with the AsaCommand object, you can either get the key columns from the parameters or reopen the DataReader.

Examples   The following examples use a table called adodotnet_primarykey that contains two columns, id and name. The primary key for the table is id. It is an INTEGER and contains an autoincremented value. The name column is CHAR(40).

These examples call the following stored procedure to retrieve the autoincremented primary key value from the database.

```
CREATE PROCEDURE sp_adodotnet_primarykey( out p_id int, in p_
      name char(40) )
BEGIN
    INSERT INTO adodotnet_primarykey( name ) VALUES(
      p_name );
    SELECT @@IDENTITY INTO p_id;
END
```

❖ **To insert a new row with an autoincremented primary key using the AsaCommand object**

1. Connect to the database.

```
AsaConnection   conn = OpenConnection();
```

2. Create a new AsaCommand object to insert new rows into the DataTable. In the following code, the line int id1 = (int) parmId.Value; verifies the primary key value of the row.

```
AsaCommand      cmd = conn.CreateCommand();
cmd.CommandText = "sp_adodotnet_primarykey";
cmd.CommandType = CommandType.StoredProcedure;
AsaParameter parmId = new AsaParameter();
parmId.AsaDbType = AsaDbType.Integer;
parmId.Direction = ParameterDirection.Output;
cmd.Parameters.Add( parmId );
AsaParameter parmName = new AsaParameter();
parmName.AsaDbType = AsaDbType.Char;
parmName.Direction = ParameterDirection.Input;
cmd.Parameters.Add( parmName );
parmName.Value = "R & D --- Command";
cmd.ExecuteNonQuery();
int id1 = ( int ) parmId.Value;
parmName.Value = "Marketing --- Command";
cmd.ExecuteNonQuery();
int id2 = ( int ) parmId.Value;
parmName.Value = "Sales --- Command";
cmd.ExecuteNonQuery();
int id3 = ( int ) parmId.Value;
parmName.Value = "Shipping --- Command";
cmd.ExecuteNonQuery();
int id4 = ( int ) parmId.Value;
```

3. Bind the results to the grid on the screen and apply the changes to the
    database.

```
cmd.CommandText = "select * from " +
    adodotnet_primarykey";
cmd.CommandType = CommandType.Text;
AsaDataReader dr = cmd.ExecuteReader();
dataGrid.DataSource = dr;
```

4. Close the connection.

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

❖ **To insert a new row with an autoincremented primary key using
    the AsaDataAdapter object**

1. Create a new AsaDataAdapter.

```
DataSet         dataSet = new DataSet();
AsaConnection   conn = OpenConnection();
AsaDataAdapter  adapter = new AsaDataAdapter();
adapter.MissingMappingAction =
    MissingMappingAction.Passthrough;
adapter.MissingSchemaAction =
    MissingSchemaAction.AddWithKey;
```

2. Fill the data and schema of the DataSet. The SelectCommand is called by
    the AsaDataAdapter.Fill method to do this. You can also create the

DataSet manually without using the Fill method and SelectCommand if
you do not need the existing records.

```
adapter.SelectCommand = new AsaCommand( "select * from  +
        adodotnet_primarykey", conn );
```

3. Create a new AsaCommand to obtain the primary key values from the
   database.

```
adapter.InsertCommand = new AsaCommand(
    "sp_adodotnet_primarykey", conn );
adapter.InsertCommand.CommandType =
    CommandType.StoredProcedure;
adapter.InsertCommand.UpdatedRowSource =
    UpdateRowSource.OutputParameters;
AsaParameter parmId = new AsaParameter();
parmId.AsaDbType = AsaDbType.Integer;
parmId.Direction = ParameterDirection.Output;
parmId.SourceColumn = "id";
parmId.SourceVersion = DataRowVersion.Current;
adapter.InsertCommand.Parameters.Add( parmId );
AsaParameter parmName = new AsaParameter();
parmName.AsaDbType = AsaDbType.Char;
parmName.Direction = ParameterDirection.Input;
parmName.SourceColumn = "name";
parmName.SourceVersion = DataRowVersion.Current;
adapter.InsertCommand.Parameters.Add( parmName );
```

4. Fill the DataSet.

```
adapter.Fill( dataSet );
```

5. Insert the new rows into the DataSet.

```
DataRow row = dataSet.Tables[0].NewRow();
row[0] = -1;
row[1] = "R & D --- Adapter";
dataSet.Tables[0].Rows.Add( row );
row = dataSet.Tables[0].NewRow();
row[0] = -2;
row[1] = "Marketing --- Adapter";
dataSet.Tables[0].Rows.Add( row );
row = dataSet.Tables[0].NewRow();
row[0] = -3;
row[1] = "Sales --- Adapter";
dataSet.Tables[0].Rows.Add( row );
row = dataSet.Tables[0].NewRow();
row[0] = -4;
row[1] = "Shipping --- Adapter";
dataSet.Tables[0].Rows.Add( row );
```

6. Apply the changes in the DataSet to the database. When the Update()
   method is called, the primary key values are changed to the values
   obtained from the database.

```
adapter.Update( dataSet );
dataGrid.DataSource = dataSet.Tables[0];
```

When you add new rows to the DataTable and call the Update method, the AsaDataAdapter calls the InsertCommand and maps the output parameters to the key columns for each new row. The Update method is called only once, but the InsertCommand is called by the Update method as many times as necessary for each new row being added.

7. Close the connection to the database.

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

Handling BLOBs

When fetching long string values or binary data, there are methods that you can use to fetch the data in pieces. For binary data, use the GetBytes method, and for string data, use the GetChars method. Otherwise, BLOB data is treated in the same manner as any other data you fetch from the database.

☞ For more information, see "GetBytes method" on page 420 and "GetChars method" on page 421.

❖ **To issue a command that returns a string using the GetChars method**

1. Declare and initialize a Connection object.

2. Open the connection.

3. Add a Command object to define and execute a SQL statement.

```
AsaCommand cmd = new AsaCommand(
    "select int_col, blob_col from test", conn );
```

4. Call the ExecuteReader method to return the DataReader object.

```
AsaDataReader reader = cmd.ExecuteReader();
```

The following code reads the two columns from the result set. The first column is an integer (GetInt32(0)), while the second column is a LONG VARCHAR. GetChars is used to read 100 characters at a time from the LONG VARCHAR column.

```
int length = 100;
char[] buf = new char[ length ];
int intValue;
long dataIndex = 0;
long charsRead = 0;
long blobLength = 0;
while( reader.Read() ) {
    intValue = reader.GetInt32( 0 );
    while ( ( charsRead = reader.GetChars(
        1, dataIndex, buf, 0, length ) ) == ( long )
            length ) {
    dataIndex += length;
}
blobLength = dataIndex + charsRead;
}
```

5. Close the DataReader and Connection objects.

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

Obtaining time values

The .NET Framework does not have a Time structure. If you want to fetch time values from Adaptive Server Anywhere, you must use the GetTimeSpan method. Using this method returns the data as a .NET Framework TimeSpan object.

☞ For more information about the GetTimeSpan method, see

❖ **To convert a time value using the GetTimeSpan method**

1. Declare and initialize a connection object.

```
AsaConnection conn = new AsaConnection(
    "Data Source=dsn-time-test;uid=dba;pwd=sql" );
```

2. Open the connection.

```
conn.Open();
```

3. Add a Command object to define and execute a SQL statement.

```
AsaCommand cmd = new AsaCommand(
    "SELECT id, time_col FROM time_test", conn )
```

4. Call the ExecuteReader method to return the DataReader object.

```
AsaDataReader reader = cmd.ExecuteReader();
```

The following code uses the GetTimeSpan method to return the time as TimeSpan.

```
while ( reader.Read() )
{
    int id = reader.GetInt32();
    TimeSpan time = reader.GetTimeSpan();
}
```

5. Close the DataReader and Connection objects.

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

Using stored procedures

You can use stored procedures with the .NET data provider. The ExecuteReader method is used to call stored procedures that return a result set, while the ExecuteNonQuery method is used to call stored procedures that do not return a result set. The ExecuteScalar method is used to call stored procedures that return only a single value.

When you call a stored procedure, you must create an AsaParameter object. Use a question mark as a placeholder for parameters, as follows:

```
sp_producttype( ?, ? )
```

☞ For more information about the Parameter object, see "AsaParameter class" on page 441.

❖ **To execute a stored procedure**

1. Declare and initialize an AsaConnection object.

```
AsaConnection conn = new AsaConnection(
    "Data Source=ASA 9.0 Sample" );
```

2. Open the connection.

```
conn.Open();
```

3. Add an AsaCommand object to define and execute a SQL statement. The following code uses the CommandType property to identify the command as a stored procedure.

```
AsaCommand cmd = new AsaCommand( "sp_product_info",
    conn );
cmd.CommandType = CommandType.StoredProcedure;
```

If you do not specify the CommandType property, then you must use a question mark as a placeholder for parameters, as follows:

```
AsaCommand cmd = new AsaCommand(
    "call sp_product_info(?)", conn );
cmd.CommandType = CommandType.Text;
```

4. Add an AsaParameter object to define the parameters for the stored procedure. You must create a new AsaParameter object for each parameter the stored procedure requires.

```
AsaParameter param = cmd.CreateParameter();
param.AsaDbType = AsaDbType.Int32;
param.Direction = ParameterDirection.Input;
param.Value = 301;
cmd.Parameters.Add( param );
```

☞ For more information about the Parameter object, see "AsaParameter class" on page 441.

5. Call the ExecuteReader method to return the DataReader object. The Get methods are used to return the results in the desired data type.

```
AsaDataReader reader = cmd.ExecuteReader();
reader.Read();
int id = reader.GetInt32(0);
string name = reader.GetString(1);
string descrip = reader.GetString(2);
decimal price = reader.GetDecimal(6);
```

6. Close the AsaDataReader and AsaConnection objects.

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

Alternative way to call a stored procedure

Step 3 in the above instructions presents two ways you can call a stored procedure. Another way you can call a stored procedure, without using a Parameter object, is to call the stored procedure from your source code, as follows:

```
AsaCommand cmd = new AsaCommand(
    "call sp_product_info( 301 )", conn );
```

☞ For information about calling stored procedures that return a result set or a single value, see "Getting data using the AsaCommand object" on page 364.

☞ For information about calling stored procedures that do not return a result set, see "Inserting, updating, and deleting rows using the AsaCommand object" on page 366.

Transaction processing

With the Adaptive Server Anywhere .NET provider, you can use the AsaTransaction object to group statements together. Each transaction ends with a COMMIT or ROLLBACK, which either makes your changes to the database permanent or cancels all the operations in the transaction. Once the transaction is complete, you must create a new AsaTransaction object to make further changes. This behavior is different from ODBC and embedded SQL, where a transaction persists after you execute a COMMIT or ROLLBACK until the transaction is closed.

If you do not create a transaction, the Adaptive Server Anywhere .NET provider operates in autocommit mode by default. There is an implicit COMMIT after each insert, update, or delete, and once an operation is completed, the change is made to the database. In this case, the changes cannot be rolled back.

☞ For more information about the AsaTransaction object, see "AsaTransaction class" on page 459.

Setting the isolation level for transactions   The database isolation level is used by default for transactions. However, you may choose to specify the isolation level for a transaction using the IsolationLevel property when you begin the transaction. The isolation level applies to all commands executed within the transaction.

☞ For more information about isolation levels, see "Isolation levels and consistency" [*ASA SQL User's Guide,* page 106].

The locks that Adaptive Server Anywhere uses when you enter a SELECT statement depend on the transaction's isolation level.

☞ For more information about locking and isolation levels, see "Locking during queries" [*ASA SQL User's Guide,* page 138].

☞ The following example uses an AsaTransaction object to issue and then roll back a SQL statement. The transaction uses isolation level 2 (RepeatableRead), which places a write lock on the row being modified so that no other database user can update the row.

❖ **To use an AsaTransaction object to issue a command**

1. Declare and initialize an AsaConnection object.

    ```
    AsaConnection conn = new AsaConnection(
        "Data Source=ASA 9.0 Sample" );
    ```

2. Open the connection.

    ```
    conn.Open();
    ```

3. Issue a SQL statement to change the price of Tee shirts.

```
string stmt = "update product set unit_price =
    2000.00 where name = 'Tee shirt'";
```

4. Create an AsaTransaction object to issue the SQL statement using a Command object.

Using a transaction allows you to specify the isolation level. Isolation level 2 (RepeatableRead) is used in this example so that another database user cannot update the row.

```
AsaTransaction trans = conn.BeginTransaction(
    IsolationLevel.RepeatableRead );
AsaCommand cmd = new AsaCommand( stmt, conn,
    trans );
int rows = cmd.ExecuteNonQuery();
```

5. Roll back the changes.

```
trans.Rollback();
```

The AsaTransaction object allows you to commit or roll back your changes to the database. If you do not use a transaction, the .NET data provider operates in autocommit mode and you cannot roll back any changes that you make to the database. If you want to make the changes permanent, you would use the following:

```
trans.Commit();
```

6. Close the AsaConnection object.

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

Error handling and the Adaptive Server Anywhere .NET data provider

Your application must be designed to handle any errors that occur, including ADO.NET errors. ADO.NET errors are handled within your code in the same way that you handle other errors in your application.

The Adaptive Server Anywhere .NET data provider throws AsaException objects whenever errors occur during execution. Each AsaException object consists of a list of AsaError objects, and these error objects include the error message and code.

Errors are different from conflicts. Conflicts arise when changes are applied to the database. Your application should include a process to compute correct values or to log conflicts when they arise.

☞ For more information about handling conflicts, see "Resolving conflicts when using the AsaDataAdapter" on page 372.

.NET provider error handling example

The following example is from the Simple sample project. Any errors that occur during execution and that originate with Adaptive Server Anywhere .NET data provider objects are handled by displaying them in a message box. The following code catches the error and displays its message:

```
catch( AsaException ex ) {
    MessageBox.Show( ex.Errors[0].Message );
}
```

Connection error handling example

The following example is from the Table Viewer sample project. If there is an error when the application attempts to connect to the database, the following code uses a try and catch block to catch the error and display its message:

```
try {
    _conn = new AsaConnection( txtConnectString.Text );
    _conn.Open();
} catch( AsaException ex ) {
    MessageBox.Show( ex.Errors[0].Source + " : "
     + ex.Errors[0].Message + " (" +
     ex.Errors[0].NativeError.ToString() + ")",
        "Failed to connect" );
```

☞ For more error handling examples, see "Understanding the Simple sample project" on page 349 and "Understanding the Table Viewer sample project" on page 354.

☞ For more information about error handling, see "AsaException class" on page 437 and "AsaError class" on page 433.

Deploying the Adaptive Server Anywhere .NET data provider

The following sections describe how to deploy the Adaptive Server Anywhere .NET data provider.

Adaptive Server Anywhere .NET data provider system requirements

In order to use the Adaptive Server Anywhere .NET data provider, you must have the following installed on your machine or handheld device:

♦ the .NET Framework and/or .NET Compact Framework

♦ Visual Studio .NET 1.0, Visual Studio .NET 2003, or a .NET language compiler, such as C# (required only for development)

Adaptive Server Anywhere .NET data provider required files

The Adaptive Server Anywhere .NET data provider consists of two DLLs for each platform.

Windows required files
For Windows (except Windows CE) the following DLLs are required:

♦ *win32\iAnywhere.Data.AsaClient.dll*

♦ *win32\dbdata9.dll*

The first DLL (*iAnywhere.Data.AsaClient.dll*) is the main DLL that is referenced by Visual Studio projects. The second DLL (*dbdata9.dll*) contains utility code.

Windows CE required files
These files must be installed in your SQL Anywhere installation directory (the default location is *C:\Program Files\Sybase\SQL Anywhere 9\win32*) because they require the language DLLs that are also located in your SQL Anywhere installation directory.

For Windows CE, *iAnywhere.Data.AsaClient.dll* is the main DLL that is referenced by Visual Studio projects. There is a separate *dbdata9.dll* file for each of the supported Windows CE platforms. The Windows CE DLLs are:

♦ *ce\iAnywhere.Data.AsaClient.dll*

♦ *ce\arm.30\dbdata9.dll*

♦ *ce\emulator.30\dbdata9.dll*

♦ *ce\mips.30\dbdata9.dll*

♦ *ce\x86\dbdata9.dll*

The utility DLL (*dbdata9.dll*) must be placed in the Windows directory on your device. Visual Studio .NET deploys the .NET data provider DLL (*iAnywhere.Data.Asaclient.dll*) to your device along with your program. If you are not using Visual Studio .NET, you need to copy the data provider DLL to the device along with your program. It can go in the same directory as your application, or in the Windows directory.

Registering the Adaptive Server Anywhere .NET data provider DLL

The Adaptive Server Anywhere .NET data provider DLL (*win32\iAnywhere.Data.AsaClient.dll*) needs to be registered in the Global Assembly Cache on Windows (except Windows CE). The Global Assembly Cache lists all the registered programs on your machine. When you install the .NET data provider, the .NET data provider installation program registers it. On Windows CE you do not need to register the DLL.

If you are deploying the .NET data provider, you must register the .NET data provider DLL (*win32\iAnywhere.Data.AsaClient.dll*) using the *gacutil* utility that is included with the .NET Framework.

Adaptive Server Anywhere .NET Data Provider API Reference

About this chapter

This chapter describes the API for the Adaptive Server Anywhere .NET data provider.

Note: Many of the properties and methods in this chapter are very similar to the OLE DB .NET data provider. You can find more information and examples in the Microsoft .NET Framework documentation.

Contents

AsaCommand class

Description	A SQL statement or stored procedure that is executed against an Adaptive Server Anywhere database.
Base classes	Component
Implemented interfaces	IDbCommand, IDisposable
See also	"Using the AsaCommand object to retrieve and manipulate data" on page 364
	"Accessing and manipulating data" on page 363

AsaCommand constructors

Description	Initializes an AsaCommand object.
Syntax 1	void **AsaCommand( )**
Syntax 2	void **AsaCommand(** string *cmdText* **)**
Syntax 3	void **AsaCommand(** string *cmdText*, AsaConnection *connection* **)**
Syntax 4	void **AsaCommand(** string *cmdText*, AsaConnection *connection*, AsaTransaction *transaction* **)**
Parameters	**cmdText** The text of the SQL statement or stored procedure. For parameterized statements, use a question mark (?) placeholder to pass parameters.
	connection The current connection.
	transaction The AsaTransaction in which the AsaConnection executes.

Cancel method

Description	Cancels the execution of an AsaCommand object.
Syntax	void **Cancel( )**
Usage	If there is nothing to cancel, nothing happens. If there is a command in process and the attempt to cancel fails, no exception is generated.
Implements	IDbCommand.Cancel

CommandText property

Description	The text of a SQL statement or stored procedure.
Syntax	string **CommandText**
Access	Read-write
Property value	The SQL statement or the name of the stored procedure to execute. The default is an empty string.
Implements	IDbCommand.CommandText
See also	"AsaCommand constructors" on page 393

CommandTimeout property

Description	The wait time in seconds before terminating an attempt to execute a command and generating an error.
Syntax	int **CommandTimeout**
Access	Read-write
Implements	IDbCommand.CommandTimeout
Default	30 seconds
Usage	A value of 0 indicates no limit. This should be avoided because it may cause the attempt to execute a command to wait indefinitely.

CommandType property

Description	The type of command represented by an AsaCommand.
Syntax	CommandType **CommandType**
Access	Read-write
Implements	IDbCommand.CommandType
Usage	Supported command types are as follows:

♦ **CommandType.StoredProcedure** When you specify this CommandType, the command text must be the name of a stored procedure and you must supply any arguments as AsaParameter objects.

♦ **CommandType.Text** This is the default value.

When the CommandType property is set to StoredProcedure, the CommandText property should be set to the name of the stored procedure. The command executes this stored procedure when you call one of the Execute methods.

Use a question mark (?) placeholder to pass parameters. For example:

```
SELECT * FROM Customers WHERE CustomerID = ?
```

The order in which AsaParameter objects are added to the AsaParameterCollection must directly correspond to the position of the question mark placeholder for the parameter.

Connection property

Description	The connection object to which the AsaCommand object applies.
Syntax	AsaConnection **Connection**
Access	Read-write
Default	The default value is a null reference. In Visual Basic it is Nothing.

CreateParameter method

Description	Provides an AsaParameter object for supplying parameters to AsaCommand objects.
Syntax	AsaParameter **CreateParameter( )**
Return value	A new parameter, as an AsaParameter object.
Usage	Stored procedures and some other SQL statements can take parameters, indicated in the text of a statement by a question mark (?).
	The CreateParameter method provides an AsaParameter object. You can set properties on the AsaParameter to specify the value, data type, and so on for the parameter.
See also	"AsaParameter class" on page 441

DesignTimeVisible property

Description	A value that indicates if the AsaCommand should be visible in a customized Windows Form Designer control. The default is false.
Syntax	bool **DesignTimeVisible**
Access	Read-write

ExecuteNonQuery method

Description	Executes a statement that does not return a result set, such as an INSERT, UPDATE, DELETE, or a data definition statement.
Syntax	int **ExecuteNonQuery( )**
Return value	The number of rows affected.
Implements	IDbCommand.ExecuteNonQuery
Usage	You can use ExecuteNonQuery to change the data in a database without using a DataSet. Do this by executing UPDATE, INSERT, or DELETE statements.
	Although ExecuteNonQuery does not return any rows, output parameters or return values that are mapped to parameters are populated with data.
	For UPDATE, INSERT, and DELETE statements, the return value is the number of rows affected by the command. For all other types of statements, and for rollbacks, the return value is -1.

ExecuteReader method

Description	Executes a SQL statement that returns a result set.
Syntax 1	AsaDataReader **ExecuteReader( )**
Syntax 2	AsaDataReader **ExecuteReader(CommandBehavior** *behavior* **)**
Parameters	**behavior** One of **CloseConnection**, **Default**, **KeyInfo**, **SchemaOnly**, **SequentialAccess**, **SingleResult**, or **SingleRow**.
	☞ For more information about this parameter, see the .NET Framework documentation for CommandBehavior Enumeration.
Return value	The result set as an AsaDataReader object.
Usage	The statement is the current AsaCommand object, with CommandText and Parameters as needed. The AsaDataReader object is a read-only, forward-only result set. For modifiable result sets, use an AsaDataAdapter.
See also	"AsaDataReader class" on page 418
	"AsaDataAdapter class" on page 409

ExecuteScalar method

Description	Executes a statement that returns a single value. If this method is called on a

query that returns multiple rows and columns, only the first column of the first row is returned.

Syntax	object **ExecuteScalar( )**
Return Value	The first column of the first row in the result set, or a null reference if the result set is empty.
Implements	IDbCommand.ExecuteScalar

Parameters property

Description	A collection of parameters for the current statement. Use question marks in the CommandText to indicate parameters.
Syntax	AsaParameterCollection **Parameters**
Access	Read-only
Property Value	The parameters of the SQL statement or stored procedure. The default value is an empty collection.
Usage	When CommandType is set to Text, pass parameters using the question mark placeholder. For example:

```
SELECT * FROM Customers WHERE CustomerID = ?
```

The order in which AsaParameter objects are added to the AsaParameterCollection must directly correspond to the position of the question mark placeholder for the parameter in the command text.

When the parameters in the collection do not match the requirements of the query to be executed, an error may result or an exception may be thrown.

See also	"AsaParameterCollection class" on page 447

Prepare method

Description	Prepares or compiles the AsaCommand on the data source.
Syntax	void **Prepare( )**
Implements	IDbCommand.Prepare
Usage	Before you call Prepare, specify the data type of each parameter in the statement to be prepared.

If you call an Execute method after calling Prepare, any parameter value that is larger than the value specified by the Size property is automatically truncated to the original specified size of the parameter, and no truncation

errors are returned.

Output parameters (whether prepared or not) must have a user-specified data type.

ResetCommandTimeout method

Description	Resets the CommandTimeout property to its default value of 30 seconds.
Syntax	void **ResetCommandTimeout( )**

Transaction property

Description	Associates the current command with a transaction.
Syntax	AsaTransaction **Transaction**
Access	Read-write
Usage	The default value is a null reference. In Visual Basic this is Nothing.
	You cannot set the Transaction property if it is already set to a specific value and the command is executing. If you set the transaction property to an AsaTransaction object that is not connected to the same AsaConnection as the AsaCommand object, an exception will be thrown the next time you attempt to execute a statement.
See also	"AsaTransaction class" on page 459
	"Transaction processing" on page 386

UpdatedRowSource property

Description	How command results are applied to the DataRow when used by the Update method of the AsaDataAdapter.
Syntax	UpdateRowSource **UpdatedRowSource**
Access	Read-write
Implements	IDbCommand.UpdatedRowSource
Property value	One of the UpdatedRowSource values. If the command is automatically generated, the default is None. Otherwise, the default is Both.

AsaCommandBuilder class

Description	A way to generate *single-table* SQL statements that reconcile changes made to a DataSet with the data in the associated database.
Base classes	Component
Implemented interfaces	IDisposable

AsaCommandBuilder constructors

Description	Initializes an AsaCommandBuilder object.
Syntax 1	void **AsaCommandBuilder( )**
Syntax 2	void **AsaCommandBuilder(** AsaDataAdapter *adapter* **)**
Parameters	**adapter** An AsaDataAdapter object for which to generate reconciliation statements.

DataAdapter property

Description	The AsaDataAdapter for which to generate statements.
Syntax	AsaDataAdapter **DataAdapter**
Access	Read-write
Property value	An AsaDataAdapter object.
Usage	When you create a new instance of AsaCommandBuilder, any existing AsaCommandBuilder that is associated with this AsaDataAdapter is released.

DeriveParameters method

Description	Populates the Parameters collection of the specified AsaCommand object. This is used for the stored procedure specified in the AsaCommand.
Syntax	void **DeriveParameters(** AsaCommand *command* **)**
Parameters	**command** An AsaCommand object for which to derive parameters.
Usage	DeriveParameters overwrites any existing parameter information for the AsaCommand.
	DeriveParameters requires an extra call to the database server. If the parameter information is known in advance, it is more efficient to populate the Parameters collection by setting the information explicitly.

GetDeleteCommand method

Description The generated AsaCommand object that performs DELETE operations on the database when AsaDataAdapter.Update() is called.

Syntax AsaCommand **GetDeleteCommand( )**

Return value The automatically generated AsaCommand object required to perform deletions.

Usage The GetDeleteCommand method returns the AsaCommand object to be executed, so it may be useful for informational or troubleshooting purposes.

You can also use GetDeleteCommand as the basis of a modified command. For example, you might call GetDeleteCommand and modify the CommandTimeout value, and then explicitly set that value on the AsaDataAdapter.

SQL statements are first generated when the application calls Update or GetDeleteCommand. After the SQL statement is first generated, the application must explicitly call RefreshSchema if it changes the statement in any way. Otherwise, the GetDeleteCommand will still be using information from the previous statement.

GetInsertCommand method

Description The generated AsaCommand object that performs INSERT operations on the database when an AsaDataAdapter.Update() is called.

Syntax AsaCommand **GetInsertCommand( )**

Return value The automatically generated AsaCommand object required to perform insertions.

Usage The GetInsertCommand method returns the AsaCommand object to be executed, so it may be useful for informational or troubleshooting purposes.

You can also use GetInsertCommand as the basis of a modified command. For example, you might call GetInsertCommand and modify the CommandTimeout value, and then explicitly set that value on the AsaDataAdapter.

SQL statements are first generated either when the application calls Update or GetInsertCommand. After the SQL statement is first generated, the application must explicitly call RefreshSchema if it changes the statement in any way. Otherwise, the GetInsertCommand will be still be using information from the previous statement, which might not be correct.

GetUpdateCommand method

Description   The generated AsaCommand object that performs UPDATE operations on
                    the database when an AsaDataAdapter.Update() is called.

Syntax   AsaCommand **GetUpdateCommand( )**

Return value   The automatically generated AsaCommand object required to perform
                    updates.

Usage   The GetUpdateCommand method returns the AsaCommand object to be
                    executed, so it may be useful for informational or troubleshooting purposes.

                    You can also use GetUpdateCommand as the basis of a modified command.
                    For example, you might call GetUpdateCommand and modify the
                    CommandTimeout value, and then explicitly set that value on the
                    AsaDataAdapter.

                    SQL statements are first generated when the application calls Update or
                    GetUpdateCommand. After the SQL statement is first generated, the
                    application must explicitly call RefreshSchema if it changes the statement in
                    any way. Otherwise, the GetUpdateCommand will be still be using
                    information from the previous statement, which might not be correct.

QuotePrefix property

Description   The beginning character or characters to use when specifying database
                    object names that contain characters such as spaces.

Syntax   string **QuotePrefix**

Access   Read-write

Property value   The beginning character or characters to use. This can be square brackets,
                    or, if the Adaptive Server Anywhere QUOTED_IDENTIFIER option is set
                    to off, it can be double quotes. The default is an empty string.

Usage   Adaptive Server Anywhere objects can contain characters such as spaces,
                    commas, and semicolons. The QuotePrefix and QuoteSuffix properties
                    specify delimiters to encapsulate the object name.

                    Although you cannot change the QuotePrefix or QuoteSuffix properties after
                    an INSERT, UPDATE, or DELETE operation, you can change their settings
                    after calling the Update method of a DataAdapter.

See also   "Identifiers" [*ASA SQL Reference,* page 7]

                    "QUOTED_IDENTIFIER option [compatibility]" [*ASA Database*

QuoteSuffix property

Description	The ending character or characters to use when specifying database objects whose names contain characters such as spaces.
Syntax	string **QuoteSuffix**
Access	Read-write
Property value	The ending character or characters to use. This can be square brackets, or, if the Adaptive Server Anywhere QUOTED_IDENTIFIER option is set to off, it can be double quotes. The default is an empty string.
Usage	Adaptive Server Anywhere objects can contain characters such as spaces, commas, and semicolons. The QuotePrefix and QuoteSuffix properties specify delimiters to encapsulate the object name.
	Although you cannot change the QuotePrefix or QuoteSuffix properties after an INSERT, UPDATE, or DELETE operation, you can change their settings after calling the Update method of a DataAdapter.
See also	"Identifiers" [*ASA SQL Reference,* page 7]
	"QUOTED_IDENTIFIER option [compatibility]" [*ASA Database Administration Guide,* page 685]

RefreshSchema method

Description	Refreshes the database schema information used to generate INSERT, UPDATE, or DELETE statements.
Syntax	void **RefreshSchema( )**
Usage	Call RefreshSchema whenever the SelectCommand value of the associated AsaDataAdapter changes.

AsaConnection class

Description   Represents a connection to an Adaptive Server Anywhere database.

Base classes   Component

Implemented interfaces IDbConnection, IDisposable

See also   "Connecting to a database" on page 360

AsaConnection constructors

Description   Initializes an AsaConnection object. The connection must then be opened before you can carry out any operations against the database.

Syntax 1   void **AsaConnection( )**

Syntax 2   void **AsaConnection(** string *connectionString* **)**

Parameters   **connectionString** An Adaptive Server Anywhere connection string. A connection string is a semicolon-separated list of keyword-value pairs.

      ☞ For a list of parameters, see "Connection parameters" [*ASA Database Administration Guide,* page 176].

Example   The following statement initializes an AsaConnection object for a connection to a database named policies running on an Adaptive Server Anywhere database server named hr. The connection uses a user ID of admin with a password of money.

```
AsaConnection conn = new AsaConnection(
"uid=admin;pwd=money;eng=hr;dbn=policies" );
conn.Open();
```

BeginTransaction method

Description   Returns a transaction object. Commands associated with a transaction object are executed as a single transaction. The transaction is terminated with a Commit() or Rollback().

Syntax 1   AsaTransaction **BeginTransaction( )**

Syntax 2   AsaTransaction **BeginTransaction(** IsolationLevel *isolationLevel* **)**

Parameters   **isolationLevel** A member of the IsolationLevel enumeration. The default value is ReadCommitted.

Return value   An object representing the new transaction.

Usage	To associate a command with a transaction object, use the AsaCommand.Transaction property.
| Example | ```
AsaTransaction tx = conn.BeginTransaction(
    IsolationLevel.ReadUncommitted );
``` |
See also	"Commit method" on page 459
	"Rollback method" on page 460
	"Transaction processing" on page 386
	"Typical types of inconsistency" [*ASA SQL User's Guide,* page 107]

ChangeDatabase method

Description	Changes the current database for an open AsaConnection.
Syntax	void **ChangeDatabase(** string *database* **)**
Parameters	**database** The name of the database to use instead of the current database.
Implements	IDbConnection.ChangeDatabase

Close method

Description	Closes a database connection.
Syntax	void **Close( )**
Implements	IDbConnection.Close
Usage	The Close method rolls back any pending transactions. It then releases the connection to the connection pool, or closes the connection if connection pooling is disabled. If Close is called while handling a StateChange event, no additional StateChange events are fired. An application can call Close more than one time.

ConnectionString property

Description	A database connection string.
Syntax	string **ConnectionString**
Access	Read-write
Implements	IDbConnection.ConnectionString
Usage	The default value of connection pooling is true (pooling=true).

The ConnectionString is designed to match the Adaptive Server Anywhere connection string format as closely as posssble with the following exception:

♦ When Persist Security Info value is set to false (the default), the connection string that is returned is the same as the user-set ConnectionString minus security information. The Adaptive Server Anywhere .NET data provider does not persist or return the password in a connection string unless you set Persist Security Info to true.

You can use the ConnectionString property to connect to a variety of data sources.

You can set the ConnectionString property only when the connection is closed. Many of the connection string values have corresponding read-only properties. When the connection string is set, all of these properties are updated, unless an error is detected. If an error is detected, none of the properties are updated. AsaConnection properties return only those settings contained in the ConnectionString.

If you reset the ConnectionString on a closed connection, all connection string values and related properties are reset, including the password.

When the property is set, a preliminary validation of the connection string is performed. When an application calls the Open method, the connection string is fully validated. A runtime exception is generated if the connection string contains invalid or unsupported properties.

Values may be delimited by single or double quotes. Either single or double quotes may be used within a connection string by using the other delimiter, for example, name="value's" or name= 'value"s', but not name='value's' or name= ""value"". Blank characters are ignored unless they are placed within a value or within quotes. Keyword-value pairs must be separated by a semicolon. If a semicolon is part of a value, it must also be delimited by quotes. Escape sequences are not supported, and the value type is irrelevant. Names are not case sensitive. If a property name occurs more than once in the connection string, the value associated with the last occurrence is used.

You should use caution when constructing a connection string based on user input, such as when retrieving a user ID and password from a dialog box, and appending it to the connection string. The application should not allow a user to embed extra connection string parameters in these values.

Example

The following statements set a connection string for an ODBC data source named **ASA 9.0 Sample** and open the connection.

```
AsaConnection conn = new AsaConnection();
conn.ConnectionString = "dsn=ASA 9.0 Sample";
conn.Open();
```

ConnectionTimeout property

Description	The number of seconds before a connection attempt times out with an error.
Syntax	int **ConnectionTimeout**
Access	Read-only
Default	15 seconds
Implements	IDbConnection.ConnectionTimeout
Example	The following statement displays the value of the ConnectionTimeout.

```
MessageBox.Show( conn.ConnectionTimeout.ToString() );
```

CreateCommand method

Description	Initializes an AsaCommand object. You can use the properties of the AsaCommand to control its behavior.
Syntax	AsaCommand **CreateCommand( )**
Return value	An AsaCommand object.
Usage	The command object is associated with the AsaConnection.

Database property

Description	The name of the current database to be used after a connection is opened.
Syntax	string **Database**
Access	Read-only
Implements	IDbConnection.Database
Usage	AsaConnection looks in the connection string in the following order: DatabaseName, dbn, DataSourceName, DataSource, dsn, DatabaseFile, dbf.

DataSource property

Description	The name of a running database server to which to connect.
Syntax	string **DataSource**
Access	Read-only
Usage	AsaConnection looks in the connection string in the following order:

Enginename, Servername, Eng.

InfoMessage event

Description	Occurs when the provider sends a warning or an informational message.
Syntax	event AsaInfoMessageEventHandler **InfoMessage**
Usage	The event handler receives an argument of type AsaInfoMessageEventArgs containing data related to this event. The following AsaInfoMessageEventArgs properties provide information specific to this event: ErrorCode, Errors, Message, and Source.

☞ For more information, see the .NET Framework documentation for OleDbConnection.InfoMessage Event.

Open method

Description	Opens a connection to a database, using the previously-specified connection string.
Syntax	void **Open( )**
Implements	IDbConnection.Open
Usage	The AsaConnection draws an open connection from the connection pool if one is available. Otherwise, it establishes a new connection to the data source.

If the AsaConnection goes out of scope, it is not closed. Therefore, you must explicitly close the connection by calling Close or Dispose.

ServerVersion property

Description	The software version of the Adaptive Server Anywhere database server.
Syntax	string **ServerVersion**
Access	Read-only
Usage	The version is *##.##.####*, where the first two digits are the major version, the next two digits are the minor version, and the last four digits are the release version. The appended string is of the form *major.minor.build*, where *major* and *minor* are two digits and *build* is four digits.

State property

Description	The current state of the connection.

Syntax	ConnectionState **State**
Access	Read-only
Default	The default value is Closed.
Implements	IDbConnection.State
See also	

StateChange event

Description	Occurs when the state of the connection changes.
Syntax	event StateChangeEventHandler **StateChange**
Usage	The event handler receives an argument of type StateChangeEventArgs with data related to this event. The following StateChangeEventArgs properties provide information specific to this event: **CurrentState** and **OriginalState**.
	☞ For more information, see the .NET Framework documentation for OleDbConnection.StateChange Event.

AsaDataAdapter class

Description	Represents a set of commands and a database connection used to fill a DataSet and to update a database.
Base classes	Component
Implemented interfaces	IDbDataAdapter, IDisposable
Usage	The DataSet provides a way to work with data offline. The AsaDataAdapter provides methods to associate a DataSet with a set of SQL statements.
See also	"Using the AsaDataAdapter object to access and manipulate data" on page 370
	"Accessing and manipulating data" on page 363

AsaDataAdapter constructors

Description	Initializes an AsaDataAdapter object.
Syntax 1	void **AsaDataAdapter( )**
Syntax 2	void **AsaDataAdapter(** AsaCommand *selectCommand* **)**
Syntax 3	void **AsaDataAdapter(** string *selectCommandText*, AsaConnection *selectConnection* **)**
Syntax 4	void **AsaDataAdapter(** string *selectCommandText*, string *selectConnectionString* **)**
Parameters	**selectCommand** An AsaCommand object that is used during Fill to select records from the data source for placement in the DataSet.
	selectCommandText A SELECT statement or stored procedure to be used by the SelectCommand property of the AsaDataAdapter.
	selectConnection An AsaConnection object that defines a connection to a database.
	selectConnectionString A connection string for an Adaptive Server Anywhere database.
Example	The following code initializes an AsaDataAdapter object:

```
AsaDataAdapter da = new AsaDataAdapter(
"SELECT emp_id, emp_lname FROM employee, conn );
```

AcceptChangesDuringFill property

Description	A value that indicates whether AcceptChanges is called on a DataRow after it is added to the DataTable.
Syntax	bool **AcceptChangesDuringFill**
Access	Read-write
Usage	When this property is true, DataAdapter calls the AcceptChanges function on the DataRow. If false, AcceptChanges is not called, and the newly added rows are treated as inserted rows. The default is true.

ContinueUpdateOnError property

Description	A value that specifies whether to generate an exception when an error is encountered during a row update.
Syntax	bool **ContinueUpdateOnError**
Access	Read-write
Usage	The default is false. Set this property to true to continue the update without generating an exception.
	If ContinueUpdateOnError is true, no exception is thrown when an error occurs during the update of a row. The update of the row is skipped and the error information is placed in the RowError property of the row. The DataAdapter continues to update subsequent rows.
	If ContinueUpdateOnError is false, an exception is thrown when an error occurs.

DeleteCommand property

Description	An AsaCommand object that is executed against the database when Update() is called to delete rows in the database that correspond to deleted rows in the DataSet.
Syntax	AsaCommand **DeleteCommand**
Access	Read-write
Usage	If this property is not set and primary key information is present in the DataSet during Update, DeleteCommand can be generated automatically by setting SelectCommand and using the AsaCommandBuilder. In that case, the AsaCommandBuilder generates any additional commands that you do not set. This generation logic requires key column information to be present

in the SelectCommand.

When DeleteCommand is assigned to an existing AsaCommand object, the AsaCommand object is not cloned. The DeleteCommand maintains a reference to the existing AsaCommand.

See also

Fill method

Description
Adds or refreshes rows in a DataSet or DataTable object with data from the database.

Syntax 1
int **Fill(** DataSet *dataSet* **)**

Syntax 2
int **Fill(**
DataSet *dataSet*,
string *srcTable*
)

Syntax 3
int **Fill(**
DataSet *dataSet*,
int *startRecord*,
int *maxRecords*,
string *srcTable*
)

Syntax 4
int **Fill(** DataTable *dataTable* **)**

Parameters
dataSet   A DataSet to fill with records and optionally schema.

srcTable   The name of the source table to use for table mapping.

startRecord   The zero-based record number to start with.

maxRecords   The maximum number of records to be read into the DataSet.

dataTable   A DataTable to fill with records and optionally schema.

Return Value
The number of rows successfully added or refreshed in the DataSet.

Usage
Even if you use the startRecord argument to limit the number of records that are copied to the DataSet, all records in the AsaDataAdapter query are fetched from the database to the client. For large result sets, this can have a significant performance impact.

An alternative is to use an AsaDataReader when a read-only, forward-only result set is sufficient, perhaps with SQL statements (ExecuteNonQuery) to carry out modifications. Another alternative is to write a stored procedure

that returns only the result you need.

If SelectCommand does not return any rows, no tables are added to the DataSet and no exception is raised.

See also "Getting data using the AsaDataAdapter object" on page 370

FillError event

Description   Returned when an error occurs during a fill operation.

Syntax   event FillErrorEventHandler **FillError**

Usage   The FillError event allows you to determine whether or not the fill operation should continue after the error occurs. Examples of when the FillError event might occur are:

♦ The data being added to a DataSet cannot be converted to a common language runtime type without losing precision.

♦ The row being added contains data that violates a Constraint that must be enforced on a DataColumn in the DataSet.

FillSchema method

Description   Adds DataTables to a DataSet and configures the schema to match the schema in the data source.

Syntax 1   DataTable[ ] **FillSchema(**
DataSet *dataSet*,
 SchemaType *schemaType*
)

Syntax 2   DataTable[ ] **FillSchema(**
DataSet *dataSet*,
SchemaType *schemaType*,
string *srcTable*
)

Syntax 3   DataTable **FillSchema(**
DataTable *dataTable*,
SchemaType *schemaType*
)

Parameters   **dataSet**   A DataSet to fill with records and optionally schema.

schemaType   One of the SchemaType values that specify how to insert the schema.

srcTable   The name of the source table to use for table mapping.

	dataTable A DataTable.
Return Value	For syntax 1 and 2, the return value is a reference to a collection of DataTable objects that were added to the DataSet. For syntax 3, the return value is a reference to a DataTable.
See also	"Obtaining AsaDataAdapter schema information" on page 377

GetFillParameters method

Description	The parameters set by the user when executing a SELECT statement.
Syntax	AsaParameter[ ] **GetFillParameters( )**
Return value	An array of IDataParameter objects that contains the parameters set by the user.
Implements	IDataAdapter.GetFillParameters

InsertCommand property

Description	An AsaCommand that is executed against the database when an Update() is called that adds rows to the database to correspond to rows that were inserted in the DataSet.
Syntax	AsaCommand **InsertCommand**
Access	Read-write
Usage	The AsaCommandBuilder does not require key columns to generate InsertCommand.
	When InsertCommand is assigned to an existing AsaCommand object, the AsaCommand is not cloned. The InsertCommand maintains a reference to the existing AsaCommand.
	If this command returns rows, the rows may be added to the DataSet depending on how you set the UpdatedRowSource property of the AsaCommand object.
See also	"Update method" on page 416
	"Inserting, updating, and deleting rows using the AsaCommand object" on page 366
	"Inserting, updating, and deleting rows using the AsaDataAdapter object" on page 371

MissingMappingAction property

Description	Determines the action to take when incoming data does not have a matching table or column.
Syntax	MissingMappingAction **MissingMappingAction**
Access	Read-write
Property Value	One of the MissingMappingAction values. The default is Passthrough.
Implements	IDataAdapter.MissingMappingAction

MissingSchemaAction property

Description	Determines the action to take when the existing DataSet schema does not match incoming data.
Syntax	MissingSchemaAction **MissingSchemaAction**
Access	Read-write
Property Value	One of the MissingSchemaAction values. The default is Add.
Implements	IDataAdapter.MissingSchemaAction

RowUpdated event

Description	Occurs during update after a command is executed against the data source. The attempt to update is made, so the event fires.
Syntax	event AsaRowUpdatedEventHandler **RowUpdated**
Usage	The event handler receives an argument of type AsaRowUpdatedEventArgs containing data related to this event. The following AsaRowUpdatedEventArgs properties provide information specific to this event:

- ♦ Command
- ♦ Errors
- ♦ RecordsAffected
- ♦ Row
- ♦ StatementType
- ♦ Status

♦ TableMapping

☞ For more information, see the .NET Framework documentation for OleDbDataAdapter.RowUpdated Event.

RowUpdating event

Description

Occurs during update before a command is executed against the data source. The attempt to update is made, so the event fires.

Syntax

event AsaRowUpdatingEventHandler **RowUpdating**

Usage

The event handler receives an argument of type AsaRowUpdatingEventArgs containing data related to this event. The following AsaRowUpdatingEventArgs properties provide information specific to this event:

♦ Command

♦ Errors

♦ Row

♦ StatementType

♦ Status

♦ TableMapping

☞ For more information, see the .NET Framework documentation for OleDbDataAdapter.RowUpdating Event.

SelectCommand property

Description

An AsaCommand that is used during Fill or FillSchema to obtain a result set from the database for copying into a DataSet.

Syntax

AsaCommand **SelectCommand**

Access

Read-write

Usage

When SelectCommand is assigned to a previously created AsaCommand, the AsaCommand is not cloned. The SelectCommand maintains a reference to the previously created AsaCommand object.

If the SelectCommand does not return any rows, no tables are added to the DataSet, and no exception is raised.

The SELECT statement can also be specified in the AsaDataAdapter constructor.

TableMappings property

Description	A collection that provides the master mapping between a source table and a DataTable.
Syntax	DataTableMappingCollection **TableMappings**
Access	Read-only
Usage	The default value is an empty collection.
	When reconciling changes, the AsaDataAdapter uses the DataTableMappingCollection collection to associate the column names used by the data source with the column names used by the DataSet.

Update method

Description	Updates the tables in a database with the changes made to the DataSet.
Syntax 1	int **Update(** DataSet *dataSet* **)**
Syntax 2	int **Update(** DataSet *dataSet*, string *srcTable* **)**
Syntax 3	int **Update(** DataTable *dataTable* **)**
Syntax 4	int **Update(** DataRow[ ] *dataRows* **)**
Parameters	**dataSet** A DataSet to update with records and optionally schema.
	srcTable The name of the source table to use for table mapping.
	dataTable A DataTable to update with records and optionally schema.
	dataRows An array of DataRow objects used to update the data source.
Return Value	The number of rows successfully updated from the DataSet.
Usage	The Update is carried out using the InsertCommand, UpdateCommand, and DeleteCommand on each row in the data set that has been inserted, updated, or deleted.
See also	

UpdateCommand property

Description
: An AsaCommand that is executed against the database when Update() is called to update rows in the database that correspond to updated rows in the DataSet.

Syntax
: AsaCommand **UpdateCommand**

Access
: Read-write

Usage
: During Update, if this property is not set and primary key information is present in the SelectCommand, the UpdateCommand can be generated automatically if you set the SelectCommand property and use the AsaCommandBuilder. Then, any additional commands that you do not set are generated by the AsaCommandBuilder. This generation logic requires key column information to be present in the SelectCommand.

: When UpdateCommand is assigned to a previously created AsaCommand, the AsaCommand is not cloned. The UpdateCommand maintains a reference to the previously created AsaCommand object.

: If execution of this command returns rows, these rows may be merged with the DataSet depending on how you set the UpdatedRowSource property of the AsaCommand object.

See also
:

AsaDataReader class

Description   A read-only, forward-only result set from a query or stored procedure.

Base classes   MarshalByRefObject

Implemented interfaces   IDataReader, IDisposable, IDataRecord

Usage   There is no constructor for AsaDataReader. To get an AsaDataReader object, execute an AsaCommand:

```
AsaCommand cmd = new AsaCommand(
    "Select emp_id from employee", conn );
AsaDataReader reader = cmd.ExecuteReader();
```

You can only move forward through an AsaDataReader. If you need a more flexible object to manipulate results, use an AsaDataAdapter.

The AsaDataReader retrieves rows as needed, whereas the AsaDataAdapter must retrieve all rows of a result set before you can carry out any action on the object. For large result sets, this difference gives the AsaDataReader a much faster response time.

See also   "ExecuteReader method" on page 396

                    "Accessing and manipulating data" on page 363

Close method

Description   Closes the AsaDataReader.

Syntax   void **Close( )**

Implements   IDataReader.Close

Usage   You must explicitly call the Close method when you are through using the AsaDataReader.

Depth property

Description   A value indicating the depth of nesting for the current row. The outermost table has a depth of zero.

Syntax   int **Depth**

Access   Read-only

Property Value   The depth of nesting for the current row.

Implements   IDataReader.Depth

Dispose method

Description	Frees the resources associated with the object.
Syntax	void **Dispose( )**

FieldCount property

Description	The number of columns in the result set.
Syntax	int **FieldCount**
Access	Read-only
Property Value	When not positioned in a valid record set, 0; otherwise the number of columns in the current record. The default is -1.
Implements	IDataRecord.FieldCount
Usage	When not positioned in a valid record set, this property has a value of 0; otherwise it is the number of columns in the current record. The default is -1. After executing a query that does not return rows, FieldCount returns 0.

GetBoolean method

Description	The value of the specified column as a Boolean.
Syntax	bool **GetBoolean(** int *ordinal* **)**
Parameters	**ordinal** An ordinal number indicating the column from which the value is obtained. The numbering is zero-based.
Return value	The value of the column.
Implements	IDataRecord.GetBoolean
Usage	No conversions are performed, so the data retrieved must already be a Boolean.

GetByte method

Description	The value of the specified column as a Byte.
Syntax	byte **GetByte(** int *ordinal* **)**
Parameters	**ordinal** An ordinal number indicating the column from which the value is obtained. The numbering is zero-based.
Return value	The value of the column.

Implements	IDataRecord.GetByte
Usage	No conversions are performed, so the data retrieved must already be a byte.

GetBytes method

Description	Reads a stream of bytes from the specified column offset into the buffer as an array starting at the given buffer offset.
Syntax	long **GetBytes(** int *ordinal*, long *dataIndex*, byte[ ] *buffer*, int *bufferIndex*, int *length* **)**
Parameters	**ordinal** An ordinal number indicating the column from which the value is obtained. The numbering is zero-based.
	dataIndex The index within the column value from which to read bytes.
	buffer An array in which to store the data.
	bufferIndex The index in the array to start copying data.
	length The maximum length to copy into the specified buffer.
Return value	The number of bytes read.
Implements	IDataRecord.GetBytes
Usage	GetBytes returns the number of available bytes in the field. In most cases this is the exact length of the field. However, the number returned may be less than the true length of the field if GetBytes has already been used to obtain bytes from the field. This may be the case, for example, when the AsaDataReader is reading a large data structure into a buffer.
	If you pass a buffer that is a null reference (Nothing in Visual Basic), GetBytes returns the length of the field in bytes.
	No conversions are performed, so the data retrieved must already be a byte array.

GetChar method

Description	The value of the specified column as a character.
Syntax	char **GetChar(** int *ordinal* **)**

Parameters	**ordinal** An ordinal number indicating the column from which the value is obtained. The numbering is zero-based.
Return value	The value of the column.
Implements	IDataRecord.GetChar
Usage	No conversions are performed, so the data retrieved must already be a character.
	Call AsaDataReader.IsDBNull to check for null values before calling this method.
See also	

GetChars method

Description	Reads a stream of characters from the specified column offset into the buffer as an array starting at the given buffer offset.
Syntax	long **GetChars(** int *ordinal*, long *dataIndex*, char[ ] *buffer*, int *bufferIndex*, int *length* **)**
Parameters	**ordinal** The zero-based column ordinal.
	dataIndex The index within the row from which to begin the read operation.
	buffer The buffer into which to copy data.
	bufferIndex The index for buffer to begin the read operation.
	length The number of characters to read.
Return value	The actual number of characters read.
Implements	IDataRecord.GetChars
Usage	GetChars returns the number of available characters in the field. In most cases this is the exact length of the field. However, the number returned may be less than the true length of the field if GetChars has already been used to obtain characters from the field. This may be the case, for example, when the AsaDataReader is reading a large data structure into a buffer.
	If you pass a buffer that is a null reference (Nothing in Visual Basic), GetChars returns the length of the field in characters.

No conversions are performed, so the data retrieved must already be a character array.

See also   "Handling BLOBs" on page 381

GetDataTypeName method

Description   The name of the source data type.

Syntax   string **GetDataTypeName(** int *index* **)**

Parameters   **index**   The zero-based column ordinal.

Return Value   The name of the back-end data type.

Implements   IDataRecord.GetDataTypeName

GetDateTime method

Description   The value of the specified column as a DateTime object.

Syntax   DateTime **GetDateTime(** int *ordinal* **)**

Parameters   **ordinal**   The zero-based column ordinal.

Return Value   The value of the specified column.

Implements   IDataRecord.GetDateTime

Usage   No conversions are performed, so the data retrieved must already be a DateTime object.

                   Call AsaDataReader.IsDBNull to check for null values before calling this method.

See also   "IsDBNull method" on page 430

GetDecimal method

Description   The value of the specified column as a Decimal object.

Syntax   decimal **GetDecimal(** int *ordinal* **)**

Parameters   **ordinal**   An ordinal number indicating the column from which the value is obtained. The numbering is zero-based.

Return Value   The value of the specified column.

Implements   IDataRecord.GetDecimal

Usage   No conversions are performed, so the data retrieved must already be a

Decimal object.

Call AsaDataReader.IsDBNull to check for null values before calling this method.

See also   "IsDBNull method" on page 430

GetDouble method

Description	The value of the specified column as a double-precision floating point number.
Syntax	double **GetDouble(** int *ordinal* **)**
Parameters	**ordinal** An ordinal number indicating the column from which the value is obtained. The numbering is zero-based.
Return Value	The value of the specified column.
Implements	IDataRecord.GetDouble
Usage	No conversions are performed, so the data retrieved must already be a double-precision floating point number.
	Call AsaDataReader.IsDBNull to check for null values before calling this method.
See also	"IsDBNull method" on page 430

GetFieldType method

Description	The Type that is the data type of the object.
Syntax	Type **GetFieldType(** int *index* **)**
Parameters	**index** The zero-based column ordinal.
Return Value	The type that is the data type of the object.
Implements	IDataRecord.GetFieldType

GetFloat method

Description	The value of the specified column as a single-precision floating point number.
Syntax	float **GetFloat(** int *ordinal* **)**
Parameters	**ordinal** An ordinal number indicating the column from which the value is obtained. The numbering is zero-based.

Return Value	The value of the specified column.
Implements	IDataRecord.GetFloat
Usage	No conversions are performed, so the data retrieved must already be a single-precision floating point number.
	Call AsaDataReader.IsDBNull to check for null values before calling this method.
See also	

GetGuid method

Description	The value of the specified column as a global unique identifier (GUID).
Syntax	Guid **GetGuid(** int *ordinal* **)**
Parameters	**ordinal** An ordinal number indicating the column from which the value is obtained. The numbering is zero-based.
Return Value	The value of the specified column.
Implements	IDataRecord.GetGuid
Usage	The data retrieved must already be a globally-unique identifier or binary(16).
	Call AsaDataReader.IsDBNull to check for null values before calling this method.
See also	

GetInt16 method

Description	The value of the specified column as a 16-bit signed integer.
Syntax	short **GetInt16(** int *ordinal* **)**
Parameters	**ordinal** An ordinal number indicating the column from which the value is obtained. The numbering is zero-based.
Return Value	The value of the specified column.
Implements	IDataRecord.GetInt16
Usage	No conversions are performed, so the data retrieved must already be a 16-bit signed integer.

GetInt32 method

Description	The value of the specified column as a 32-bit signed integer.

Syntax	int **GetInt32(** int *ordinal* **)**
Parameters	**ordinal** An ordinal number indicating the column from which the value is obtained. The numbering is zero-based.
Return Value	The value of the specified column.
Implements	IDataRecord.GetInt32
Usage	No conversions are performed, so the data retrieved must already be a 32-bit signed integer.

GetInt64 method

Description	The value of the specified column as a 64-bit signed integer.
Syntax	long **GetInt64(** int *ordinal* **)**
Parameters	**ordinal** An ordinal number indicating the column from which the value is obtained. The numbering is zero-based.
Return Value	The value of the specified column.
Implements	IDataRecord.GetInt64
Usage	No conversions are performed, so the data retrieved must already be a 64-bit signed integer.

GetName method

Description	The name of the specified column.
Syntax	string **GetName(** int *index* **)**
Parameters	**index** The zero-based index of the column.
Return value	The name of the specified column.
Implements	IDataRecord.GetName

GetOrdinal method

Description	The column ordinal, given the column name.
Syntax	int **GetOrdinal(** string *name* **)**
Parameters	**name** The column name.
Return Value	The zero-based column ordinal.
Implements	IDataRecord.GetOrdinal

Usage

GetOrdinal performs a case-sensitive lookup first. If it fails, a second case-insensitive search is made.

GetOrdinal is Japanese kana-width insensitive.

Because ordinal-based lookups are more efficient than named lookups, it is inefficient to call GetOrdinal within a loop. Save time by calling GetOrdinal once and assigning the results to an integer variable for use within the loop.

GetSchemaTable method

Description

Returns a DataTable that describes the column metadata of the AsaDataReader.

Syntax

DataTable **GetSchemaTable( )**

Return value

A DataTable that describes the column metadata.

Implements

IDataReader.GetSchemaTable

Usage

This method returns metadata about each column in the following order:

♦ ColumnName

♦ ColumnOrdinal

♦ ColumnSize

♦ NumericPrecision

♦ NumericScale

♦ IsUnique

♦ IsKey

♦ BaseCatalogName

♦ BaseColumnName

♦ BaseSchemaName

♦ BaseTableName

♦ DataType

♦ AllowDBNull

♦ ProviderType

♦ IsAliased

♦ IsExpression

♦ IsIdentity

♦ IsAutoIncrement

♦ IsRowVersion

♦ Is Hidden

♦ IsLong

♦ IsReadOnly

☞ For more information about these columns, see the .NET Framework documentation for SqlDataReader.GetSchemaTable.

See also   "Obtaining DataReader schema information" on page 369

GetString method

Description	The value of the specified column as a string.
Syntax	string **GetString(** int *ordinal* **)**
Parameters	**ordinal** An ordinal number indicating the column from which the value is obtained. The numbering is zero-based.
Return Value	The value of the specified column.
Implements	IDataRecord.GetString
Usage	No conversions are performed, so the data retrieved must already be a string.
	Call AsaDataReader.IsDBNull to check for null values before calling this method.
See also	"IsDBNull method" on page 430

GetTimeSpan method

Description	The value of the specified column as a TimeSpan object.
Syntax	TimeSpan **GetTimeSpan(** int *ordinal* **)**
Parameters	**ordinal** An ordinal number indicating the column from which the value is obtained. The numbering is zero-based.
Return Value	The value of the specified column.
Usage	The column must be ASA time data type. The data is converted to TimeSpan. The Days property of TimeSpan is always set to 0.

Call AsaDataReader.IsDBNull to check for null values before calling this
method.

See also

GetUInt16 method

Description	The value of the specified column as a 16-bit unsigned integer.
Syntax	UInt16 **GetUInt16(** int *ordinal* **)**
Parameters	**ordinal** An ordinal number indicating the column from which the value is obtained. The numbering is zero-based.
Return Value	The value of the specified column.
Usage	No conversions are performed, so the data retrieved must already be a 16-bit unsigned integer.

GetUInt32 method

Description	The value of the specified column as a 32-bit unsigned integer.
Syntax	UInt32 **GetUInt32(** int *ordinal* **)**
Parameters	**ordinal** An ordinal number indicating the column from which the value is obtained. The numbering is zero-based.
Return Value	The value of the specified column.
Usage	No conversions are performed, so the data retrieved must already be a 32-bit unsigned integer.

GetUInt64 method

Description	The value of the specified column as a 64-bit unsigned integer.
Syntax	UInt64 **GetUInt64(** int *ordinal* **)**
Parameters	**ordinal** An ordinal number indicating the column from which the value is obtained. The numbering is zero-based.
Return Value	The value of the specified column.
Usage	No conversions are performed, so the data retrieved must already be a 64-bit unsigned integer.

GetValue method

Description	The value of the column at the specified ordinal in its native format.

Syntax	object **GetValue(** int *ordinal* **)**
Parameters	**ordinal** An ordinal number indicating the column from which the value is obtained. The numbering is zero-based.
Return Value	The value to return.
Implements	IDataRecord.GetValue
Usage	This method returns DBNull for null database columns.

GetValues method

Description	All the attribute columns in the current row.
Syntax	int **GetValues(** object[ ] *values* **)**
Parameters	**values** An array of objects that holds an entire row of the result set.
Return value	The number of objects in the array.
Implements	IDataRecord GetValues
Usage	For most applications, the GetValues method provides an efficient means for retrieving all columns, rather than retrieving each column individually.
	You can pass an Object array that contains fewer than the number of columns contained in the resulting row. Only the amount of data the Object array holds is copied to the array. You can also pass an Object array whose length is more than the number of columns contained in the resulting row.
	This method returns DBNull for null database columns.
	Gets the value of the column at the specified ordinal in its native format.

IsClosed property

Description	Returns true if the AsaDataReader is closed. Otherwise it returns false.
Syntax	bool **IsClosed**
Access	Read-only
Property Value	True if the AsaDataReader is closed; otherwise, false.
Implements	IDataReader.IsClosed
Usage	IsClosed and RecordsAffected are the only properties that you can call after the AsaDataReader is closed.

IsDBNull method

Description	A value indicating whether the column contains null values.
Syntax	bool **IsDBNull(** int *ordinal* **)**
Parameters	**ordinal** The zero-based column ordinal.
Return value	True if the specified column value is equivalent to DBNull. Otherwise, false.
Implements	IDataRecord.IsDbNull
Usage	Call this method to check for null column values before calling the typed get methods (for example, GetByte, GetChar, and so on) to avoid raising an exception.

Item property

Description	The value of a column in its native format. In C#, this property is the indexer for the AsaDataReader class.
Syntax 1	object **this[** int *index* **]**
Syntax 2	object **this[** string *name* **]**
Parameters	**index** The column ordinal.
	name The column name.
Access	Read-only
Implements	IDataRecord.Item

NextResult method

Description	Advances the AsaDataReader to the next result, when reading the results of batch SQL statements.
Syntax	bool **NextResult( )**
Return value	True if there are more result sets. Otherwise, false.
Implements	IDataReader.NextResult
Usage	Used to process multiple results, which can be generated by executing batch SQL statements.
	By default, the data reader is positioned on the first result.

Read method

Description	Reads the next row of the result set and moves the AsaDataReader to that row.
Syntax	bool **Read( )**
Return value	Returns true if there are more rows. Otherwise, it returns false.
Implements	IDataReader.Read
Usage	The default position of the AsaDataReader is prior to the first record. Therefore, you must call Read to begin accessing any data.
Example	The following code fills a list box with the values in a single column of results.

```
while( reader.Read() )
{
    listResults.Items.Add(
        reader.GetValue( 0 ).ToString() );
}
listResults.EndUpdate();
reader.Close();
```

RecordsAffected property

Description	The number of rows changed, inserted, or deleted by execution of the SQL statement.
Syntax	int **RecordsAffected**
Access	Read-only
Property Value	The number of rows changed, inserted, or deleted. This is 0 if no rows were affected or the statement failed, or -1 for SELECT statements.
Implements	IDataReader.RecordsAffected
Usage	The number of rows changed, inserted, or deleted. The value is 0 if no rows were affected or the statement failed, and -1 for SELECT statements.
	The value of this property is cumulative. For example, if two records are inserted in batch mode, the value of RecordsAffected will be two.
	IsClosed and RecordsAffected are the only properties that you can call after the AsaDataReader is closed.

AsaDbType enum

Specifies Adaptive Server Anywhere data types.

Members

BigInt

Binary

Bit

Char

Date

Decimal

Double

Float

Integer

LongBinary

LongVarchar

Numeric

SmallInt

Time

TimeStamp

TinyInt

UnsignedBigInt

UnsignedInt

UnsignedSmallInt

VarBinary

VarChar

UniqueIdentifier

AsaError class

Description	Collects information relevant to a warning or error returned by the data source.
Base classes	Object
	There is no constructor for AsaError.
See also	"Error handling and the Adaptive Server Anywhere .NET data provider" on page 388

Message property

Description	A short description of the error.
Syntax	string **Message**
Access	Read-only

NativeError property

Description	Database-specific error information.
Syntax	int **NativeError**
Access	Read-only

Source property

Description	The name of the provider that generated the error.
Syntax	string **Source**
Access	Read-only

SqlState property

Description	The Adaptive Server Anywhere five-character SQL state following the ANSI SQL standard. If the error can be issued from more than one place, the five-character error code identifies the source of the error.
Syntax	string **SqlState**
Access	Read-only

ToString method

Description	The complete text of the error message.
Syntax	string **ToString( )**
Usage	The return value is a string is in the form "AsaError:", followed by the Message. For example,

`AsaError:UserId or Password not valid.`

AsaErrorCollection class

Description	Collects all errors generated by the Adaptive Server Anywhere ADO.NET data provider.
Base classes	Object
Implemented interfaces	ICollection, IEnumerable
	There is no constructor for AsaErrorCollection. Typically, an AsaErrorCollection is obtained from the AsaException.Errors property.
See also	"Errors property" on page 437
	"Error handling and the Adaptive Server Anywhere .NET data provider" on page 388

CopyTo method

Description	Copies the elements of the AsaErrorCollection into an array, starting at the given index within the array.
Syntax	void **CopyTo(** Array *array*, int *index* **)**
Parameters	**array** The array into which to copy the elements.
	index The starting index of the array.
Implements	ICollection.CopyTo

Count property

Description	The number of errors in the collection.
Syntax	int **Count**
Access	Read-only
Implements	ICollection.Count

Item property

Description	The error at the specified index.
Syntax	AsaError **this[** int *index* **]**
Parameters	**index** The zero-based index of the error to retrieve.

435

Property Value   An AsaError that contains the error at the specified index.

Access   Read-only

AsaException class

Description	The exception that is thrown when Adaptive Server Anywhere returns a warning or error.
Base classes	SystemException

There is no constructor for AsaException. Typically, an AsaException object is declared in a catch. For example:

```
...
catch( AsaException ex )
{
    MessageBox.Show( ex.Errors[0].Message, "Error" );
}
```

See also	"Error handling and the Adaptive Server Anywhere .NET data provider" on page 388

Errors property

Description	A collection of one or more AsaError objects.
Syntax	AsaErrorCollection **Errors**
Access	Read-only
Usage	The AsaErrorCollection class always contains at least one instance of the AsaError class.

GetObjectData method

Description	This member overrides Exception.GetObjectData.
Syntax	void **GetObjectData(** SerializationInfo *info*, StreamingContext *context* **)**
Parameters	**info** The SerializationInfo that holds the serialized object data about the exception being thrown.
	context The StreamingContext that contains contextual information about the source or destination.

Message property

Description	The text describing the error.
Syntax	string **Message**

Access	Read-only
Usage	This method returns a single string that contains a concatenation of all of the Message properties of all of the AsaError objects in the Errors collection. Each message, except the last one, is followed by a carriage return.

Source property

Description	The name of the provider that generated the error.
Syntax	string **Source**
Access	Read-only

AsaInfoMessageEventArgs class

Description	Provides data for the InfoMessage event.
Base classes	EventArgs

There is no constructor for AsaInfoMessageEventArgs.

Errors property

Description	The collection of warnings sent from the data source.
Syntax	AsaErrorCollection **Errors**
Access	Read-only

Message property

Description	The full text of the error sent from the data source.
Syntax	string **Message**
Access	Read-only

Source property

Description	The name of the object that generated the error.
Syntax	string **Source**
Access	Read-only

ToString method

Description	Retrieves a string representation of the InfoMessage event.
Syntax	string **ToString( )**
Return value	A string representing the InfoMessage event.

AsaInfoMessageEventHandler delegate

Description	Represents the method that will handle the InfoMessage event of an AsaConnection.
Syntax	void **AsaInfoMessageEventHandler (** object *sender*, AsaInfoMessageEventArgs *e* **)**
Parameters	**sender** The source of the event.
	e The AsaInfoMessageEventArgs object that contains the event data.

AsaParameter class

Description	Represents a parameter to an AsaCommand and optionally, its mapping to a DataSet column.
Base classes	MarshalByRefObject
Implemented interfaces	IDbDataParameter, IDataParameter

AsaParameter constructors

Syntax 1

void **AsaParameter( )**

Syntax 2

void **AsaParameter(**
string *parameterName*,
object *value*
)

Syntax 3

void **AsaParameter(**
string *parameterName*,
AsaDbType *dbType*
)

Syntax 4

void **AsaParameter(**
string *parameterName*,
AsaDbType *dbType*,
int *size*
)

Syntax 5

void **AsaParameter(**
string *parameterName*,
AsaDbType *dbType*,
int *size*,
string *sourceColumn*
)

Syntax 6

void **AsaParameter(**
string *parameterName*,
AsaDbType *dbType*,
int *size*,
ParameterDirection *direction*,
bool *isNullable*,
byte *precision*,
byte *scale*,
string *sourceColumn*,
DataRowVersion *sourceVersion*,
object *value*
)

Parameters	**value** An Object that is the value of the parameter.
	size The length of the parameter.
	sourceColumn The name of the source column to map.
	parameterName The name of the parameter.
	dbType One of the AsaDbType values.
	direction One of the ParameterDirection values.
	isNullable True if the value of the field can be null; otherwise, false.
	precision The total number of digits to the left and right of the decimal point to which Value is resolved.
	scale The total number of decimal places to which Value is resolved.
	sourceVersion One of the DataRowVersion values.

AsaDbType property

Description	The AsaDbType of the parameter.
Syntax	AsaDbType **AsaDbType**
Access	Read-write
Usage	The AsaDbType and DbType are linked. Therefore, setting the DbType changes the AsaDbType to a supporting AsaDbType.
	The value must be a member of the AsaDbType enumerator.

DbType property

Description	The DbType of the parameter.
Syntax	DbType **DbType**
Access	Read-write
Usage	The AsaDbType and DbType are linked. Therefore, setting the DbType changes the AsaDbType to a supporting AsaDbType.
	The value must be a member of the AsaDbType enumerator.

Direction property

Description	A value indicating whether the parameter is input-only, output-only, bidirectional, or a stored procedure return value parameter.

Syntax	ParameterDirection **Direction**
Access	Read-write
Usage	If the ParameterDirection is output, and execution of the associated AsaCommand does not return a value, the AsaParameter contains a null value. After the last row from the last result set is read, Output, InputOut, and ReturnValue parameters are updated.

IsNullable property

Description	A value indicating whether the parameter accepts null values.
Syntax	bool **IsNullable**
Access	Read-write
Usage	This property is true if null values are accepted; otherwise it is false. The default is false. Null values are handled using the DBNull class.

Offset property

Description	The offset to the Value property.
Syntax	int **Offset**
Access	Read-write
Property value	The offset to the value. The default is 0.

ParameterName property

Description	The name of the AsaParameter.
Syntax	string **ParameterName**
Access	Read-write
Implements	IDataParameter.ParameterName
Usage	The Adaptive Server Anywhere .NET data provider uses positional parameters that are marked with a question mark (?) instead of named parameters.
	The default is an empty string.

Precision property

Description	The maximum number of digits used to represent the Value property.

Syntax	byte **Precision**
Access	Read-write
Implements	IDbDataParameter.Precision
Usage	The value of this property is the maximum number of digits used to represent the Value property. The default value is 0, which indicates that the data provider sets the precision for the Value property.
	The Precision property is only used for decimal and numeric input parameters.

Scale property

Description	The number of decimal places to which Value is resolved.
Syntax	byte **Scale**
Access	Read-write
Implements	IDbDataParameter.Scale
Usage	The default is 0. The Scale property is only used for decimal and numeric input parameters.

Size property

Description	The maximum size, in bytes, of the data within the column.
Syntax	int **Size**
Access	Read-write
Implements	IDbDataParameter.Size
Usage	The value of this property is the maximum size, in bytes, of the data within the column. The default value is inferred from the parameter value.
	The Size property is used for binary and string types.
	For variable length data types, the Size property describes the maximum amount of data to transmit to the server. For example, the Size property can be used to limit the amount of data sent to the server for a string value to the first one hundred bytes.
	If not explicitly set, the size is inferred from the actual size of the specified parameter value. For fixed width data types, the value of Size is ignored. It can be retrieved for informational purposes, and returns the maximum

amount of bytes the provider uses when transmitting the value of the parameter to the server.

SourceColumn property

Description	The name of the source column mapped to the DataSet and used for loading or returning the value.
Syntax	string **SourceColumn**
Access	Read-write
Implements	IDbDataParameter.SourceColumn
Usage	When SourceColumn is set to anything other than an empty string, the value of the parameter is retrieved from the column with the SourceColumn name. If Direction is set to Input, the value is taken from the DataSet. If Direction is set to Output, the value is taken from the data source. A Direction of InputOutput is a combination of both.

SourceVersion property

Description	The DataRowVersion to use when loading Value.
Syntax	DataRowVersion **SourceVersion**
Access	Read-write
Implements	IDbDataParameter.SourceVersion
Usage	Used by UpdateCommand during an Update operation to determine whether the parameter value is set to Current or Original. This allows primary keys to be updated. This property is ignored by InsertCommand and DeleteCommand. This property is set to the version of the DataRow used by the Item property, or the GetChildRows method of the DataRow object.

ToString method

Description	A string containing the ParameterName.
Syntax	string **ToString( )**
Access	Read-write

Value property

Description	The value of the parameter.

Syntax	object **Value**
Access	Read-write
Implements	IDataParameter.Value
Usage	For input parameters, the value is bound to the AsaCommand that is sent to the server. For output and return value parameters, the value is set on completion of the AsaCommand and after the AsaDataReader is closed.
	When sending a null parameter value to the server, the user must specify DBNull, not null. The null value in the system is an empty object that has no value. DBNull is used to represent null values.
	If the application specifies the database type, the bound value is converted to that type when the provider sends the data to the server. The provider attempts to convert any type of value if it supports the IConvertible interface. Conversion errors may result if the specified type is not compatible with the value.
	Both the DbType and AsaDbType properties can be inferred by setting the Value.
	The Value property is overwritten by Update.

AsaParameterCollection class

Description	Represents all parameters to an AsaCommand and optionally, their mapping to a DataSet column.
Base classes	Object
Implemented interfaces	ICollection, IEnumerable, IDataParameterCollection
Usage	There is no constructor for AsaParameterCollection. You obtain an AsaParameterCollection from the AsaCommand.Parameters property.
See also	"Parameters property" on page 397

Add method

Description	Adds an AsaParameter to the AsaCommand.
Syntax 1	int **Add(** object *value* **)**
Syntax 2	int **Add(** AsaParameter *value* **)**
Syntax 3	int **Add(** string *parameterName*, object *value* **)**
Syntax 4	int **Add(** string *parameterName*, AsaDbType *asaDbType* **)**
Syntax 5	int **Add(** string *parameterName*, AsaDbType *asaDbType*, int *size* **)**
Syntax 6	int **Add(** string *parameterName*, AsaDbType *asaDbType*, int *size*, string *sourceColumn* **)**
Parameters	**value** For syntax 1 and 2, value is the AsaParameter object to add to the collection. For syntax 3, value is the value of the parameter to add to the connection.
	parameterName The name of the parameter.

	asaDbType One of the AsaDbType values.
	size The length of the column.
	sourceColumn The name of the source column.
Return Value	The index of the new AsaParameter object.

Clear method

Description	Removes all items from the collection.
Syntax	void **Clear( )**
Implements	IList.Clear

Contains method

Description	A value indicating whether an AsaParameter exists in the collection.
Syntax 1	bool **Contains(** object *value* **)**
Syntax 2	bool **Contains(** string *value* **)**
Parameters	**value** The value of the AsaParameter object to find. In syntax 2, this is the name.
Return value	True if the collection contains the AsaParameter. Otherwise, it is false.
Implements	Syntax 1 implements IList.Contains
	Syntax 2 implements IDataParameterCollection.Contains

CopyTo method

Description	Copies AsaParameter objects from the AsaParameterCollection to the specified array.
Syntax	void **CopyTo(** array *array* int *index* **)**
Parameters	**array** The array into which to copy the AsaParameter objects.
	index The starting index of the array.
Implements	ICollection.CopyTo

Count property

Description	The number of AsaParameter objects in the collection.
Syntax	int **Count**
Access	Read-only
Implements	ICollection.Count

IndexOf method

Description	The location of the AsaParameter in the collection.
Syntax 1	int **IndexOf(** object *value* **)**
Syntax 2	int **IndexOf(** string *parameterName* **)**
Parameters	**value** The AsaParameter object to locate.
	parameterName The name of the AsaParameter object to locate.
Return Value	The zero-based location of the AsaParameter in the collection.
Implements	Syntax 1 implements IList.IndexOf
	Syntax 2 implements IDataParameterCollection.IndexOf

Insert method

Description	Inserts an AsaParameter in the collection at the specified index.
Syntax	void **Insert(** int *index* object *value* **)**
Parameters	**index** The zero-based index where the parameter is to be inserted within the collection.
	value The AsaParameter to add to the collection.
Implements	IList.Insert

Item property

Description	The AsaParameter at the specified index or name.
Syntax 1	AsaParameter **this[** int *index* **]**
Syntax 2	AsaParameter **this[** string *parameterName* **]**

Parameters	**index** The zero-based index of the parameter to retrieve.
	parameterName The name of the parameter to retrieve.
Property value	An AsaParameter.
Access	Read-write
Usage	In C#, this property is the indexer for the AsaParameterCollection class.

Remove method

Description	Removes the specified AsaParameter from the collection.
Syntax	void **Remove(** object *value* **)**
Parameters	**value** The AsaParameter object to remove from the collection.
Implements	IList.Remove

RemoveAt method

Description	Removes the specified AsaParameter from the collection.
Syntax 1	void **RemoveAt(** int *index* **)**
Syntax 2	void **RemoveAt(** string *parameterName* **)**
Parameters	**index** The zero-based index of the parameter to remove.
	parameterName The name of the AsaParameter object to remove.
Implements	Syntax 1 implements IList.RemoveAt
	Syntax 2 implements IDataParameterCollection.RemoveAt

AsaPermission class

Description
Enables the Adaptive Server Anywhere .NET data provider to ensure that a user has a security level adequate to access an Adaptive Server Anywhere data source.

Base classes
DBDataPermission

AsaPermission constructors

Description
Initializes a new instance of the AsaPermission class.

Syntax 1
void **AsaPermission( )**

Syntax 2
void **AsaPermission(** PermissionState *state* **)**

Syntax 3
void **AsaPermission(**
PermissionState *state*,
bool *allowBlankPassword*
)

Parameters
state   One of the PermissionState values.

allowBlankPassword   Indicates whether a blank password is allowed.

AsaPermissionAttribute class

Description   Associates a security action with a custom security attribute.

Base classes   DBDataPermissionAttribute

AsaPermissionAttribute constructor

Description   Initializes a new instance of the AsaPermissionAttribute class.

Syntax   void **AsaPermissionAttribute(** SecurityAction *action* **)**

Parameters   **action**   One of the SecurityAction values representing an action that can be
                     performed using declarative security.

Return Value   An AsaPermissionAttribute object.

CreatePermission method

Description   Returns an AsaPermission object that is configured according to the attribute
                     properties.

Syntax   IPermission **CreatePermission( )**

AsaRowUpdatedEventArgs class

Description	Provides data for the RowUpdated event.
Base classes	RowUpdatedEventArgs

AsaRowUpdatedEventArgs constructors

Description	Initializes a new instance of the AsaRowUpdatedEventArgs class.
Syntax	void **AsaRowUpdatedEventArgs(** DataRow *dataRow*, IDbCommand *command*, StatementType *statementType*, DataTableMapping *tableMapping* **)**
Parameters	**dataRow** The DataRow sent through an Update.
	command The IDbCommand executed when Update is called.
	statementType One of the StatementType values that specifies the type of query executed.
	tableMapping The DataTableMapping sent through an Update.

Command property

Description	The AsaCommand executed when Update is called.
Syntax	AsaCommand **Command**
Access	Read-only

Errors property

Description	Any errors generated by Adaptive Server Anywhere when the Command was executed. Inherited from RowUpdatedEventArgs.
Syntax	Exception **Errors**
Property value	The errors generated by Adaptive Server Anywhere when the Command was executed.
Access	Read-write

RecordsAffected property

Description	The number of rows changed, inserted, or deleted by execution of the SQL

statement. Inherited from RowUpdatedEventArgs.

Syntax	int **RecordsAffected**
Property value	The number of rows changed, inserted, or deleted; 0 if no rows were affected or the statement failed; and -1 for SELECT statements.
Access	Read-only

Row property

Description	The DataRow sent through an Update. Inherited from RowUpdatedEventArgs.
Syntax	DataRow **Row**
Access	Read-only

StatementType property

Description	The type of the SQL statement that was executed. Inherited from RowUpdatedEventArgs.
Syntax	StatementType **StatementType**
Access	Read-only
Usage	StatementType can be one of **Select**, **Insert**, **Update**, or **Delete**.

Status property

Description	The UpdateStatus of the Command property. Inherited from RowUpdatedEventArgs.
Syntax	UpdateStatus **Status**
Property Value	One of the UpdateStatus values: **Continue**, **ErrorsOccurred**, **SkipAllRemainingRows**, **or SkipCurrentRow**. The default is **Continue**.
Access	Read-write

TableMapping property

Description	The DataTableMapping sent through an Update. Inherited from RowUpdatedEventArgs.
Syntax	DataTableMapping **TableMapping**
Access	Read-only

AsaRowUpdatingEventArgs class

Description	Provides data for the RowUpdating event.
Base classes	RowUpdatingEventArgs

AsaRowUpdatingEventArgs constructors

Description	Initializes a new instance of the AsaRowUpdatingEventArgs class.
Syntax	void **AsaRowUpdatingEventArgs(** DataRow *row*, IDbCommand *command*, StatementType *statementType*, DataTableMapping *tableMapping* **)**
Parameters	**row** The DataRow to update.
	command The IDbCommand to execute during update.
	statementType One of the StatementType values that specifies the type of query executed.
	tableMapping The DataTableMapping sent through an Update.

Command property

Description	The AsaCommand to execute when performing the Update.
Syntax	AsaCommand **Command**
Access	Read-write

Errors property

Description	Any errors generated by Adaptive Server Anywhere when the Command was executed. Inherited from RowUpdatingEventArgs.
Syntax	Exception **Errors**
Property value	The errors generated by Adaptive Server Anywhere when the Command was executed.
Access	Read-write

Row property

Description	The DataRow sent through an Update. Inherited from

	RowUpdatingEventArgs.
Syntax	DataRow **Row**
Access	Read-only

StatementType property

Description	The type of the SQL statement that was executed. Inherited from RowUpdatingEventArgs.
Syntax	StatementType **StatementType**
Access	Read-only
Usage	StatementType can be one of **Select**, **Insert**, **Update**, or **Delete**.

Status property

Description	The UpdateStatus of the Command property. Inherited from RowUpdatingEventArgs.
Syntax	UpdateStatus **Status**
Property Value	One of the UpdateStatus values: **Continue**, **ErrorsOccurred**, **SkipAllRemainingRows**, **or SkipCurrentRow**. The default is **Continue**.
Access	Read-write

TableMapping property

Description	The DataTableMapping sent through an Update. Inherited from RowUpdatingEventArgs.
Syntax	DataTableMapping **TableMapping**
Access	Read-only

AsaRowUpdatedEventHandler delegate

Description	Represents the method that will handle the RowUpdated event of an AsaDataAdapter.
Syntax	void **AsaRowUpdatedEventHandler (** object *sender*, AsaRowUpdatedEventArgs *e* **)**
Parameters	**sender** The source of the event.
	e The AsaRowUpdatedEventArgs that contains the event data.

AsaRowUpdatingEventHandler delegate

Description   Represents the method that will handle the RowUpdating event of an
                    AsaDataAdapter.

Syntax   void **AsaRowUpdatingEventHandler (**
                    object *sender*,
                    AsaRowUpdatingEventArgs *e*
)

Parameters   **sender**   The source of the event.

                    e   The AsaRowUpdatingEventArgs that contains the event data.

AsaTransaction class

Description	Represents a SQL transaction.
Base classes	Object
Implemented interfaces	IDbTransaction
Usage	There is no constructor for AsaTransaction. To obtain an AsaTransaction object, use the AsaConnection.BeginTransaction() method.
	To associate a command with a transaction, use the AsaCommand.Transaction property.
See also	"BeginTransaction method" on page 403
	"Transaction property" on page 398
	"Transaction processing" on page 386
	"Inserting, updating, and deleting rows using the AsaCommand object" on page 366

Commit method

Description	Commits the database transaction.
Syntax	void **Commit( )**
Implements	IDbTransaction.Commit

Connection property

Description	The AsaConnection object associated with the transaction, or a null reference (Nothing in Visual Basic) if the transaction is no longer valid.
Syntax	AsaConnection **Connection**
Access	Read-only
Usage	A single application may have multiple database connections, each with zero or more transactions. This property enables you to determine the connection object associated with a particular transaction created by BeginTransaction.

IsolationLevel property

Description	Specifies the isolation level for this transaction.
Syntax	IsolationLevel *IsolationLevel*

Access	Read-only
Property Value	The IsolationLevel for this transaction. This can be one of **ReadCommitted**, **ReadUncommitted**, **RepeatableRead**, or **Serializable.** The default is **ReadCommitted**.
Implements	IDbTransaction.IsolationLevel
Usage	Parallel transactions are not supported. Therefore, the IsolationLevel applies to the entire transaction.

Rollback method

Description	Rolls back a transaction from a pending state.
Syntax 1	void **Rollback( )**
Syntax 2	void **Rollback(** string *savePoint* **)**
Parameters	**savePoint** The name of the savepoint to which to roll back.
Usage	The transaction can only be rolled back from a pending state (after BeginTransaction has been called, but before Commit is called).

Save method

Description	Creates a savepoint in the transaction that can be used to roll back a portion of the transaction, and specifies the savepoint name.
Syntax	void **Save(** string *savePoint* **)**
Parameters	**savePoint** The name of the savepoint to which to roll back.

CHAPTER 14

The Open Client Interface

About this chapter   This chapter describes the Open Client programming interface for Adaptive
Server Anywhere.

The primary documentation for Open Client application development is the
Open Client documentation, available from Sybase. This chapter describes
features specific to Adaptive Server Anywhere, but it is not an exhaustive
guide to Open Client application programming.

Contents

What you need to build Open Client applications

To run Open Client applications, you must install and configure Open Client components on the machine where the application is running. You may have these components present as part of your installation of other Sybase products or you can optionally install these libraries with Adaptive Server Anywhere, subject to the terms of your license agreement.

Open Client applications do not need any Open Client components on the machine where the database server is running.

To build Open Client applications, you need the development version of Open Client, available from Sybase.

By default, Adaptive Server Anywhere databases are created as case-insensitive, while Adaptive Server Enterprise databases are case sensitive.

☞ For more information on running Open Client applications with Adaptive Server Anywhere, see "Adaptive Server Anywhere as an Open Server" [*ASA Database Administration Guide,* page 101].

Data type mappings

Open Client has its own internal data types, which differ in some details from those available in Adaptive Server Anywhere. For this reason, Adaptive Server Anywhere internally maps some data types between those used by Open Client applications and those available in Adaptive Server Anywhere.

To build Open Client applications, you need the development version of Open Client. To use Open Client applications, the Open Client runtimes must be installed and configured on the computer where the application runs.

The Adaptive Server Anywhere server does not require any external communications runtime in order to support Open Client applications.

Each Open Client data type is mapped onto the equivalent Adaptive Server Anywhere data type. All Open Client data types are supported

Adaptive Server Anywhere data types with no direct counterpart in Open Client

The following table lists the mappings of data types supported in Adaptive Server Anywhere that have no direct counterpart in Open Client.

ASA data type	Open Client data type
unsigned short	int
unsigned int	bigint
unsigned bigint	bigint
date	smalldatetime
time	smalldatetime
serialization	longbinary
string	varchar
timestamp struct	datetime

Range limitations in data type mapping

Some data types have different ranges in Adaptive Server Anywhere than in Open Client. In such cases, overflow errors can occur during retrieval or insertion of data.

The following table lists Open Client application data types that can be mapped to Adaptive Server Anywhere data types, but with some restriction in the range of possible values.

In most cases, the Open Client data type is mapped to an Adaptive Server

Anywhere data type that has a greater range of possible values. As a result, it is possible to pass a value to Adaptive Server Anywhere that will be accepted and stored in a database, but one that is too large to be fetched by an Open Client application.

Data type	Open Client lower range	Open Client upper range	ASA lower range	ASA upper range
MONEY	–922 377 203 685 477.5808	922 377 203 685 477.5807	–1e15 + 0.0001	1e15 – 0.0001
SMALLMONEY	–214 748.3648	214 748.3647	–214 748.3648	214 748.3647
DATETIME	Jan 1, 1753	Dec 31, 9999	Jan 1, 0001	Dec 31, 9999
SMALLDATETIME	Jan 1, 1900	June 6, 2079	March 1, 1600	Dec 31, 7910

Example

For example, the Open Client MONEY and SMALLMONEY data types do not span the entire numeric range of their underlying Adaptive Server Anywhere implementations. Therefore, it is possible to have a value in an Adaptive Server Anywhere column which exceeds the boundaries of the Open Client data type MONEY. When the client fetches any such offending values via Adaptive Server Anywhere, an error is generated.

Timestamps

The Adaptive Server Anywhere implementation of the Open Client TIMESTAMP data type, when such a value is passed in Adaptive Server Anywhere, is different from that of Adaptive Server Enterprise. In Adaptive Server Anywhere, the value is mapped to the Adaptive Server Anywhere DATETIME data type. The default value is NULL in Adaptive Server Anywhere and no guarantee is made of its uniqueness. By contrast, Adaptive Server Enterprise ensures that the value is monotonically increasing in value, and so, is unique.

By contrast, the Adaptive Server Anywhere TIMESTAMP data type contains year, month, day, hour, minute, second, and fraction of second information. In addition, the DATETIME data type has a greater range of possible values than the Open Client data types that are mapped to it by Adaptive Server Anywhere.

Using SQL in Open Client applications

This section provides a very brief introduction to using SQL in Open Client applications, with a particular focus on Adaptive Server Anywhere-specific issues.

☞ For an introduction to the concepts, see "Using SQL in Applications" on page 11. For a complete description, see your Open Client documentation.

Executing SQL statements

You send SQL statements to a database by including them in Client Library function calls. For example, the following pair of calls executes a DELETE statement:

```
ret = ct_command(cmd, CS_LANG_CMD,
                    "DELETE FROM employee
                     WHERE emp_id=105"
                  CS_NULLTERM,
                  CS_UNUSED);
ret = ct_send(cmd);
```

The `ct_command` function is used for a wide range of purposes.

Using prepared statements

The `ct_dynamic` function is used to manage prepared statements. This function takes a *type* parameter which describes the action you are taking.

❖ **To use a prepared statement in Open Client**

1. Prepare the statement using the `ct_dynamic` function, with a CS_PREPARE *type* parameter.

2. Set statement parameters using `ct_param`.

3. Execute the statement using `ct_dynamic` with a CS_EXECUTE *type* parameter.

4. Free the resources associated with the statement using `ct_dynamic` with a CS_DEALLOC *type* parameter.

☞ For more information on using prepared statements in Open Client, see your Open Client documentation

Using cursors

The `ct_cursor` function is used to manage cursors. This function takes a *type* parameter which describes the action you are taking.

Supported cursor types
Not all the types of cursor that Adaptive Server Anywhere supports are available through the Open Client interface. You cannot use scroll cursors, dynamic scroll cursors, or insensitive cursors through Open Client.

Uniqueness and updatability are two properties of cursors. Cursors can be unique (each row carries primary key or uniqueness information, regardless of whether it is used by the application) or not. Cursors can be read only or updatable. If a cursor is updatable and not unique, performance may suffer, as no prefetching of rows is done in this case, regardless of the CS_CURSOR_ROWS setting (see below).

The steps in using cursors
In contrast to some other interfaces, such as Embedded SQL, Open Client associates a cursor with a SQL statement expressed as a string. Embedded SQL first prepares a statement and then the cursor is declared using the statement handle.

❖ **To use cursors in Open Client**

1. To declare a cursor in Open Client, you use `ct_cursor` with CS_CURSOR_DECLARE as the *type* parameter.

2. After declaring a cursor, you can control how many rows are prefetched to the client side each time a row is fetched from the server using `ct_cursor` with CS_CURSOR_ROWS as the *type* parameter.

   Storing prefetched rows at the client side cuts down the number of calls to the server and this improves overall throughput as well as turnaround time. Prefetched rows are not immediately passed on to the application; they are stored in a buffer at the client side ready for use.

   The setting of the PREFETCH database option controls prefetching of rows for other interfaces. It is ignored by Open Client connections. The CS_CURSOR_ROWS setting is ignored for non-unique, updatable cursors.

3. To open a cursor in Open Client, you use `ct_cursor` with CS_CURSOR_OPEN as the *type* parameter.

4. To fetch each row in to the application, you use `ct_fetch`.

5. To close a cursor, you use ct_cursor with CS_CURSOR_CLOSE.

6. In Open Client, you also need to deallocate the resources associated with a cursor. You do this using ct_cursor with CS_CURSOR_DEALLOC. You can also use CS_CURSOR_CLOSE with the additional parameter CS_DEALLOC to carry out these operations in a single step.

Modifying rows through a cursor

With Open Client, you can delete or update rows in a cursor, as long as the cursor is for a single table. The user must have permissions to update the table and the cursor must be marked for update.

❖ **To modify rows through a cursor**

1. Instead of carrying out a fetch, you can delete or update the current row of the cursor using ct_cursor with CS_CURSOR_DELETE or CS_CURSOR_UPDATE, respectively.

You cannot insert rows through a cursor in Open Client applications.

Describing query results in Open Client

Open Client handles result sets in a different way than some other Adaptive Server Anywhere interfaces.

In Embedded SQL and ODBC, you **describe** a query or stored procedure in order to set up the proper number and types of variables to receive the results. The description is done on the statement itself.

In Open Client, you do not need to describe a statement. Instead, each row returned from the server can carry a description of its contents. If you use `ct_command` and `ct_send` to execute statements, you can use the `ct_results` function to handle all aspects of rows returned in queries.

If you do not wish to use this row-by-row method of handling result sets, you can use `ct_dynamic` to prepare a SQL statement and use `ct_describe` to describe its result set. This corresponds more closely to the describing of SQL statements in other interfaces.

Known Open Client limitations of Adaptive Server Anywhere

Using the Open Client interface, you can use an Adaptive Server Anywhere database in much the same way as you would an Adaptive Server Enterprise database. There are some limitations, including the following:

♦ **Commit Service** Adaptive Server Anywhere does not support the Adaptive Server Enterprise Commit Service.

♦ **Capabilities** A client/server connection's **capabilities** determine the types of client requests and server responses permitted for that connection. The following capabilities are not supported:

- CS_REG_NOTIF
- CS_CSR_ABS
- CS_CSR_FIRST
- CS_CSR_LAST
- CS_CSR_PREV
- CS_CSR_REL
- CS_DATA_BOUNDARY
- CS_DATA_SENSITIVITY
- CS_PROTO_DYNPROC
- CS_REQ_BCP

♦ Security options, such as SSL and encrypted passwords, are not supported.

♦ Open Client applications may connect to Adaptive Server Anywhere using TCP/IP or using local machine NamedPipes protocol where available.

☞ For more information on capabilities, see the *Open Server Server-Library C Reference Manual*.

CHAPTER 15

The DBD::ASAny Perl Interface

About this chapter   This chapter describes how to install and use the DBD::ASAny interface.
This interface allows scripts written in Perl to access and manipulate
information stored in Adaptive Server Anywhere databases.

Contents

About DBD::ASAny

The DBD::ASAny interface provides access to Adaptive Server Anywhere databases from scripts written in Perl. ASAny is a driver for the Database Independent Interface for Perl (DBI) module written by Tim Bunce. Once you have installed the DBI module and DBD::ASAny, you can access and change the information in Adaptive Server Anywhere databases from Perl.

The DBD::ASAny driver is thread-safe when using Perl with ithreads.

Requirements   The DBD::ASAny interface requires the following components.

♦ Perl 5.6.0 or newer. On Windows, ActivePerl 5.6.0 build 616 or later is required.

♦ DBI 1.34 or newer

♦ A C compiler. On Windows, only the Microsoft Visual C compiler is supported.

Installing DBD::ASAny on Windows

The following procedure documents how to install the DBD::ASAny interface on Windows.

❖ **To prepare your machine**

1. Install ActivePerl 5.6.0 or later. You can use the ActivePerl installer to install Perl and configure your machine. You do not need to recompile Perl.

2. Install Microsoft Visual C or Microsoft Visual Studio .NET and configure your environment.

   If you did not choose to configure your environment at install time, you must set your PATH, LIB, and INCLUDE environment variables correctly before proceeding. Microsoft provides a batch file for this purpose. For example, a batch file called *vsvars32.bat* is included in the *Common7\Tools* subdirectory of the Visual Studio .NET 2003 installation. Open a new system command prompt and run this batch file before continuing.

❖ **To install the DBI Perl module on Windows**

1. At a command prompt, change to the *bin* subdirectory of your ActivePerl installation directory.

   The system command prompt is strongly recommended as the following steps may not work from alternative shells.

2. Start the Perl Module Manager by executing the following command.

   ```
   ppm
   ```

   If ppm fails to start, check that Perl is installed correctly.

3. Enter the following command at the ppm prompt.

   ```
   query dbi
   ```

   This command should generate two lines of text similar to those shown below. In this case, the information indicates that ActivePerl version 5.8.1 build 807 is running and that DBI version 1.38 is installed.

   ```
   Querying target 1 (ActivePerl 5.8.1.807)
     1. DBI [1.38] Database independent interface for Perl
   ```

   If DBI is not installed, you must install it. To do so, enter the following command at the ppm prompt.

   ```
   install dbi
   ```

❖ To install DBD::ASAny on Windows

1. At a system command prompt, change to the *src\perl* subdirectory of your SQL Anywhere Studio installation.

2. Enter the following commands to build and test ASAny.

   ```
   perl Makefile.PL
   ```

   ```
   nmake
   ```

   If for any reason your need to start over, you can run the command **make clean** to remove any partially built targets.

3. To test DBD::ASAny, copy the Adaptive Anywhere Sample Database file to your DBD::ASAny directory and make the tests.

   ```
   copy "C:\Program Files\Sybase\SQL Anywhere 9\asademo.db" .
   ```

   ```
   dbeng9 asademo
   ```

   ```
   nmake test
   ```

   If the tests do not run, ensure that the *win32* subdirectory of the SQL Anywhere installation is in your path.

4. To complete the installation, execute the following command at the same prompt.

   ```
   nmake install
   ```

The DBD::ASAny interface is now ready to use.

Installing DBD::ASAny on UNIX

The following procedure documents how to install the DBD::ASAny interface on the supported UNIX platforms, including Mac OS X.

❖ **To prepare your machine**

1. Install 5.6.0 build 616 or later.

2. Install a C compiler.

❖ **To install the DBI Perl module on UNIX**

1. Download the DBI module source from www.cpan.org. Extract the contents of this file into a new directory.

2. At a command prompt, change to the new directory and execute the following commands to build the DBI module.

   ```
   perl Makefile.PL
   ```

   ```
   make
   ```

   If for any reason your need to start over, you can use the command **make clean** to remove any partially built targets.

3. Use the following command to test the DBI module.

   ```
   make test
   ```

4. To complete the installation, execute the following command at the same prompt.

   ```
   make install
   ```

5. Optionally, you can now delete the DBI source tree. It is no longer required.

❖ **To install DBD::ASAny on UNIX**

1. Make sure the environment is set up for Adaptive Server Anywhere.

   Depending on which shell you're using, enter the appropriate command to source the Adaptive Server Anywhere configuration script from the SQL Anywhere Studio installation directory:

In this shell . . .	. . . use this command
sh, ksh, or bash	`.   /bin/asa_config.sh`
csh or tcsh	`source /bin/asa_config.csh`

2. At a shell prompt, change to the *src/perl* subdirectory of your SQL Anywhere Studio installation.

3. At a system command prompt, enter the following commands to build ASAny.

```
perl Makefile.PL
```

```
make
```

If for any reason your need to start over, you can use the command **make clean** to remove any partially built targets.

4. To test DBD::ASAny, copy the Adaptive Anywhere Sample Database file to your DBD::ASAny directory and make the tests.

```
cp /opt/sybase/SYBSsa9/asademo.db .
```

```
dbeng9 asademo
```

```
make test
```

If the tests do not run, ensure that the *win32* subdirectory of the SQL Anywhere installation is in your path.

5. To complete the installation, execute the following command at the same prompt.

```
make install
```

The DBD::ASAny interface is now ready to use.

Writing Perl scripts that use DBD::ASAny

This section provides an overview of how to write Perl scripts that use the DBD::ASAny interface. DBD::ASAny is a driver for the DBI module. Complete documentation for the DBI module is available online at dbi.perl.org.

Loading the DBI module

To use the DBD::ASAny interface from a Perl script, you must first tell Perl that you plan to use the DBI module. To do so, include the following line at the top of the file.

```
use DBI;
```

In addition, it is highly recommended that you run Perl in strict mode. This statement, which for example makes explicit variable definitions mandatory, is likely to greatly reduce the chance that you will run into mysterious errors due to such common mistakes as typographical errors.

```
#!/usr/local/bin/perl -w
#
use DBI;
use strict;
```

The DBI module automatically loads the DBD drivers, of which ASAny is one, as required.

Opening and closing a connection

Generally, you open a single connection to a database and then perform all the required operations through it by executing a sequence of SQL statements. To open a connection, you use the connect method. The return value is a handle to the database connection that you use to perform subsequent operations on that connection.

The parameters to the connect method are as follows:

1. ”DBI:ASAny:” and additional connection parameters separated by semicolons.

2. A user name. Unless this string is blank, “;UID=*value*” is appended to the connection string.

3. A password value. Unless this string is blank, “;PWD=*value*” is appended to the connection string.

4. A pointer to a hash of default values. Settings such as AutoCommit,

RaiseError, and PrintError may be set in this manner.

The following code sample opens and closes a connection to the Adaptive Server Anywhere Sample database. You must start the database server and Sample database before running this script.

```
#!/usr/local/bin/perl -w
#
use DBI;
use strict;
my $database = "asademo";
my $data_src = "DBI:ASAny:ENG=$database;DBN=$database";
my $uid      = "dba";
my $pwd      = "sql";
my %defaults = (
     AutoCommit => 1, # Autocommit enabled.
     PrintError => 0  # Errors not automatically printed.
   );
my $dbh = DBI->connect($data_src, $uid, $pwd, \%defaults)
  or die "Can't connect to $data_source: $DBI::errstr\n";
$dbh->disconnect;
exit(0);
__END__
```

Optionally, you can append the user name or password value to the data-source string instead of supplying them as separate parameters. If you do so, supply a blank string for the corresponding argument. For example, in the above script may be altered by replacing the statement that opens the connections with these statements:

```
$data_src .= ";UID=$uid";
$data_src .= ";PWD=$pwd";
my $dbh = DBI->connect($data_src, '', '', \%defaults)
  or die "Can't connect to $data_source: $DBI::errstr\n";
```

Selecting data

Once you have obtained a handle to an open connection, you can access and modify data stored in the database. Perhaps the simplest operation is to retrieve some rows and print them out.

SQL statements that return row sets must be prepared before being executed. The prepare method returns a handle to the statement. You use the handle to execute the statement, then retrieve meta information about the result set and the rows of the result set.

```
#!/usr/local/bin/perl -w
#
use DBI;
use strict;
my $database = "asademo";
my $data_src = "DBI:ASAny:ENG=$database;DBN=$database";
my $uid      = "dba";
my $pwd      = "sql";
my $sel_stmt = "SELECT id, fname, lname
                FROM customer
                ORDER BY fname, lname";
my %defaults = (
    AutoCommit => 0, # Require explicit commit or rollback.
    PrintError => 0
  );
my $dbh = DBI->connect($data_src, $uid, $pwd, \%defaults)
  or die "Can't connect to $data_src: $DBI::errstr\n";
&db_query($sel_stmt, $dbh);
$dbh->rollback;
$dbh->disconnect;
exit(0);

sub db_query {
    my($sel, $dbh) = @_;
    my($row, $sth) = undef;
    $sth = $dbh->prepare($sel);
    $sth->execute;
    print "Fields:    $sth->{NUM_OF_FIELDS}\n";
    print "Params:    $sth->{NUM_OF_PARAMS}\n\n";
    print join("\t\t", @{$sth->{NAME}}), "\n\n";
    while($row = $sth->fetchrow_arrayref) {
        print join("\t\t", @$row), "\n";
    }
    $sth = undef;
}
__END__
```

Prepared statements are not dropped from the database server until the Perl
statement handle is destroyed. To destroy a statement handle, reuse the
variable or set it to undef. Calling the finish method does not drop the
handle. In fact, the finish method should not be called, except when you
have decided not to finish reading a result set.

To detect handle leaks, the Adaptive Server Anywhere database server limits
the number of cursors and prepared statements permitted to a maximum of
50 per connection by default. The resource governor automatically generates
an error if these limits are exceeded. If you get this error, check for
undestroyed statement handles. Use prepare_cached sparingly, as the
statement handles are not destroyed.

If necessary, you can alter these limits by setting the

MAX_CURSOR_COUNT and MAX_STATEMENT_COUNT options.

☞ For more information, see "MAX_CURSOR_COUNT option [database]" [*ASA Database Administration Guide,* page 668] and "MAX_STATEMENT_COUNT option [database]" [*ASA Database Administration Guide,* page 670]

Inserting rows

Inserting rows requires a handle to an open connection. The simplest method is to use a parameterized select statement, meaning that question marks are used as place holders for values. The statement is first prepared, then executed once per new row. The new row values are supplied as parameters to the execute method.

The following sample program inserts two new customers. Although the row values appear as literal strings, you may want to read the values from a file.

```perl
#!/usr/local/bin/perl -w
#
use DBI;
use strict;
my $database = "asademo";
my $data_src = "DBI:ASAny:ENG=$database;DBN=$database";
my $uid      = "dba";
my $pwd      = "sql";
my $ins_stmt = "INSERT INTO customer (id, fname, lname,
                         address, city, state, zip,
                         phone, company_name)
                VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?)";
my %defaults = (
    AutoCommit => 0, # Require explicit commit or rollback.
    PrintError => 0
   );
my $dbh = DBI->connect($data_src, $uid, $pwd, \%defaults)
    or die "Can't connect to $data_src: $DBI::errstr\n";
&db_insert($ins_stmt, $dbh);
$dbh->commit;
$dbh->disconnect;
exit(0);

sub db_insert {
    my($ins, $dbh) = @_;
    my($sth) = undef;
    my @rows = (
      "801,Alex,Alt,5 Blue Ave,New York,NY,78312,43234,BXM",
      "802,Zach,Zed,82 Fair St,New York,NY,12333,72234,Zap"
       );
    $sth = $dbh->prepare($ins);
    my $row = undef;
    foreach $row ( @rows ) {
      my @values = split(/,/, $row);
      $sth->execute(@values);
    }
}
__END__
```

Using the SQLAnywhere PHP module

About this chapter

This chapter describes how to install and use the SQLAnywhere PHP module, which provides access to SQL Anywhere databases from PHP scripts.

Contents

About the SQLAnywhere PHP module

PHP, which stands for Hypertext Preprocessor, is an Open Source scripting language. Although it can be used as a general-purpose scripting language, it was designed to be a convenient language in which to write scripts that could be embedded with HTML documents. Unlike scripts written in JavaScript, which are frequently executed by the client, PHP scripts are processed by the Web Server, and the resulting HTML output sent to the clients. The syntax of PHP is derived from that of other popular languages, such as Java and Perl.

To make it a convenient language in which to develop dynamic web pages, PHP provides the ability to retrieve information from many popular databases, such as Adaptive Server Anywhere. Included with SQL Anywhere, are two modules that provide access to Adaptive Server Anywhere databases from PHP. You can use these modules and the PHP language to write stand-alone scripts and create dynamic web pages that rely on information stored in Adaptive Server Anywhere databases.

The PHP functionality is provided by a DLL on Windows versions of SQL Anywhere, and by a shared object on all versions of UNIX, Linux, and Mac OS X.

Requirements   The following is a list of the required software.

♦ Adaptive Server Anywhere 9.0.2 or higher.

♦ The PHP binaries for your platform, which can be downloaded from *http://www.php.net*.

♦ A web server, if you want to run PHP scripts within a web server. Adaptive Server Anywhere can be run on the same machine as the web server, or on a different machine.

♦ The Adaptive Server Anywhere client software *dblib9.dll* (Windows), *libdblib9.so* (Linux and UNIX), or *libdblib9.dylib.1* (Mac OS X)

Installing and configuring SQLAnywhere PHP

The following sections describe how to install and configure the SQLAnywhere PHP module.

Choosing which PHP module to use

On Windows, SQL Anywhere includes a module for PHP version 4 and one for PHP version 5.

On UNIX, SQL Anywhere includes both a threaded and a non-threaded module for PHP version 4 and for PHP version 5. On UNIX, PHP is supplied in a CGI version that can be used on the command line, and also as part of the Apache web server. If you are using the CGI version of PHP or if you are using Apache 1.x, use the non-threaded module. If you are using Apache 2.x, use the threaded module.

The filenames are as follows:

Filename	Description
php4_sqlanywhere9.dll	PHP version 4 library for Windows
php5_sqlanywhere9.dll	PHP version 4 library for Windows
php4_sqlanywhere9.so	Non-threaded PHP version 4 library for UNIX
php4_sqlanywhere9_r.so	Threaded PHP version 4 library for UNIX
php5_sqlanywhere9.so	Non-threaded PHP version 5 library for UNIX
php5_sqlanywhere9_r.so	Threaded PHP version 5 library for UNIX

Installing the PHP module on UNIX and Mac OS X

To use the SQLAnywhere PHP module on UNIX, Linux, or Mac OS X, you must copy the shared object from the SQL Anywhere installation directory and add it to your PHP installation. Optionally, you may then add an entry to your PHP initialization file, *php.ini*, to load the module, so you do not need to load it manually in each script.

❖ **To install the PHP module on UNIX or Mac OS X**

1. Locate the *php.ini* file of your PHP installation, and open it in a text editor. Locate the line that specifies the location of the extension_dir directory.

2. Copy the shared object from the *lib* subdirectory of your SQL Anywhere

installation to the directory specified by the extension_dir entry in the *php.ini* file.

☞ For information on which version of the shared object to use, see "Choosing which PHP module to use" on page 483.

3. Optionally, add the following line to the php.ini file to automatically load the SQLAnywhere PHP driver. Alternatively, you can load it manually with a few extra lines of code at the start of each script that requires it. The entry must identify the shared object you copied, which will be either

```
extension=phpX_sqlanywhere9.so
```

or, for the thread-safe shared object,

```
extension=phpX_sqlanywhere9_r.so
```

where *X* is the PHP version number.

4. Before attempting to use the PHP module, verify that your environment is set up for Adaptive Server Anywhere. Depending on which shell you're using, enter the appropriate command to source the Adaptive Server Anywhere configuration script from the SQL Anywhere Studio installation directory:

In this shell . . .	. . . use this command
sh, ksh, or bash	`. /bin/asa_config.sh`
csh or tcsh	`source /bin/asa_config.-csh`

5. At a command prompt, change to the root directory of your SQL Anywhere installation. Enter the following command to start the Adaptive Server Anywhere Sample Database.

```
dbeng9 asademo.db
```

6. At a command prompt, change to the *src/php/examples* subdirectory of your SQL Anywhere installation. Enter the following command:

```
php connect.php
```

The message `Connected successfully` should appear. If the php command is not recognized, verify that it is in your path.

7. When you are done, stop the database server by pressing Q in the console window where it is running.

For more information, see "Creating PHP test pages" on page 488.

Installing the PHP module on Windows

To use the SQLAnywhere PHP module on Windows, you must copy the DLL from the SQL Anywhere installation directory and add it to your PHP installation. Optionally, you may then add an entry to your PHP initialization file to load the module, so you do not need to load it manually in each script.

❖ **To install the PHP module on Windows**

1. Locate the *php.ini* file of your PHP installation, and open it in a text editor. Locate the line that specifies the location of the extension_dir directory.

2. Copy the file *php_sqlanywhere9.dll* from the *win32* subdirectory of your SQL Anywhere installation to the directory specified by the extension_dir entry in the *php.ini* file.

3. Add the following line to the *php.ini* file to automatically load the SQLAnywhere PHP driver.

   ```
   extension=phpX_sqlanywhere9.dll
   ```

   An alternative to automatically loading the PHP driver is to load it manually in each script that requires it. For more information, see "Configuring SQLAnywhere PHP" on page 486.

4. At a command prompt, change to the root directory of your SQL Anywhere installation. Enter the following command to start the Adaptive Server Anywhere Sample Database.

   ```
   dbeng9 asademo.db
   ```

   The command starts a database server with a server name of **asademo**, as required by samples in this chapter. If you start the server from the Windows menu, it has a server name of **asademo9**, which is not compatible with the samples in this chapter.

5. At a command prompt, change to the *src\php\examples* subdirectory of your SQL Anywhere installation. Enter the following command:

   ```
   php connect.php
   ```

   The message Connected successfully should appear. If the php command is not recognized, verify that it is in your path.

6. When you are done, stop the Adaptive Server Anywhere database server by clicking Shutdown in the console window.

For more information, see

Configuring SQLAnywhere PHP

The behavior of the SQL Anywhere PHP driver can be controlled by setting values in the PHP initialization file, *php.ini*. The following entries are currently supported.

♦ **extension**   Causes PHP to automatically load the SQLAnywhere PHP module every time PHP starts. Adding this entry to your PHP initialization file is optional, but if you don't add it, each script you write must start with a few lines of code that ensure that this module is loaded. The following entry is used for Windows platforms.

```
extension=phpX_sqlanywhere9.dll
```

On UNIX platforms and Mac OS X, use one of the following entries. The second entry uses threads.

```
extension=phpX_sqlanywhere9.so

extension=phpX_sqlanywhere9_r.so
```

To ensure that the SQLAnywhere PHP module is loaded, you can prefix each script you write with the following lines of code. They are required only if the SQLAnywhere module is not always automatically loaded when PHP starts.

```
# Ensure that the SQLAnywhere PHP module is loaded
  if( !extension_loaded('sqlanywhere')) {
    if( strtoupper(substr(PHYP_OS, 0, 3) == 'WIN' )) {
      dl('phpX_sqlanywhere9.dll');
    } else {
      dl('phpX_sqlanywhere9.so');
    }
  }
```

♦ **allow_persistent**   The PHP module allows persistent connection when set to 1. It does not allow them when set to 0. The default value is 1.

```
sqlanywhere.allow_persistent=1
```

♦ **max_persistent**   Sets the maximum number of persistent connections. The default value is -1, which means no limit. The default value is -1.

```
sqlanywhere.max_persistent=-1
```

♦ **max_connections**   Sets the maximum number of connections that can be opened at once through the SQLAnywhere PHP module. The default value is -1, which means no limit.

```
sqlanywhere.max_connections=-1
```

♦ **auto_commit**   The database server to perform a commit operation
automatically immediately following the execution of each statement
when set to 1. Transactions should be ended manually with either the
sqlanywhere_commit() or sqlanywhere_rollback() procedures, as
appropriate, when set to 0. The default value is 1.

```
sqlanywhere.auto_commit=1
```

For more information, see

Running PHP test scripts in your web pages

This section describes how to write PHP test scripts that query the Adaptive Anywhere Sample database and display information about PHP.

Creating PHP test pages

The following instructions apply to all configurations.

To test whether PHP is set up properly, the following procedure describes how to create and run a web page that calls phpinfo(). phpinfo() is a PHP function that generates a page of system setup information. The output tells you whether PHP is working properly.

☞ For information on installing PHP, see *http://us2.php.net/install*.

❖ **To create a PHP information test page**

1. Create a file in your root web content directory named *info.php*.

   If you are not sure which directory to use, check your web server's configuration file. In Apache installations, the content directory is often called *htdocs*. If you are using Mac OS X, the web content directory name may depend on which account you are using:

   ♦ If you are the System Administrator on a Mac OS X system, use */Library/WebServer/Documents*.

   ♦ If you are a Mac OS X user, place the file in */Users/[your username]/Sites/*.

2. Insert the following code into this file:

   ```
   <? phpinfo() ?>
   ```

   Alternatively, once PHP is properly installed and configured, you can also create a test web page by issuing the following command at a command prompt.

   ```
   php -i > info.html
   ```

   This confirms that your installation of PHP and Apache are working together properly.

3. To test that PHP and Apache are working correctly with Adaptive Server Anywhere:

   a. Copy the file *connect.php* from your PHP examples directory to your root web content directory.

   b. From a web browser, access the *connect.php* page.

      The message Connected successfully should appear.

❖ **To create the query page that uses the SQLAnywhere PHP module**

1. Create a file containing the following PHP code in your root web content directory named *asa_test.php*:

2. Insert the following PHP code into this file:

```
<?
  $conn = sqlanywhere_connect( "uid=DBA;pwd=SQL" );
  $result = sqlanywhere_query( $conn, "select * from
        employee" );
  sqlanywhere_result_all( $result );
  sqlanywhere_free_result( $result );
  sqlanywhere_disconnect( $conn );
?>
```

Accessing your test web pages

The following procedure describes how to view your test pages from a web browser, after installing and configuring PHP and the SQLAnywhere PHP module.

❖ **To view your web pages**

1. Restart your web server.

   For example, to start the Apache web server, run the following command from the *bin* subdirectory of your Apache installation:

   ```
   apachectl start
   ```

2. On UNIX or Mac OS X, set the Adaptive Server Anywhere environment variables using one of the supplied scripts.

   Depending on which shell you're using, enter the appropriate command to source the Adaptive Server Anywhere configuration script from your SQL Anywhere installation directory:

In this shell . . .	. . . use this command
sh, ksh, or bash	`.  /bin/asa_config.sh`
csh or tcsh	`source /bin/asa_config.-` `csh`

3. Start the Adaptive Server Anywhere database server.

   For example, to access the test web pages described above, use the following command to start the Adaptive Server Anywhere Sample database.

```
dbeng9 asademo.db
```

4. To access the test pages from a browser that is running on the same machine as the server, enter the following URLs:

For this test page . . .	. . . use this URL
info.php	`http://localhost/info.php`
asa_test.php	`http://localhost/asa_-` `test.php`

If everything is configured correctly, the asa_test pages display the contents of the employee table.

Writing PHP scripts

This section describes how to write PHP scripts that use the SQLAnywhere PHP module to access Adaptive Server Anywhere databases.

The source code for these examples, as well as others, is located in the src/php/examples subdirectory of your SQL Anywhere installation.

Connecting to a database

To make a connection to a database, pass a standard Adaptive Server Anywhere connection string to the database server as a parameter to the sqlanywhere_connect function. The $<?$ and $?>$ tags tell the web server that it should let PHP execute the code that lies between them and replace it with the PHP output.

The source code for this example is contained in your SQL Anywhere installation in a file called *connect.php*.

```
<?
  # Ensure that the SQLAnywhere PHP module is loaded
  if( !extension_loaded('sqlanywhere')) {
    if( strtoupper(substr(PHYP_OS, 0, 3) == 'WIN' )) {
      dl('phpX_sqlanywhere9.dll');
    } else {
      dl('phpX_sqlanywhere9.so');
    }
  }
  # Connect using the default userid and password
  $conn = sqlanywhere_connect( "uid=DBA;pwd=SQL" );
  if( ! $conn ) {
    echo "Connection failed\n";
    return 0;
  } else {
    echo "Connected successfully\n";
    sqlanywhere_disconnect( $conn );
  }
?>
```

The first block of code verifies that the PHP module is loaded. If you added the line to your PHP initialization file to load it automatically, this block of code is unnecessary. If you did not configure PHP to automatically load the SQLAnywhere PHP module at start time, you must add this code to the other sample scripts.

The second block attempts to make a connection. For this code to succeed, the Adaptive Server Anywhere Sample database must be running.

Retrieving data from a database

One use of PHP scripts in web pages is to retrieve and display information contained in a database. The following examples demonstrate some useful techniques.

Simple select query
The following PHP code demonstrates a convenient way to include the result set of a select statement in a web page. This sample is designed to connect to the Adaptive Server Anywhere sample database a return a list of customers.

This code can be embedded in a web page, provided your web server is configured to execute PHP scripts.

The source code for this sample is contained in your SQL Anywhere installation in a file called *query.php*.

```
<?
  # Connect using the default userid and password
  $conn = sqlanywhere_connect( "uid=DBA;pwd=SQL" );
  if( ! $conn ) {
      echo "Connection failed\n";
      return 0;
  } else {
      echo "Connected successfully\n";
      sqlanywhere_disconnect( $conn );
  }
  # Execute a SELECT statement
  $result = sqlanywhere_query( $conn, "SELECT * FROM CUSTOMER"
          );
  if( ! $result ) {
      echo "sqlanywhere_query failed!";
      return 0;
  } else {
      echo "query completed successfully\n";
  }
  # Generate HTML from the result set
  sqlanywhere_result_all( $result );
  sqlanywhere_free_result( $result );

  # Disconnect
  sqlanywhere_disconnect( $conn );
?>
```

The sqlanywhere_result_all function fetches all the rows of the result set and generates an HTML output table to display them. The sqlanywhere_free_result function releases the resources used to store the result set.

Fetching by column name
In certain cases, you may not want to display all the data from a result set, or you may wish to display the data in a different manner. The following sample illustrates how you can exercise greater control over the output

492

format of the result set. PHP allows you to display as much information as you wish in whatever manner you choose.

The source code for this sample is contained in your SQL Anywhere installation in a file called *fetch.php*.

```
<?
  # Connect using the default userid and password
  $conn = sqlanywhere_connect( "uid=DBA;pwd=SQL" );
  if( ! $conn ) {
      echo "Connection failed\n";
      return 0;
  } else {
      echo "Connected successfully\n";
      sqlanywhere_disconnect( $conn );
  }
  # Execute a SELECT statement
  $result = sqlanywhere_query( $conn, "SELECT * FROM CUSTOMER"
          );
  if( ! $result ) {
      echo "sqlanywhere_query failed!";
      return 0;
  } else {
      echo "query completed successfully\n";
  }
  # Retrieve meta information about the results
  $num_cols = sqlanywhere_num_fields( $result );
  $num_rows = sqlanywhere_num_rows( $result );
  echo "Num of rows = $num_rows\n";
  echo "Num of cols = $num_cols\n";

  while( ($field = sqlanywhere_fetch_field( $result ) ) ) {
      echo "Field # : $field->id \n";
      echo "\tname   : $field->name \n";
      echo "\tlength : $field->length \n";
      echo "\ttype   : $field->type \n";
  }
  # Fetch all the rows
  $curr_row = 0;
  while( ($row = sqlanywhere_fetch_row( $result ))) {
      $curr_row++;
      $curr_col = 0;
      while( $curr_col < $num_cols ) {
          echo "$row[$curr_col]\t|";
          $curr_col++;
      }
      echo "\n";
  }
  # Clean up.
  sqlanywhere_free_result( $result );
  sqlanywhere_disconnect( $conn );
?>
```

The sqlanywhere_fetch_array function returns a single row from the table. The data can be retrieved by column names and column indexes. Two other similar methods are provided in the PHP interface: sqlanywhere_fetch_row returns a row that can be searched by column indexes only, while sqlanywhere_fetch_object returns a row that can be searched by column names only.

See the *fetch_object.php* example script for an example of sqlanywhere_fetch_object.

Nested result sets

When a SELECT statement is sent to the database, a result set is returned. The sqlanywhere_fetch_row and sqlanywhere_fetch_array functions retrieve data from the individual rows of a result set, returning each row as an array of columns that can be queried further.

The source code for this sample is contained in your SQL Anywhere installation in a file called *nested.php*.

```
<?
  # Connect using the default userid and password
  $conn = sqlanywhere_connect( "uid=DBA;pwd=SQL" );
  if( ! $conn ) {
      echo "Connection failed\n";
      return 0;
  } else {
      echo "Connected successfully\n";
      sqlanywhere_disconnect( $conn );
  }
```

```
# Retrieve the data and output HTML
echo "<BR>\n";
$query1 = "select table_id, table_name from systable";
$result = sqlanywhere_query( $conn, $query1 );
if( $result ) {
  $num_rows = sqlanywhere_num_rows( $result );
  echo "Returned : $num_rows <BR>\n";
  $i = 1;
  while( ($row = sqlanywhere_fetch_array( $result ))) {
    echo "$i:  table_id:$row[table_id]" +
        " --- table_name:$row[table_name] <br>\n";
    $query2 = "select table_id, column_name " +
    "from syscolumn " +
    "where table_id = '$row[table_id]'";
    echo " $query2 <br>\n";
    echo "  Columns: ";
    $result2 = sqlanywhere_query( $conn, $query2 );
    if( $result2 ) {
    while(($detailed = sqlanywhere_fetch_array($result2))) {
        echo " $detailed[column_name]";
    }
    sqlanywhere_free_result( $result2 );
    } else {
    echo "******FAILED********";
    }
    echo "<br>\n";
    $i++;
  }
}
echo "<BR>\n";
?>
```

In the above sample, the table ID and name are selected for each table from
SYSTABLE. The sqlanywhere_query function returns an array of rows. The
script iterates through the rows using sqlanywhere_fetch_array to retrieve
the columns of the row as an array. An inner iteration goes through the
columns of the row and prints their names.

Web forms

PHP can take user input from a web form, pass it to the database server as a
SQL query, and display the result that is returned. The following example
demonstrates a simple web form that gives the user the ability to query the
asademo sample database using SQL statements and display the results in an
HTML table.

The source code for this sample is contained in your SQL Anywhere
installation in a file called *webisql.php*.

```
<?
  # Connect using the default userid and password
  $conn = sqlanywhere_connect( "uid=DBA;pwd=SQL" )
          or die("Can not connect to database");

  echo "<HTML>\n";
  $qname = $_POST[qname];
  $qname = str_replace( "\\", "", $qname );
  echo "<form method=post action=webisql.php>\n";
  echo "<br>Query: <input type=text size=80 name=qname value=\
        "$qname\">\n";
  echo "<input type=submit>\n";
  echo "</form>\n";
  echo "<HR><br>\n";

  if( ! $qname ) {
      echo "No Current Query\n";
      return;
  }
  # Connect to the database
  $con_str = "uid=DBA;pwd=SQL;eng=asademo;links=tcpip";
  $conn = sqlanywhere_connect( $con_str );
  if( ! $conn ) {
      echo "sqlanywhere_connect failed\n";
      echo "</html>\n";
      return 0;
  }

  $qname = str_replace( "\\", "", $qname );
  $result = sqlanywhere_query( $conn, $qname );

  if( ! $result ) {
      echo "sqlanywhere_query failed!";
  } else {
      // echo "query completed successfully\n";
      sqlanywhere_result_all( $result, "border=1" );
      sqlanywhere_free_result( $result );
  }
  sqlanywhere_disconnect( $conn );
  echo "</html>\n";
?>
```

This design could be extended to handle complex web forms by formulating customized SQL queries based on the values entered by the user.

Working with BLOBs

Adaptive Server Anywhere databases have the ability to store any type of data as a binary large object (BLOB). If that data is of a type readable by a web browser, a PHP script can easily retrieve it from the database and display it on a dynamically generated page.

BLOB fields are often used for storing non-text data, such as images.

Numerous types of data can be passed to a web browser without any need for third-party software or data type conversion. The following sample illustrates the process of adding an image to the database and then retrieving it again to be displayed in a web browser.

This sample example illustrates the use of a BLOB column for storing large pieces of text.

The source code for this sample is contained in your SQL Anywhere installation in a file called *image_insert.php*.

```
<?
  $conn = sqlanywhere_connect( "uid=DBA;pwd=SQL" )
        or die("Can not connect to database");

  $create_table = "CREATE TABLE images (id INTEGER PRIMARY KEY,
        img IMAGE)";
  sqlanywhere_query( $conn, $create_table);

  $insert = "INSERT INTO images VALUES (99, xp_read_
        file('ianywhere_logo.gif'))";
  sqlanywhere_query( $conn, $insert );

  $query = "SELECT img FROM images WHERE id = 99";
  $result = sqlanywhere_query($conn, $query);

  $data = sqlanywhere_fetch_row($result);
  $img = $data[0];

  header("Content-type: image/gif");
  echo $img;

  sqlanywhere_disconnect($conn);
?>
```

In order to be able to send the binary data from the database directly to a web browser, the script must set the data's MIME type using the header function. In this case, the browser is told to expect a GIF image so it can display it correctly.

Troubleshooting

♦ For more help, you can look in the following places:

- **The Newsgroups** **sybase.public.sqlanywhere.general** and **sybase.public.sqlanywhere.linux** are most likely to be relevant, and can be read from the web or with a newsgroup reader. Be sure to look through old threads for your problem, as it may already have been addressed.

  If you use a news reader, the newsgroups are available on **forums.sybase.com**. On the web, they can be viewed at *http://www.sybase.com/support/newsgroups*.

- **A FAQ** is available for the UNIX version of Adaptive Server Anywhere at *http://www.sybase.com/detail?id=1011965*.

- Check if there have been any bug fixes or updates posted at *http://downloads.sybase.com/swx/sdmain.stm*.

SQLAnywhere PHP Module interface reference

This section provides reference information about the procedures provided by the PHP module.

sqlanywhere_commit

Prototype	bool **sqlanywhere_commit(** int *link_identifier* **)**
Description	Ends a transaction on the Adaptive Server Anywhere database and makes any changes made during the transaction permanent. Useful only when the AUTO_COMMIT option is OFF.
Parameters	♦ **link_identifier** The link identifier returned by a sqlanywhere_connect function.
Returns	TRUE on success or FALSE on failure.
Example	`$result = sqlanywhere_commit( $conn );`
Related functions	♦ "sqlanywhere_rollback" on page 505 ♦ "sqlanywhere_set_option" on page 505

sqlanywhere_connect

Prototype	int **sqlanywhere_connect(** string *con_str* **)**
Description	Establishes a connection to an Adaptive Server Anywhere database.

Parameters	♦ **con_str** A connection string as recognized by Adaptive Server Anywhere.
Returns	A positive Adaptive Server Anywhere link identifier on success, or an error and 0 on failure.
Example	`$conn = sqlanywhere_connect( "UID=dba;PWD=sql" );`
Related functions	♦ "sqlanywhere_pconnect" on page 504
	♦ "sqlanywhere_disconnect" on page 500

sqlanywhere_data_seek

Prototype	bool **sqlanywhere_data_seek(** int *result_identifier*, *int row_num* **)**
Description	Positions the cursor on row *row_num* on the *result_identifier* that was opened using sqlanywhere_query.
Parameters	♦ **result_identifier** The result identifier returned by a sqlanywhere_query function.
	♦ **row_num** An integer that represents the position that you want to move to within the result_identifier. For example, specify 5 to move to the 5th row in the result set. Negative numbers represent a position prior to the current position or moving backwards through the result set.-1 moves you to the last row in the result set.
Returns	TRUE on success or FALSE on error.
Example	`sqlanywhere_data_seek( $result, 5 );`
Related functions	♦ "sqlanywhere_fetch_field" on page 501
	♦ "sqlanywhere_fetch_array" on page 501
	♦ "sqlanywhere_fetch_row" on page 502
	♦ "sqlanywhere_fetch_object" on page 502
	♦ "sqlanywhere_query" on page 504

sqlanywhere_disconnect

Prototype	bool **sqlanywhere_disconnect(** int *link_identifier* **)**
Description	Closes a connection that has already been opened with sqlanywhere_connect.
Parameters	♦ **link_identifier** The link identifier returned by a sqlanywhere_connect function.
Returns	TRUE on success or FALSE on error.
Example	`sqlanywhere_disconnect( $conn );`

| Related functions | ♦ "sqlanywhere_connect" on page 499 |
| | ♦ "sqlanywhere_pconnect" on page 504 |

sqlanywhere_fetch_array

Prototype	array **sqlanywhere_fetch_array(** int *result_identifier* **)**
Description	Fetches one row from the result set. This row is returned as an array that can be indexed by the column names or by the column indexes.
Parameters	♦ **result_identifier** The result set returned by a sqlanywhere_query function.
Returns	An array that represents a row from the result set, or FALSE when no rows are available.
Example	

```
$result = sqlanywhere_query( $conn, "SELECT fname, lname FROM
        employee" );
  While( ($row = sqlanywhere_fetch_array( $result )) ) {
        echo " fname = $row["fname"] \n" ;
        echo " lname = $row[1] \n";
  }
```

Related functions	♦ "sqlanywhere_data_seek" on page 500
	♦ "sqlanywhere_fetch_field" on page 501
	♦ "sqlanywhere_fetch_row" on page 502
	♦ "sqlanywhere_fetch_object" on page 502

sqlanywhere_fetch_field

Prototype	object **sqlanywhere_fetch_field(** int *result_identifier* [, *field_offset*] **)**
Description	Returns an object that contains information about a specific column.
Parameters	♦ **result_identifier** The result identifier returned by a sqlanywhere_query function.
	♦ **field_offset** An integer representing the column/field on which you wish to retrieve information. Columns are zero based; to get the first column, specify the value **0**. If this parameter is omitted, then the next field object is returned.
Returns	An object that has the following properties:
	♦ **id** contains the field/column number
	♦ **name** contains the field/column name
	♦ **numeric** indicates whether or not the field is a numeric value **length** returns field length **type** returns field type

501

Related functions	♦ "sqlanywhere_data_seek" on page 500
	♦ "sqlanywhere_fetch_array" on page 501
	♦ "sqlanywhere_fetch_row" on page 502
	♦ "sqlanywhere_fetch_object" on page 502

sqlanywhere_fetch_object

Prototype	array **sqlanywhere_fetch_object(** int *result_identifier* **)**
Description	Fetches one row from the result set. This row is returned as an array that can be indexed by the column name only.
Parameters	♦ **result_identifier** The result identifier returned by a sqlanywhere_query function.
Returns	An array that represents a row from the result set, or FALSE when no rows are available.
Example	`$result = sqlanywhere_query( $conn, "SELECT fname, lname FROM`
	`      employee" );`
	`  While( ($row = sqlanywhere_fetch_array( $result )) ) {`
	`        echo "$row["fname"] \n";     # output the data in the`
	`        first column only.`
	`  }`
Related functions	♦ "sqlanywhere_data_seek" on page 500
	♦ "sqlanywhere_fetch_field" on page 501
	♦ "sqlanywhere_fetch_array" on page 501
	♦ "sqlanywhere_fetch_row" on page 502

sqlanywhere_fetch_row

Prototype	array **sqlanywhere_fetch_row(** int *result_identifier* **)**
Description	Fetches one row from the result set. This row is returned as an array that can be indexed by the column indexes only.
Parameters	♦ **result_identifier** The result identifier returned by a sqlanywhere_query function.
Returns	An array that represents a row from the result set, or FALSE when no rows are available.
Example	

```
while( ($row = sqlanywhere_fetch_array( $result )) ) {
        echo "$row[0] \n";     # output the data in the first
         col only.
     }
```

Related functions
- "sqlanywhere_data_seek" on page 500
- "sqlanywhere_fetch_field" on page 501
- "sqlanywhere_fetch_array" on page 501
- "sqlanywhere_fetch_object" on page 502

sqlanywhere_free_result

Prototype	bool **sqlanywhere_free_result(** int *result_identifier* **)**
Description	Frees database resources associated with a result identifier returned from sqlanywhere_query.
Parameters	♦ **result_identifier** The result identifier returned by a sqlanywhere_query function.
Returns	TRUE on success or FALSE on error.
Example	`sqlanywhere_free_result( $result );`
Related functions	♦ "sqlanywhere_query" on page 504

sqlanywhere_num_fields

Prototype	int **sqlanywhere_num_fields(** int *result_identifier* **)**
Description	Returns the number of columns (fields) the *result_identifier* contains.
Parameters	♦ **result_identifier** The result identifier returned by a sqlanywhere_query function.
Returns	A positive number of columns, or an error if *result_identifier* is not valid.
Example	`sqlanywhere_num_fields( $result );`

sqlanywhere_num_rows

Prototype	int **sqlanywhere_num_rows(** int *result_identifier* **)**
Description	Returns the number of rows that the *result_identifier* contains.
Parameters	♦ **result_identifier** The result identifier returned by a sqlanywhere_query function.
Returns	A positive number if the number of rows is exact, or a negative number if it is an estimate. To get the exact number of rows, the database option

ROW_COUNTS must be set permanently on the database, or temporarily on the connection.

Example

```
sqlanywhere_num_rows( $result );
```

Related functions

♦ "sqlanywhere_query" on page 504

sqlanywhere_pconnect

Prototype

int **sqlanywhere_pconnect(** string *con_str* **)**

Description

Establishes a persistent connection to an Adaptive Server Anywhere database. Because of the way Apache creates child processes, you may observe a performance gain when using sqlanywhere_pconnect instead of sqlanywhere_connect. Persistent connections may provide improved performance in a similar fashion to connection pooling. If your database server has a limited number of connections (for example, the personal database server is limited to 10 concurrent connections), caution should be exercised when using persistent connections. Persistent connections could be attached to each of the child processes, and if you have more child processes in Apache than there are available connections, you will receive connection errors.

Parameters

♦ **con_str** A connection string as recognized by Adaptive Server Anywhere.

Returns

A positive SQLAnywhere persistent link identifier on success, or an error and 0 on failure.

Example

```
$conn = sqlanywhere_pconnect( "UID=dba;PWD=sql" );
```

Related functions

♦ "sqlanywhere_connect" on page 499
♦ "sqlanywhere_disconnect" on page 500

sqlanywhere_query

Prototype

int **sqlanywhere_query(** int *link_identifier*, string *sql_str* **)**

Description

Prepares and executes the SQL query *sql_str* on the connection identified by *link_identifier* that has already been opened using sqlanywhere_connect or sqlanywhere_pconnect.

Parameters

♦ **link_identifier** The link identifier returned by a sqlanywhere_connect function.

♦ **sql_str** A SQL statement supported by Adaptive Server Anywhere. For more information about SQL statements, see the Adaptive Server Anywhere SQL Reference manual.

Returns	A positive value representing the result ID on success, or 0 and an error message on failure.

Example	`$result = sqlanywhere_query( $conn, "SELECT * FROM systable" );`

Related functions	♦ "sqlanywhere_free_result" on page 503

sqlanywhere_result_all

Prototype	bool **sqlanywhere_result_all(** int *result_identifier* [, *html_table_format_string*] **)**

Description	Fetches all results of the *result_identifier* and generates an HTML output table with an optional formatting string.

Parameters	♦ **result_identifier** The result identifier returned by a sqlanywhere_query function.
	♦ **html_table_format_string** A format string that applies to HTML tables. For example, **"Border=1; Cellpadding=5"**.

Returns	TRUE on success or FALSE on failure.

Example	`$result = sqlanywhere_query( $conn, "SELECT fname, lname FROM`
	`          employee" );`
	`  sqlanywhere_result_all( $result );`

Related functions	♦ "sqlanywhere_query" on page 504

sqlanywhere_rollback

Prototype	bool **sqlanywhere_rollback(** int *link_identifier* **)**

Description	Ends a transaction on the Adaptive Server Anywhere database and discards any changes made during the transaction. Useful only when the AUTO_COMMIT option is OFF.

Parameters	♦ **link_identifier** The link identifier returned by a sqlanywhere_connect function.

Returns	TRUE on success or FALSE on failure.

Example	`$result = sqlanywhere_rollback( $conn );`

Related functions	♦ "sqlanywhere_commit" on page 499 ♦ "sqlanywhere_set_option" on page 505

sqlanywhere_set_option

Prototype	bool **sqlanywhere_rollback(** int *link_identifier*, string *option*, string *value* **)**

Description	Sets the value of the specified option on the specified connection. Currently, the only supported option is AUTO_COMMIT. The default value of this option is ON, which causes the database server to commit after executing each statement.

You can change the default value to OFF by including the following line in the *php.ini* file.

```
sqlanywhere.auto_commit=0
```

Parameters
- ♦ **link_identifier**   The link identifier returned by a sqlanywhere_connect function.

- ♦ **option**   The name of the option you want to set.

- ♦ **value**   The new option value.

Returns   TRUE on success or FALSE on failure.

Example
```
$result = sqlanywhere_set_option( $conn, "AUTO_COMMIT", "OFF" );
```

Related functions
- ♦
- ♦

CHAPTER 17

Three-Tier Computing and Distributed Transactions

About this chapter   This chapter describes how to use Adaptive Server Anywhere in a three-tier environment with an application server. It focuses on how to enlist Adaptive Server Anywhere in distributed transactions.

Contents

Introduction

You can use Adaptive Server Anywhere as a database server or **resource manager**, participating in distributed transactions coordinated by a transaction server.

A three-tier environment, where an application server sits between client applications and a set of resource managers, is a common distributed-transaction environment. Sybase EAServer and some other application servers are also transaction servers.

Sybase EAServer and Microsoft Transaction Server both use the Microsoft Distributed Transaction Coordinator (DTC) to coordinate transactions. Adaptive Server Anywhere provides support for distributed transactions controlled by the DTC service, so you can use Adaptive Server Anywhere with either of these application servers, or any other product based on the DTC model.

When integrating Adaptive Server Anywhere into a three-tier environment, most of the work needs to be done from the Application Server. This chapter provides an introduction to the concepts and architecture of three-tier computing, and an overview of relevant Adaptive Server Anywhere features. It does not describe how to configure your Application Server to work with Adaptive Server Anywhere. For more information, see your Application Server documentation.

Three-tier computing architecture

In three-tier computing, application logic is held in an application server, such as Sybase EAServer, which sits between the resource manager and the client applications. In many situations, a single application server may access multiple resource managers. In the Internet case, client applications are browser-based, and the application server is generally a Web server extension.
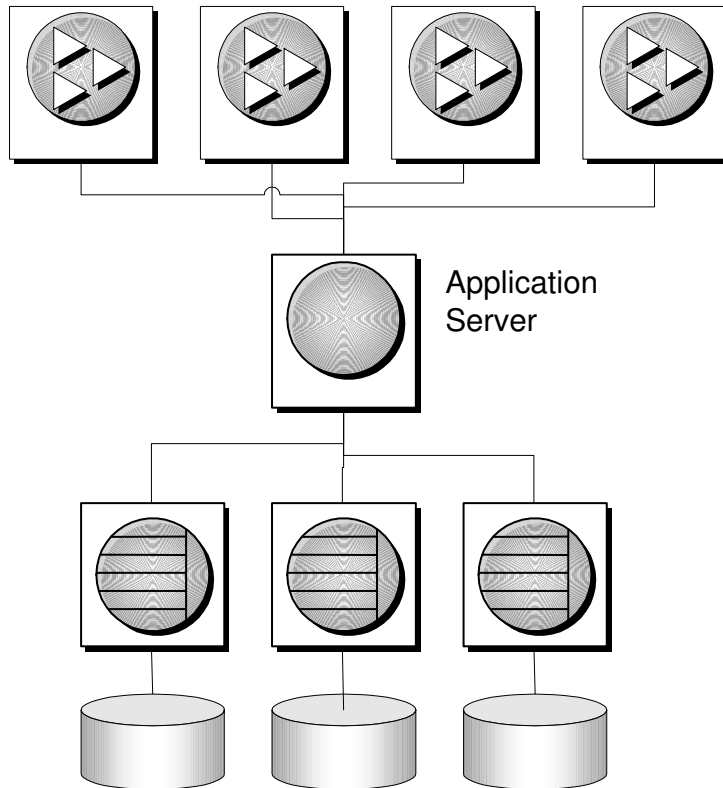
Sybase EAServer stores application logic in the form of components, and makes these components available to client applications. The components may be PowerBuilder components, JavaBeans, or COM components.

☞ For more information, see the Sybase EAServer documentation.

Distributed transactions in three-tier computing

When client applications or application servers work with a single transaction processing database, such as Adaptive Server Anywhere, there is

no need for transaction logic outside the database itself, but when working with multiple resource managers, transaction control must span the resources involved in the transaction. Application servers provide transaction logic to their client applications—guaranteeing that sets of operations are executed atomically.

Many transaction servers, including Sybase EAServer, use the Microsoft Distributed Transaction Coordinator (DTC) to provide transaction services to their client applications. DTC uses **OLE transactions**, which in turn use the **two-phase commit** protocol to coordinate transactions involving multiple resource managers. You must have DTC installed in order to use the features described in this chapter.

Adaptive Server Anywhere in distributed transactions

Adaptive Server Anywhere can take part in transactions coordinated by DTC, which means that you can use Adaptive Server Anywhere databases in distributed transactions using a transaction server such as Sybase EAServer or Microsoft Transaction Server. You can also use DTC directly in your applications to coordinate transactions across multiple resource managers.

The vocabulary of distributed transactions

This chapter assumes some familiarity with distributed transactions. For information, see your transaction server documentation. This section describes some commonly used terms.

♦ **Resource managers** are those services that manage the data involved in the transaction.

The Adaptive Server Anywhere database server can act as a resource manager in a distributed transaction when accessed through OLE DB or ODBC. The ODBC driver and OLE DB provider act as resource manager proxies on the client machine.

♦ Instead of communicating directly with the resource manager, application components may communicate with **resource dispensers**, which in turn manage connections or pools of connections to the resource managers.

Adaptive Server Anywhere supports two resource dispensers: the ODBC driver manager and OLE DB.

♦ When a transactional component requests a database connection (using a resource manager), the application server **enlists** each database connection takes part in the transaction. DTC and the resource dispenser carry out the enlistment process.

Two-phase commit

Distributed transactions are managed using two-phase commit. When the work of the transaction is complete, the transaction manager (DTC) asks all

the resource managers enlisted in the transaction whether they are ready to commit the transaction. This phase is called **preparing** to commit.

If all the resource managers respond that they are prepared to commit, DTC sends a commit request to each resource manager, and responds to its client that the transaction is completed. If one or more resource manager does not respond, or responds that it cannot commit the transaction, all the work of the transaction is rolled back across all resource managers.

How application servers use DTC

Sybase EAServer and Microsoft Transaction Server are both component servers. The application logic is held in the form of components, and made available to client applications.

Each component has a transaction attribute that indicates how the component participates in transactions. The application developer building the component must program the work of the transaction into the component—the resource manager connections, the operations on the data for which each resource manager is responsible. However, the application developer does not need to add transaction management logic to the component. Once the transaction attribute is set, to indicate that the component needs transaction management, EAServer uses DTC to enlist the transaction and manage the two-phase commit process.

Distributed transaction architecture

The following diagram illustrates the architecture of distributed transactions. In this case, the resource manager proxy is either ODBC or OLE DB.
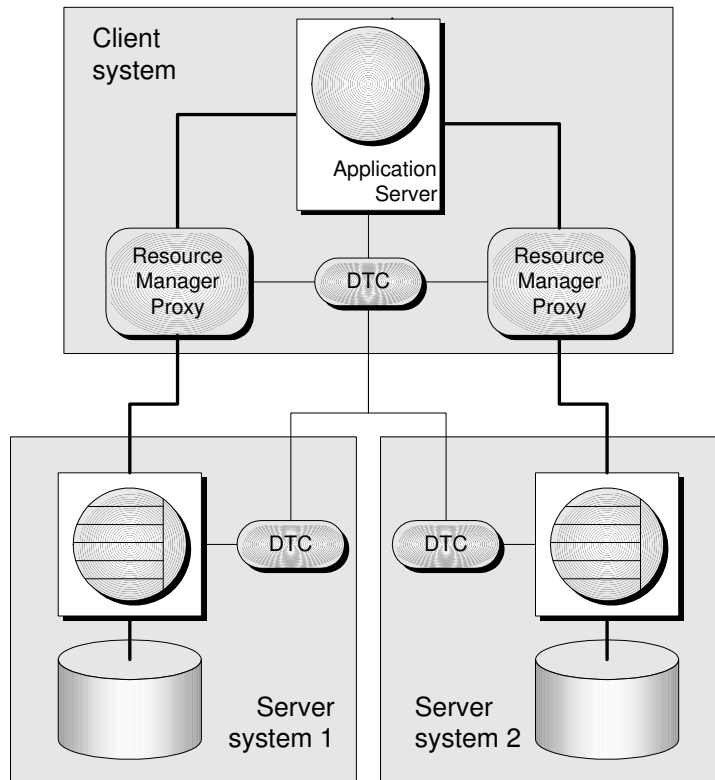
In this case, a single resource dispenser is used. The Application Server asks DTC to prepare a transaction. DTC and the resource dispenser enlist each connection in the transaction. Each resource manager must be in contact with both DTC and the database, so as to carry out the work and to notify DTC of its transaction status when required.

A DTC service must be running on each machine in order to operate distributed transactions. You can control DTC services from the Services icon in the Windows control panel; the DTC service is named **MSDTC**.

☞ For more information, see your DTC or EAServer documentation.

Using distributed transactions

While Adaptive Server Anywhere is enlisted in a distributed transaction, it hands transaction control over to the transaction server, and Adaptive Server Anywhere ensures that it does not carry out any implicit transaction management. The following conditions are imposed automatically by Adaptive Server Anywhere when it participates in distributed transactions:

♦ Autocommit is automatically turned off, if it is in use.

♦ Data definition statements (which commit as a side effect) are disallowed during distributed transactions.

♦ An explicit COMMIT or ROLLBACK issued by the application directly to Adaptive Server Anywhere, instead of through the transaction coordinator, generates an error. The transaction is not aborted, however.

♦ A connection can participate in only a single distributed transaction at a time.

♦ There must be no uncommitted operations at the time the connection is enlisted in a distributed transaction.

DTC isolation levels

DTC has a set of isolation levels, which the application server specifies. These isolation levels map to Adaptive Server Anywhere isolation levels as follows:

DTC isolation level	Adaptive Server Anywhere isolation level
ISOLATIONLEVEL_UNSPECIFIED	0
ISOLATIONLEVEL_CHAOS	0
ISOLATIONLEVEL_-READUNCOMMITTED	0
ISOLATIONLEVEL_BROWSE	0
ISOLATIONLEVEL_CURSORSTABILITY	1
ISOLATIONLEVEL_READCOMMITTED	1
ISOLATIONLEVEL_REPEATABLEREAD	2
ISOLATIONLEVEL_SERIALIZABLE	3
ISOLATIONLEVEL_ISOLATED	3

Recovery from distributed transactions

If the database server faults while uncommitted operations are pending, it must either rollback or commit those operations on startup to preserve the atomic nature of the transaction.

If uncommitted operations from a distributed transaction are found during recovery, the database server attempts to connect to DTC and requests that it be re-enlisted in the pending or in-doubt transactions. Once the re-enlistment is complete, DTC instructs the database server to roll back or commit the outstanding operations.

If the reenlistment process fails, Adaptive Server Anywhere has no way of knowing whether the in-doubt operations should be committed or rolled back, and recovery fails. If you want the database in such a state to recover, regardless of the uncertain state of the data, you can force recovery using the following database server options:

♦ **-tmf**   If DTC cannot be located, the outstanding operations are rolled back and recovery continues.

☞ For more information, see "-tmf server option" [*ASA Database Administration Guide,* page 158].

♦ **-tmt**   If re-enlistment is not achieved before the specified time, the outstanding operations are rolled back and recovery continues.

☞ For more information, see "-tmt server option" [*ASA Database Administration Guide,* page 159].

Using EAServer with Adaptive Server Anywhere

This section provides an overview of the actions you need to take in EAServer 3.0 or later to work with Adaptive Server Anywhere. For more detailed information, see the EAServer documentation.

Configuring EAServer

All components installed in a Sybase EAServer share the same transaction coordinator.

EAServer 3.0 and later offer a choice of transaction coordinators. You must use DTC as the transaction coordinator if you are including Adaptive Server Anywhere in the transactions. This section describes how to configure EAServer 3.0 to use DTC as its transaction coordinator.

The component server in EAServer is named Jaguar.

❖ **To configure an EAServer to use the Microsoft DTC transaction model**

1. Ensure that your Jaguar server is running.

   On Windows, the Jaguar server commonly runs as a service. To manually start the installed Jaguar server that comes with EAServer 3.0, select Start ➤ Programs ➤ Sybase ➤ EAServer ➤ EAServer.

2. Start Jaguar Manager.

   From the Windows desktop, select Start ➤ Programs ➤ Sybase ➤ EAServer ➤ Jaguar Manager.

3. Connect to the Jaguar server from Jaguar Manager.

   From the Sybase Central menu, choose Tools ➤ Connect ➤ Jaguar Manager. In the connection dialog, enter **jagadmin** as the User Name, leave the Password field blank, and enter a Host Name of **localhost**. Click OK to connect.

4. Set the transaction model for the Jaguar server.

   In the left pane, open the Servers folder. In the right pane, right click on the server you wish to configure, and select Server Properties from the drop down menu. Click the Transactions tab, and choose Microsoft DTC as the transaction model. Click OK to complete the operation.

Setting the component transaction attribute

In EAServer you may implement a component that carries out operations on more than one database. You assign a **transaction attribute** to this

component that defines how it participates in transactions. The transaction attribute can have the following values:

- **Not Supported**   The component's methods never execute as part of a transaction. If the component is activated by another component that is executing within a transaction, the new instance's work is performed outside the existing transaction. This is the default.

- **Supports Transaction**   The component can execute in the context of a transaction, but a connection is not required in order to execute the component's methods. If the component is instantiated directly by a base client, EAServer does not begin a transaction. If component A is instantiated by component B, and component B is executing within a transaction, component A executes in the same transaction.

- **Requires Transaction**   The component always executes in a transaction. When the component is instantiated directly by a base client, a new transaction begins. If component A is activated by component B, and B is executing within a transaction, then A executes within the same transaction; if B is not executing in a transaction, then A executes in a new transaction.

- **Requires New Transaction**   Whenever the component is instantiated, a new transaction begins. If component A is activated by component B, and B is executing within a transaction, then A begins a new transaction that is unaffected by the outcome of B's transaction; if B is not executing in a transaction, then A executes in a new transaction.

For example, in the Sybase Virtual University sample application, included with EAServer as the SVU package, the **SVUEnrollment** component **enroll()** method carries out two separate operations (reserves a seat in a course, bills the student for the course). These two operations need to be treated as a single transaction.

Microsoft Transaction Server provides the same set of attribute values.

❖ **To set the transaction attribute of a component**

1. In Jaguar Manager, locate the component.

   To find the **SVUEnrollment** component in the Jaguar sample application, connect to the Jaguar server, open the Packages folder, and open the SVU package. The components in the package are listed in the right pane.

2. Set the transaction attribute for the desired component.

   Right click the component, and select Component Properties from the popup menu. Click the Transaction tab, and choose the transaction attribute value from the list. Click OK to complete the operation.

The **SVUEnrollment** component is already marked as Requires Transaction.

Once the component transaction attribute is set, you can carry out Adaptive Server Anywhere operations from that component, and be assured of transaction processing at the level you have specified.

Deploying Databases and Applications

About this chapter

This chapter describes how to deploy Adaptive Server Anywhere components. It identifies the files required for deployment, and addresses related issues such as connection settings.

> **Check your license agreement**
> Redistribution of files is subject to your license agreement. No statements in this document override anything in your license agreement. Please check your license agreement before considering deployment.

Contents

Deployment overview

When you have completed a database application, you must deploy the application to your end users. Depending on the way in which your application uses Adaptive Server Anywhere (as an embedded database, in a client/server fashion, and so on) you may have to deploy components of the Adaptive Server Anywhere software along with your application. You may also have to deploy configuration information, such as data source names, that enable your application to communicate with Adaptive Server Anywhere.

Check your license agreement

Redistribution of files is subject to your license agreement with Sybase. No statements in this document override anything in your license agreement. Please check your license agreement before considering deployment.

The following deployment steps are examined in this chapter:

♦ Determining required files based on the choice of application platform and architecture.

♦ Configuring client applications.

Much of the chapter deals with individual files and where they need to be placed. However, the recommended way of deploying Adaptive Server Anywhere components is to use the InstallShield objects or to use a silent installation. For information, see "Using InstallShield for deployment" on page 526, and "Using a silent installation for deployment" on page 527.

Deployment models

The files you need to deploy depend on the deployment model you choose. Here are some possible deployment models:

♦ **Client deployment**   You may deploy only the client portions of Adaptive Server Anywhere to your end-users, so that they can connect to a centrally located network database server.

♦ **Network server deployment**   You may deploy network servers to offices, and then deploy clients to each of the users within those offices.

♦ **Embedded database deployment**   You may deploy an application that runs with the personal database server. In this case, both client and personal server need to be installed on the end-user's machine.

♦ **SQL Remote deployment**   Deploying a SQL Remote application is an extension of the embedded database deployment model.

♦ **MobiLink deployment**   For information on deploying MobiLink synchronization servers, see "Deploying MobiLink Applications" [*MobiLink Administration Guide,* page 549].

♦ **Administration tools deployment**   You may deploy Interactive SQL, Sybase Central and other management tools.

Ways to distribute files

There are two ways to deploy Adaptive Server Anywhere:

♦ **Use the Adaptive Server Anywhere installation**   You can make the Setup program available to your end-users. By selecting the proper option, each end-user is guaranteed of getting the files they need.

This is the simplest solution for many deployment cases. In this case, you must still provide your end users with a method for connecting to the database server (such as an ODBC data source).

☞ For more information, see "Using a silent installation for deployment" on page 527.

♦ **Develop your own installation**   There may be reasons for you to develop your own installation program that includes Adaptive Server Anywhere files. This is a more complicated option, and most of this chapter addresses the needs of those who are developing their own installation.

If Adaptive Server Anywhere has already been installed for the server type and operating system required by the client application architecture, the required files can be found in the appropriately named subdirectory, located in the Adaptive Server Anywhere installation directory.

For example, assuming the default installation directory was chosen, the *win32* subdirectory of your installation directory contains the files required to run the server for Windows operating systems.

As well, users of InstallShield Professional 5.5 and up can use the SQL Anywhere Studio InstallShield Template Projects to deploy their own application. This feature allows you to quickly build your application's installation using the entire template project, or just the parts that apply to your install.

Whichever option you choose, you must not violate the terms of your license agreement.

Understanding installation directories and file names

For a deployed application to work properly, the database server and client libraries must each be able to locate the files they need. The deployed files should be located relative to each other in the same fashion as your Adaptive Server Anywhere installation.

In practice, this means that on PCs, most files belong in a single directory. For example, on Windows both client and database server required files are installed in a single directory, which is the *win32* subdirectory of the Adaptive Server Anywhere installation directory.

☞ For a full description of the places where the software looks for files, see "How Adaptive Server Anywhere locates files" [*ASA Database Administration Guide,* page 274].

UNIX deployment issues

UNIX deployments are different from PC deployments in some ways:

♦ **Directory structure** For UNIX installations, the directory structure is as follows:

Directory	Contents
/opt/sybase/SYBSsa9/bin	Executable files
/opt/sybase/SYBSsa9/lib	Shared objects and libraries
/opt/sybase/SYBSsa9/res	String files

On AIX, the default root directory is */usr/lpp/sybase/SYBSsa9* instead of */opt/sybase/SYBSsa9.*

♦ **File extensions** In the tables in this chapter, the shared objects are listed with an extension *.so.* For HP-UX, the extension is *.sl.*

On the AIX operating system, shared objects that applications need to link to are given the extension *.a.*

♦ **Symbolic links** Each shared object is installed as a symbolic link to a file of the same name with the additional extension *.1* (one). For example, the *libdblib9.so* is a symbolic link to the file *libdblib9.so.1* in the same directory.

If patches are required to the Adaptive Server Anywhere installation, these will be supplied with extension *.2,* and the symbolic link must be redirected.

♦ **Threaded and unthreaded applications**   Most shared objects are provided in two forms, one of which has the additional characters _*r* before the file extension. For example, in addition to *libdblib9.so*, there is a file named *libdblib9_r.so*. In this case, threaded applications must be linked to the _*r* shared object, while non-threaded applications must be linked to the shared object without the _*r* characters.

♦ **Character set conversion**   If you want to use database server character set conversion (the -ct server option), you need to include the following files:

 • *libunic.so*

 • *charsets/* directory subtree

 • *asa.cvf*

 • *asa.cvu*

♦ **Environment variables**   On UNIX, environment variables must be set for the system to be able to locate SQL Anywhere Studio applications and libraries. It is recommended that you use the appropriate file for your shell, either *asa_config.sh* or *asa_config.csh* (located in the directory */opt/sybase/SYBSsa9/bin*) as a template for setting the required environment variables. Some of the environment variables set by these files include PATH, LD_LIBRARY_PATH, ASANY9, and ASANYSH9.

☞ For a description of how Adaptive Server Anywhere looks for files, see "How Adaptive Server Anywhere locates files" [*ASA Database Administration Guide,* page 274].

File naming conventions

Adaptive Server Anywhere uses consistent file naming conventions to help identify and group system components.

These conventions include:

♦ **Version number**   The Adaptive Server Anywhere version number is indicated in the filename of the main server components (.*exe* and .*dll* files).

For example, the file *dbeng9.exe* is a Version 9 executable.

♦ **Language**   The language used in a language resource library is indicated by a two-letter code within its filename. The two characters before the version number indicate the language used in the library. For example, *dblgen9.dll* is the language resource library for English. These two-letter codes are specified by ISO standard 639.

☞ For more information about language labels, see "Understanding the locale language" [*ASA Database Administration Guide,* page 331].

You can download an International Resources Deployment Kit containing language resource deployment DLLs free of charge from the Sybase Web site.

❖ **To download the International Resources Deployment Kit from the Sybase Web site**

1. Open the following URL in your Web browser:

    http://www.ianywhere.com/developer/

2. Under the heading Downloads on the left hand side of the page, click EBFs/Patches.

3. Login to your Sybase Web account.

    Click Create a New Account to create a Sybase Web account if you do not have one already.

4. From the list of available downloads, select the International Resources Deployment Kit that matches the platform and version of Adaptive Server Anywhere that you are currently using.

☞ For a list of the languages available in Adaptive Server Anywhere, see "Choosing collations" [*ASA Database Administration Guide,* page 335].

Identifying other file types

The following table identifies the platform and function of Adaptive Server Anywhere files according to their file extension. Adaptive Server Anywhere follows standard file extension conventions where possible.

File extension	Platform	File type
.nlm	Novell NetWare	NetWare Loadable Module
.chm, .chw	Windows	Help system file
.lib	Varies by development tool	Static runtime libraries for the creation of embedded SQL executables
.cfg, .cpr, .dat, .loc, .spr, .srt, .xlt	Windows	Sybase Adaptive Server Enterprise components
.cmd .bat	Windows	Command files

File extension	Platform	File type
.res	NetWare, UNIX	Language resource file for non-Windows environments
.dll	Windows	Dynamic Link Library
.so .sl .a	UNIX	Shared object (Sun Solaris and IBM AIX) or shared library (HP-UX) file. The equivalent of a Windows DLL.

Database file names

Adaptive Server Anywhere databases are composed of two elements:

♦ **Database file** This is used to store information in an organized format. This file uses a *.db* file extension.

♦ **Transaction log file** This is used to record all changes made to data stored in the database file. This file uses a *.log* file extension, and is generated by Adaptive Server Anywhere if no such file exists and a log file is specified to be used. A mirrored transaction log has the default extension of *.mlg*.

♦ **Write file** If your application uses a write file, it typically has a *.wrt* file extension.

♦ **Compressed database file** If you supply a read-only compressed database file, it typically has extension *.cdb*.

These files are updated, maintained and managed by the Adaptive Server Anywhere relational database-management system.

Using InstallShield for deployment

InstallShield has used a variety of ways for including install components into an InstallShield project:

♦ If you are using InstallShield 7 or later, you can use SQL Anywhere Studio InstallShield Merge Module Projects. These projects can be found in the *deployment\MergeModules* subdirectory of your SQL Anywhere Studio installation. Merge Module Projects allow you to generate merge modules that you can use to deploy the software currently installed on your machine.

♦ If you are using InstallShield 6 and later, you can use SQL Anywhere Studio InstallShield Object Projects. The projects for deploying clients, personal database servers, network servers, and administration tools are found in the *deployment\Object* directory under your SQL Anywhere directory. Object Projects allow you to generate objects that you can use to deploy the software currently installed on your machine.

♦ If you are using InstallShield Professional 5.5 and later, you can use SQL Anywhere Studio InstallShield Template Projects to ease the deployment workload. Templates for deploying a network server, personal server, client interfaces, and administration tools can be found in the *SQL Anywhere 9\deployment\Templates* folder.

When building the media, you may see warnings about empty file groups. These warnings are caused by empty file groups which have been added to the templates as placeholders for your application's files. To remove these warnings, you can either add your application's files to the file groups, or delete or rename the file groups.

If you are using InstallShield 7 or later, the Merge Module Projects are recommended. If you have InstallShield 6, the Object Projects are recommended rather than the templates, as they are more easily incorporated into an install along with other components.

☞ For instructions on incorporating these InstallShield Projects and Templates into your install program, see your InstallShield documentation.

Using a silent installation for deployment

Silent installations run without user input and with no indication to the user that an installation is occurring. On Windows operating systems you can call the Adaptive Server Anywhere InstallShield setup program from your own setup program in such a way that the Adaptive Server Anywhere installation is silent. Silent installs are also used with Microsoft's Systems Management Server (see).

You can use a silent installation for any of the deployment models described in . You can also use a silent installation for deploying MobiLink synchronization servers.

Creating a silent install

The installation options used by a silent installation are obtained from a **response file**. The response file is created by running the Adaptive Server Anywhere *setup* program using the -r option. A silent install is performed by running *setup* using the -s option.

Do not use the browse buttons
When creating a silent install do not use the browse buttons. The recording of the browse buttons is not reliable.

❖ **To create a silent install**

1. (Optional) Remove any existing installations of Adaptive Server Anywhere.

2. Open a system command prompt, and change to the directory containing the install image (including *setup.exe*, *setup.ins*, and so on).

3. Install the software, using Record mode.

   Type the following command:

   ```
   setup -r
   ```

   This command runs the Adaptive Server Anywhere setup program and creates the response file from your selections. The response file is named *setup.iss*, and is located in your *Windows* directory. This file contains the responses you made to the dialog boxes during installation.

   When run in record mode, the installation program does not offer to reboot your operating system, even if a reboot is needed.

4. Install Adaptive Server Anywhere using the options, and settings that you want to be used when you deploy Adaptive Server Anywhere on the

end-user's machine for use with your application. You can override the
paths during the silent install.

Running a silent install

Your own installation program must call the Adaptive Server Anywhere
silent install using the -s option. This section describes how to use a silent
install.

❖ **To use a silent install**

1. Add the command to invoke the Adaptive Server Anywhere silent install
   to your installation procedure.

   If the response file is present in the install image directory, you can run
   the silent install by entering the following command from the directory
   containing the install image:

   ```
   setup -s
   ```

   If the response file is located elsewhere you must specify the response file
   location using the -f1 option. There must be no space between f1 and
   the quotation mark in the following command line.

   ```
   setup -s -f1"c:\winnt\setup.iss"
   ```

   To invoke the install from another InstallShield script you could use the
   following:

   ```
   DoInstall( "ASA_install_image_path\SETUP.INS",
       "-s", WAIT );
   ```

   You can use options to override the choices of paths for both the Adaptive
   Server Anywhere directory and the shared directory:

   ```
   setup TARGET_DIR=dirname SHARED_DIR=shared_dir -s
   ```

   The TARGET_DIR and SHARED_DIR arguments must precede all
   other options.

2. Check whether the target computer needs to reboot.

   Setup creates a file named *silent.log* in the target directory. This file
   contains a single section called **ResponseResult** containing the following
   line:

   ```
   Reboot=value
   ```

   This line indicates whether the target computer needs to be rebooted to
   complete the installation, and has a value of 0 or 1, with the following
   meanings.

♦ **Reboot=0**   No reboot is needed.

♦ **Reboot=1**   The BATCH_INSTALL flag was set during the installation, and the target computer does need to be rebooted. The installation procedure that called the silent install is responsible for checking the Reboot entry and for rebooting the target computer, if necessary.

3. Check that the setup completed properly.

   Setup creates a file named *setup.log* in the directory containing the response file. The log file contains a report on the silent install. The last section of this file is called **ResponseResult**, and contains the following line:

   ```
   ResultCode=value
   ```

   This line indicates whether the installation was successful. A non-zero ResultCode indicates an error occurred during installation. For a description of the error codes, see your InstallShield documentation.

SMS Installation

Microsoft System Management Server (SMS) requires a silent install that does not reboot the target computer. The Adaptive Server Anywhere silent install does not reboot the computer.

Your SMS distribution package should contain the response file, the install image and the *asa9.pdf* package definition file (provided on the Adaptive Server Anywhere CD ROM in the *extras* folder). The setup command in the PDF file contains the following options:

♦ The -s option for a silent install

♦ The -SMS option to indicate that it is being invoked by SMS.

♦ The -m option to generate a MIF file. The MIF file is used by SMS to determine whether the installation was successful.

Deploying client applications

In order to deploy a client application that runs against a network database server, you must provide each end user with the following items:

♦ **Client application**   The application software itself is independent of the database software, and so is not described here.

♦ **Database interface files**   The client application requires the files for the database interface it uses (ODBC, JDBC, embedded SQL, or Open Client).

♦ **Connection information**   Each client application needs database connection information.

The interface files and connection information required varies with the interface your application is using. Each interface is described separately in the following sections.

The simplest way to deploy clients is to use the supplied InstallShield objects. For more information, see "Using InstallShield for deployment" on page 526.

Deploying OLE DB and ADO clients

The simplest way to deploy OLE DB client libraries is to use the InstallShield objects or templates. For information, see "Using InstallShield for deployment" on page 526. If you wish to create your own installation, this section describes the files to deploy to the end users.

Each OLE DB client machine must have the following:

♦ **A working OLE DB installation**   OLE DB files and instructions for their redistribution are available for redistribution from Microsoft Corporation. They are not described in detail here.

♦ **The Adaptive Server Anywhere OLE DB provider**   The following table shows the files needed for a working Adaptive Server Anywhere OLE DB provider. These files should be placed in a single directory. The Adaptive Server Anywhere installation places them all in the operating-system subdirectory of your SQL Anywhere installation directory (for example: *win32*).

Description	Windows	Windows CE
OLE DB driver file	*dboledb9.dll*	*dboledb9.dll*
OLE DB driver file	*dboledba9.dll*	*dboledba9.dll*
Language-resource library	*dblgen9.dll*	*dblgen9.dll*
Connect dialog	*dbcon9.dll*	*N/A*

OLE DB providers require many registry entries. You can make these by self-registering the DLLs using the *regsvr32* utility on Windows or the *regsvrce* utility on Windows CE.

☞ For more information, see "Creating databases for Windows CE" [*ASA Database Administration Guide,* page 341], and "Linking ODBC applications on Windows CE" on page 233.

Deploying ODBC clients

The simplest way to deploy ODBC clients is to use the InstallShield objects or templates. For information, see "Using InstallShield for deployment" on page 526.

Each ODBC client machine must have the following:

♦ **A working ODBC installation**   ODBC files and instructions for their redistribution are available for redistribution from Microsoft Corporation. They are not described in detail here.

Microsoft provides their ODBC Driver Manager for Windows operating systems. SQL Anywhere Studio includes an ODBC Driver Manager for UNIX. There is no ODBC Driver Manager for Windows CE.

ODBC applications can run without the driver manager. On platforms for which an ODBC driver manager is available, this is not recommended.

Update ODBC if needed
The SQL Anywhere Setup program updates old installations of the Microsoft Data Access Components, including ODBC. If you are deploying your own application, you must ensure that the ODBC installation is sufficient for your application.

♦ **The Adaptive Server Anywhere ODBC driver**   This is the file *dbodbc9.dll* together with some additional files.

☞ For more information, see "ODBC driver required files" on page 532.

♦ **Connection information**   The client application must have access to the information needed to connect to the server. This information is typically

included in an ODBC data source.

ODBC driver required files

The following table shows the files needed for a working Adaptive Server Anywhere ODBC driver. These files should be placed in a single directory. The Adaptive Server Anywhere installation places them all in the operating-system subdirectory of your SQL Anywhere installation directory (for example: *win32*).

Description	Windows	Windows CE	UNIX
ODBC driver	*dbodbc9.dll*	*dbodbc9.dll*	*libdbodbc9.so* *libdbtasks9.so*
Language- resource library	*dblgen9.dll*	*dblgen9.dll*	*dblgen9.res*
Connect dialog	*dbcon9.dll*	*N/A*	*N/A*

♦ Your end user must have a working ODBC installation, including the driver manager. Instructions for deploying ODBC are included in the Microsoft ODBC SDK.

♦ The Connect dialog is needed if your end users are to create their own data sources, if they need to enter user IDs and passwords when connecting to the database, or if they need to display the Connect dialog for any other purpose.

♦ For multi-threaded applications on UNIX, use *libdbodbc9_r.so* and *libdbtasks9_r.so.*

Configuring the ODBC driver

In addition to copying the ODBC driver files onto disk, your Setup program must also make a set of registry entries to install the ODBC driver properly.

Windows The Adaptive Server Anywhere Setup program makes changes to the Registry to identify and configure the ODBC driver. If you are building a setup program for your end users, you should make the same settings.

You can use the *regedit* utility to inspect registry entries.

The Adaptive Server Anywhere ODBC driver is identified to the system by a set of registry values in the following registry key:

```
HKEY_LOCAL_MACHINE\
    SOFTWARE\
        ODBC\
            ODBCINST.INI\
                Adaptive Server Anywhere 9.0
```

The values are as follows:

Value name	Value type	Value data
Driver	String	*path\dbodbc9.dll*
Setup	String	*path\dbodbc9.dll*

There is also a registry value in the following key:

```
HKEY_LOCAL_MACHINE\
    SOFTWARE\
        ODBC\
            ODBCINST.INI\
                ODBC Drivers
```

The value is as follows:

Value name	Value type	Value data
Adaptive Server Anywhere 9.0	String	Installed

Third party ODBC drivers If you are using a third-party ODBC driver on an operating system other than Windows, consult the documentation for that driver on how to configure the ODBC driver.

Deploying connection information

ODBC client connection information is generally deployed as an ODBC data source. You can deploy an ODBC data source in one of the following ways:

♦ **Programmatically** Add a data source description to your end-user's Registry or ODBC initialization files.

♦ **Manually** Provide your end-users with instructions, so that they can create an appropriate data source on their own machine.

   You create a data source manually using the ODBC Administrator, from the User DSN tab or the System DSN tab. The Adaptive Server Anywhere ODBC driver displays the configuration dialog for entering settings. Data source settings include the location of the database file, the name of the database server, as well as any start up parameters and other options.

This section provides you with the information you need to know for either approach.

Types of data source There are three kinds of data sources: User data sources, System data sources, and File data sources.

User data source definitions are stored in the part of the registry containing settings for the specific user currently logged on to the system. System data

sources, however, are available to all users and to Windows services, which run regardless of whether a user is logged onto the system or not. Given a correctly configured System data source named MyApp, any user can use that ODBC connection by providing DSN=MyApp in the ODBC connection string.

File data sources are not held in the registry, but are held in a special directory. A connection string must provide a FileDSN connection parameter to use a File data source.

Data source registry entries

Each user data source is identified to the system by registry entries.

You must enter a set of registry values in a particular registry key. For User data sources the key is as follows:

```
HKEY_CURRENT_USER\
    SOFTWARE\
        ODBC\
            ODBC.INI\
                userdatasourcename
```

For System data sources the key is as follows:

```
HKEY_LOCAL_MACHINE\
    SOFTWARE\
        ODBC\
            ODBC.INI\
                systemdatasourcename
```

The key contains a set of registry values, each of which corresponds to a connection parameter. For example, the ASA 9.0 Sample key corresponding to the ASA 9.0 Sample data source contains the following settings:

Value name	Value type	Value data
Autostop	String	Yes
DatabaseFile	String	*Path*\asademo.db
Description	String	Adaptive Server Anywhere Sample Database
Driver	String	*Path*\win32\dbodbc9.dll
PWD	String	sql
Start	String	*Path*\win32\dbeng9.exe -c 8m
UID	String	dba

> **Note**
>
> It is recommended that you include the EngineName parameter in connection strings for deployed applications. This ensures that the application connects to the correct server in the event that a machine is running multiple Adaptive Server Anywhere database servers and can help prevent timing-dependent connection failures

In these entries, *path* is the Adaptive Server Anywhere installation directory.

In addition, you must add the data source to the list of data sources in the registry. For User data sources, you use the following key:

```
HKEY_CURRENT_USER\
    SOFTWARE\
        ODBC\
            ODBC.INI\
                ODBC Data Sources
```

For System data sources, use the following key:

```
HKEY_LOCAL_MACHINE\
    SOFTWARE\
        ODBC\
            ODBC.INI\
                ODBC Data Sources
```

The value associates each data source with an ODBC driver. The value name is the data source name, and the value data is the ODBC driver name. For example, the User data source installed by Adaptive Server Anywhere is named ASA 9.0 Sample, and has the following value:

Value name	Value type	Value data
ASA 9.0 Sample	String	Adaptive Server Anywhere 9.0

> **Caution: ODBC settings are easily viewed**
>
> *User data source configurations can contain sensitive database settings such as a user's ID and password. These settings are stored in the registry in plain text, and can be view using the Windows registry editors regedit.exe or regedt32.exe, which are provided by Microsoft with the operating system. You can choose to encrypt passwords, or require users to enter them on connecting.*

Required and optional connection parameters

You can identify the data source name in an ODBC configuration string in this manner,

```
DSN=userdatasourcename
```

When a DSN parameter is provided in the connection string, the Current

User data source definitions in the Registry are searched, followed by System data sources. File data sources are searched only when FileDSN is provided in the ODBC connection string.

The following table illustrates the implications to the user and developer when a data source exists and is included in the application's connection string as a DSN or FileDSN parameter.

When the data source...	The connection string must also identify...	The user must supply...
Contains the ODBC driver name and location; the name of the database file/server; startup parameters; and the user ID and password.	No additional information	No additional information.
Contains only the name and location of the ODBC driver.	The name of the database file/ server; and, optionally, the user ID and the password.	User ID and password if not provided in the DSN or ODBC connection string.
Does not exist	The name of the ODBC driver to be used, in the following format: `Driver={ODBCdrivername}` Also, the name of the database, the database file or the database server; and, optionally, other connection parameters such as user ID and password.	User ID and password if not provided in the ODBC connection string.

☞ For more information on ODBC connections and configurations, see the following:

♦ "Connecting to a Database" [*ASA Database Administration Guide,* page 37].

♦ The Open Database Connectivity (ODBC) SDK, available from Microsoft.

Deploying embedded SQL clients

The simplest way to deploy embedded SQL clients is to use the InstallShield

objects or templates. For information, see "Using InstallShield for deployment" on page 526.

Deploying embedded SQL clients involves the following:

♦ **Installed files**   Each client machine must have the files required for an Adaptive Server Anywhere embedded SQL client application.

♦ **Connection information**   The client application must have access to the information needed to connect to the server. This information may be included in an ODBC data source.

Installing files for embedded SQL clients

The following table shows which files are needed for embedded SQL clients.

Description	Windows	UNIX
Interface library	*dblib9.dll*	*libdblib9.so, libdbtasks9.so*
Language resource library	*dblgen9.dll*	*dblgen9.res*
Connect dialog	*dbcon9.dll*	*N/A*

Notes

♦ The network ports DLL is not required if the client is working only with the personal database server.

♦ If the client application uses an ODBC data source to hold the connection parameters, your end user must have a working ODBC installation. Instructions for deploying ODBC are included in the Microsoft ODBC SDK.

☞ For more information on deploying ODBC information, see "Deploying ODBC clients" on page 531.

♦ The Connect dialog is needed if your end users will be creating their own data sources, if they will need to enter user IDs and passwords when connecting to the database, or if they need to display the Connect dialog for any other purpose.

♦ For multi-threaded applications on UNIX, use *libdblib9_r.so* and *libdbtasks9_r.so.*

Connection information

You can deploy embedded SQL connection information in one of the following ways:

- ♦ **Manual**   Provide your end-users with instructions for creating an appropriate data source on their machine.

- ♦ **File**   Distribute a file that contains connection information in a format that your application can read.

- ♦ **ODBC data source**   You can use an ODBC data source to hold connection information. In this case, you need a subset of the ODBC redistributable files, available from Microsoft. For details see "Deploying ODBC clients" on page 531.

- ♦ **Hard coded**   You can hard code connection information into your application. This is an inflexible method, which may be limiting, for example when databases are upgraded.

Deploying JDBC clients

In addition to a Java Runtime Environment, each JDBC client requires jConnect or the iAnywhere JDBC driver.

☞ For jConnect documentation, see *http://sybooks.sybase.com/jc.html*.

To deploy the iAnywhere JDBC driver, you must deploy the following files:

- ♦ *jodbc.jar*   This must be in the application's classpath.

- ♦ *dbjodbc9.dll*   This must be in the system path. On UNIX or Linux environments, the file is a shared library (*dbjodbc9.so*).

- ♦ The ODBC driver files. For more information, see "ODBC driver required files" on page 532.

Your Java application needs a URL in order to connect to the database. This URL specifies the driver, the machine to use, and the port on which the database server is listening.

☞ For more information on URLs, see "Supplying a URL for the server" on page 112.

Deploying Open Client applications

In order to deploy Open Client applications, each client machine needs the Sybase Open Client product. You must purchase the Open Client software separately from Sybase. It contains its own installation instructions.

☞ Connection information for Open Client clients is held in the interfaces file. For information on the interfaces file, see the Open Client documentation and "Configuring Open Servers" [*ASA Database Administration Guide,* page 106].

Deploying administration tools

Subject to your license agreement, you can deploy a set of administration tools including Interactive SQL, Sybase Central, and the Adaptive Server Anywhere Console utility.

The simplest way to deploy the administration tools is to use the supplied InstallShield merge modules or objects. For more information, see "Using InstallShield for deployment" on page 526.

Deploying Interactive SQL
If your customer application is running on machines with limited resources, you may want to deploy *dbisqlc.exe* instead of Interactive SQL. However, *dbisqlc* is a limited application. It does not contain all the features of Interactive SQL and compatibility between the two is not guaranteed.

The *dbisqlc* executable requires the standard embedded SQL client-side libraries.

☞ For information on system requirements for administration tools, see "SQL Anywhere Studio Supported Platforms" [*Introducing SQL Anywhere Studio,* page 95].

Deploying administration tools on Windows without InstallShield

This section explains how to install Interactive SQL (dbisql), Sybase Central (including the Adaptive Server Anywhere and MobiLink plug-ins), and the Adaptive Server Anywhere Console utility (dbconsole) on a Windows machine without using InstallShield. It is intended for those who want to create an installer for these administration tools.

This information applies to all Windows platforms except Windows CE. The instructions given here are specific to version 9.0.2 and cannot be applied to earlier or later versions of the software.

Check your license agreement
Redistribution of files is subject to your license agreement. No statements in this document override anything in your license agreement. Check your license agreement before considering deployment.

Before you begin
Before reading this section, you should have an understanding of the Windows registry, including the REGEDIT application.

Modifying your registry is dangerous
Modify your registry at your own risk. It is recommended that you back up your system before modifying the registry.

The following steps are required to deploy the administration tools:

1. Decide what you want to deploy.

2. Copy the required files.

3. Register the administration tools with Windows.

4. Update the system path.

5. Register the plug-ins with Sybase Central.

6. Register the Adaptive Server Anywhere ODBC driver with Windows.

7. Register the online help files with Windows.

Each of these steps is explained in detail in the following sections.

Step 1: Deciding what software to deploy

You can install any combination of the following software bundles:

♦ Interactive SQL

♦ Sybase Central with the Adaptive Server Anywhere plug-in

♦ Sybase Central with the MobiLink plug-in

♦ Adaptive Server Anywhere Console utility (DBConsole)

The following components are required when installing any of the software bundles:

♦ The Java Runtime Environment (JRE) version 1.4.2_02

♦ The jConnect runtime library

♦ The iAnywhere JDBC Driver and the Adaptive Server Anywhere ODBC Driver

The instructions in the following sections are structured so that you can install any (or all) of these four bundles without conflicts.

Step 2: Copying the required files

The administration tools require a specific directory structure. You are free to put the directory tree in any directory, on any drive, but it must have the following layout:

Directory	Description
SA9	The root folder. While this document assumes you are installing into *C:\SA9*, you are free to put the directory anywhere (for example, *C:\Program Files\SA9*).
SA9\java	Holds the program JAR files.
SA9\win32	Holds the native (Win32) components used by the program, including the programs that launch the applications.
SA9\Shared	This tree contains files that are shared among different applications.
SA9\Shared\java	Shared utility libraries.
SA9\Shared\jConnect-5_5	The jConnect 5_5 runtime library.
SA9\Shared\Sun\JavaHelp-1_1	The JavaHelp runtime library.
SA9\Shared\win32	Shared utility libraries.

Itanium 64

This document assumes you are installing on a 32-bit Windows platform. If you are installing the administration tools on a 64-bit Windows XP Itanium machine, you must install the 64-bit versions of the software. Specifically, you must install the 64-bit version of the Java runtime environment and 64-bit versions of all *.exe* and *.dll* files. In general, the 32-bit versions of these files are found in folders called *win32*, while the 64-bit versions are in folders called *ia64*. Files with the *.jar* extension work on both 32- and 64-bit machines.

The following table lists the files required for each of the software bundles. Make a list of the files you need, and then copy them into the directory structure outlined above. In general, you should take the files from an already-installed copy of SQL Anywhere Studio 9.0.2.

File	Adaptive Server Anywhere plug-in	MobiLink plug-in	Interactive SQL	Adaptive Server Anywhere Console
java\jcchart400k.jar	X	X		
java\jlogon.jar	X	X	X	X

File	Adaptive Server Anywhere plug-in	MobiLink plug-in	Interactive SQL	Adaptive Server Anywhere Console
java\jodbc.jar	X	X	X	X
java\sybasecentral.jar	X	X		
java\ASA.jar	X			
java\asaplugin.jar	X			
java\debugger.jar	X			
java\dbconsole.jar				X
java\isql.jar	X		X	
java\mlplugin.jar		X		
java\xerces.jar	X		X	
java\xml4j.jar	X		X	
java\apache_files.txt	X		X	
java\apache_license.txt	X		X	
java\dbmaen9.jar	X	X	X	X
java\dbmaen9.chm	X	X	X	X
java\scjview.exe	X	X		
java\scjview.cls	X	X		
java\scjlgen.dll	X	X		
java\scvwen43.chm	X	X		
java\scvwen43.jar	X	X		
Shared\java\HelpManager11.jar	X	X	X	X
Shared\java\JComponents142.jar	X		X	X
Shared\java\jsyblib142.jar	X	X	X	X
Shared\java\SCEditor142.jar	X	X	X	
Shared\jConnect-5_5\classes\jconn2.jar	X	X	X	X
Shared\Sun\jre142*.* [1]	X	X	X	X

---

[1]The administration tools require JRE 1.4.2_02. You should not substitute a later patch

File	Adaptive Server Anywhere plug-in	MobiLink plug-in	Interactive SQL	Adaptive Server Anywhere Console
Shared\Sun\JavaHelp-1_1\jh.jar	X	X	X	X
Shared\win32\jsyblib142.dll	X	X	X	X
win32\dbcon9.dll	X	X	X	X
win32\dbconsole.exe				X
win32\dbisql.exe			X	
win32\dbisqlg.exe			X	
win32\dbisql.cls			X	
win32\dbjodbc9.dll	X	X	X	X
win32\dblgen9.dll	X		X	X
win32\dblib9.dll	X	X	X	X
win32\dbmlput9.dll		X		
win32\dbodbc9.dll	X	X	X	X
win32\dbput9.dll	X			
win32\dbtool9.dll	X			
c:\windows\system32\keyhh.exe [2]	X	X	X	X

International message and help files

All displayed text and online help for the administration tools is translated from English into French, German, Japanese, and simplified Chinese. The resources for each language are held in separate files. The English files contain **en** in the file names. French files have similar names, but use **fr** instead of **en**. German file names contain **de**, Japanese file names contain **ja**, and Chinese files contain **zh**.

In the preceding procedures, only the English files are listed. If you want to install support for different languages, you have to add the resource and help files for those other languages. The translated files are as follows:

version of the JRE (e.g. 1.4.2_03) unless you have a specific need to do so. Copy the JRE files from an installed copy of Adaptive Server Anywhere 9.0.2 from the *C:\Program Files\Sybase\Shared\Sun\jre142* directory. Copy the entire *jre142* tree, including subdirectories.

[2]The exact name of the Windows system directory differs depending on which operating system you are using.

```
dbmaxx9.jar - SQL Anywhere Help Files
dbmaen9.jar   English
dbmafr9.jar   French
dbmade9.jar   German
dbmaja9.jar   Japanese
dbmazh9.jar   Chinese

dblgxx9.res - Adaptive Server Anywhere Messages Files
dblgen9.res   English
dblgfr9.res   French
dblgde9.res   German
dblgja9.res   Japanese
dblgzh9.res   Chinese

scvwen43.jar - Sybase Central Help Files
scvwen43.jar  English
scvwfr43.jar  French
scvwde43.jar  German
scvwja43.jar  Japanese
scvwzh43.jar  Chinese
```

These files are included with localized versions of SQL Anywhere Studio.

Step 3: Registering the administration tools with Windows

You must set the following registry values for the administration tools.

♦ In *HKLM\SOFTWARE\Sybase\Sybase Central\4.3*

- **Location** The fully-qualified path to the folder containing the Sybase Central files.

- **Shared Location** The fully-qualified path to the *Shared* folder.

- **Language** The two-letter code for the language used by Sybase Central. This must be one of the following: **en** , **fr** , **de** , **ja** , **zh** for English, French, German, Japanese, and Simplified Chinese, respectively.

♦ In *HKLM\SOFTWARE\Sybase\Adaptive Server Anywhere\9.0*

- **Location** The fully-qualified path to the root of the installation folder.

- **Shared Location** The fully-qualified path to the *shared* folder.

Paths should *not* end in a backslash.

Your installer can encapsulate all this information by creating a *.reg* file and then executing it. The following is a sample *.reg* file:

```
Windows Registry Editor Version 5.00

[HKEY_LOCAL_MACHINE\\SOFTWARE\\Sybase\\Sybase Central\\4.3]
Location=C:\\SA9\\java
Shared Location=C:\\SA9\\Shared
Language=EN

[HKEY_LOCAL_MACHINE\SOFTWARE\Sybase\Adaptive Server Anywhere\\
          9.0]
Location=c:\\SA9
Shared Location=c:\\SA9\\shared
```

Backslashes must be escaped by another backslash in a *.reg* file.

Step 4: Updating the system path

To run the administration tools, the directories with *.exe* and *.dll* files must
be included in the path. You must add the *SA9\win32*, *SA9\java*, and
SA9\Shared\win32 directories to the system path.

On Windows NT/2000/XP, the system path is stored in the following registry
key:

```
HKLM\
  SYSTEM\
    CurrentControlSet\
      Control\
        Session Manager\
          Environment\Path.
```

When you deploy Interactive SQL or Sybase Central, add the following
directories to the end of the existing path:

```
C:\SA9\win32;c:\SA9\java;C:\SA9\Shared\win32
```

Step 5: Registering the plug-ins

Sybase Central requires a configuration file that lists the installed plug-ins.
Your installer should create this file.

The file is called *.scRepository*. It resides in the same directory as
scjview.exe. It is a plain text file that contains some basic information about
the plug-ins that Sybase Central should load.

Here is a sample *.scRepository* file that contains information for the
Adaptive Server Anywhere and MobiLink plug-ins. Some entries have been
split across multiple lines for display purposes. In the file, each entry must
be on a single line:

```
SCRepositoryInfo/Version=4
#
Providers/asa90/Name=Adaptive Server Anywhere 9
Providers/asa90/Classpath=C:\\sa9\\java\\asaplugin.jar
Providers/asa90/UseClassloader=true
Providers/asa90/AdditionalClasspath=
C:\\sa9\\Shared\\jConnect-5_5\\classes\\jconn2.jar;
C:\\sa9\\java\\isql.jar;
C:\\SA9\\java\\asa.jar;
C:\\SA9\\shared\\java\\JComponents142.jar;
C:\\SA9\\java\\xerces.jar;
C:\\SA9\\java\\xml4j.jar;
C:\\SA9\\java\\jlogon.jar;
C:\\SA9\\java\\debugger.jar;
C:\\SA9\\java\\jodbc.jar
Providers/asa90/ProviderId=asa90
Providers/asa90/Version=9.0.2.0000
Providers/asa90/Provider=com.sybase.asa.plugin.ASAPlugin

#
Providers/mobilink90/Name=MobiLink Synchronization 9
 Providers/mobilink90/Classpath=C:\\SA9\\java\\mlplugin.jar
 Providers/mobilink90/UseClassloader=true
Providers/mobilink90/AdditionalClasspath=
C:\\SA9\\Shared\\jConnect-5_5\\classes\\jconn2.jar;
C:\\SA9\\java\\asa.jar;
C:\\SA9\\java\\jlogon.jar;
C:\\SA9\\java\\jodbc.jar
 Providers/mobilink90/Version=9.0.2.0000
Providers/mobilink90/ProviderId=mobilink90
Providers/mobilink90/Provider=com.sybase.mobilink.plugin.MLPlugi
        n
```

Notes

♦ The AdditionalClasspath lines must be entered on a single line in the *.scRepository* file.

♦ The first line indicates the version of the *.scRepository* file.

♦ Lines beginning with a # are comments.

♦ The next 7 lines are the information for the Adaptive Server Anywhere plug-in. These are followed by another comment line, and then the information for the MobiLink plug-in. Your installer should write out a file similar to this one. The only changes required are the fully-qualified paths to the JAR files in the Classpath and AdditionalClasspath lines.

♦ The file should only contain information about the plug-ins you are deploying. For example, if you are not deploying the MobiLink plug-in, you should not include the last section of the file (the lines that start Providers/mobilink90).

Step 6: Registering the Adaptive Server Anywhere ODBC driver

You must install the Adaptive Server Anywhere ODBC driver before they can be used by the iAnywhere JDBC Driver in the administration tools.

☞ For more information, see "Configuring the ODBC driver" on page 532.

Step 7: Registering the online help files

The administration tools come with HTML Help files for the online help. These files must be registered with Windows before they can be used. Registering them requires a registry string to be written that names the help file and gives its fully-qualified path.

For all of the administration tools, you need to define a string value called *HKLM\SOFTWARE\Microsoft\Windows\HTML Help\dbmaen9.chm.* Its value should be *C:\SA9\java\dbmaen9.chm.*

For the Sybase Central help file, you must also define a string value called *HKLM\SOFTWARE\Microsoft\Windows\HTML Help\scvwen43.chm.* Its value should be *C:\SA9\java\scvwen43.chm.*

If you are installing translated online help files, they must also be registered. Follow the procedure provided earlier for the English help file, but use the translated file name instead.

The computer must already have the HTML Help viewer installed. It comes with current versions of Windows, but may not be present on some older machines. You can check for its existence by looking for the *hh.exe* file that is installed in the *C:\Windows* or *C:\WINNT* folder.

☞ For more information on deploying databases and database applications, see "Deploying Databases and Applications" on page 519.

Deploying administration tools on Linux and Solaris

This section explains how to install Interactive SQL (dbisql), Sybase Central (including the Adaptive Server Anywhere and MobiLink plug-ins), and the Adaptive Server Anywhere Console utility (dbconsole) on Linux and Solaris machines. It is intended for those who want to create an installer for these administration tools.

The instructions given here are specific to version 9.0.2 and cannot be applied to earlier or later versions of the software.

> **Check your license agreement**
>
> Redistribution of files is subject to your license agreement. No statements in this document override anything in your license agreement. Please check your license agreement before considering deployment.

Before you begin

Before you begin, you must install SQL Anywhere Studio on one machine as a source for program files. This is the **reference installation** for your deployment.

The general steps involved are as follows:

1. Decide which programs you want to deploy.

2. Copy the required files.

3. Set environment variables.

4. Register the Sybase Central plug-ins.

Each of these steps is explained in detail in the following sections.

Step 1: Deciding what you want to deploy

You can install any combination of the following software bundles:

♦ Interactive SQL

♦ Sybase Central with the Adaptive Server Anywhere plug-in

♦ Sybase Central with the MobiLink plug-in

♦ Adaptive Server Anywhere Console utility (DBConsole)

All of these bundles require the following components:

♦ Version 1.4.2_02 of the Java Runtime Environment (JRE)

♦ The jConnect runtime library

♦ The iAnywhere JDBC Driver and the Adaptive Server Anywhere ODBC Driver

The instructions in the next section are structured so that you can install any (or all) of these four bundles without conflicts.

Step 2: Copying the required files

Your installer should copy a subset of the files that are installed by the SQL Anywhere Studio installer. You must keep the same directory structure.

You should preserve the permissions on the files when you copy them from your reference SQL Anywhere Studio installation. In general, all users and groups are allowed to read and execute all files.

Base component files   All four bundles require the files and links listed in this section.

Copy the following files from your reference installation:

```
/opt/sybase/SYBSsa9/java/jodbc.jar
/opt/sybase/SYBSsa9/lib/libdbjodbc9.so.1
/opt/sybase/SYBSsa9/lib/libdbodbc9_r.so.1
/opt/sybase/SYBSsa9/lib/libdbodm9_r.so.1
/opt/sybase/SYBSsa9/lib/libdbtasks9_r.so.1
```

Copy the following directories:

```
/opt/sybase/shared/jre_1.4.2_linux_sun_i386 (Linux)
/opt/sybase/shared/jre_1.4.2_solaris_sun_sparc (Solaris)
/opt/sybase/shared/jConnect-5_5
```

Create the following symbolic links in */opt/SYBSsa9/lib*:

```
libdbjodbc9.so -> libdbjodbc9.so.1
libdbodbc9_r.so -> libdbodbc9_r.so.1
libdbodm9_r.so -> libdbodm9_r.so.1
libdbtasks9_r.so -> libdbtasks9_r.so.1
```

Create the following symbolic links in */opt/Sybase/SYBSsa9*:

```
jre142 -> /opt/sybase/shared/jre_1.4.2_linux_sun_i386 (Linux)
 jre142 -> /opt/sybase/shared/jre_1.4.2_solaris_sun_sparc
         (Solaris)
 shared -> /opt/sybase/shared
```

Interactive SQL files   Copy the following files from your reference installation:

```
/opt/sybase/SYBSsa9/bin/dbisql
/opt/sybase/SYBSsa9/java/dbmaen9.jar
/opt/sybase/SYBSsa9/java/isql.jar
/opt/sybase/SYBSsa9/java/jlogon.jar
/opt/sybase/SYBSsa9/java/xerces.jar
/opt/sybase/SYBSsa9/java/xml4j.jar

/opt/sybase/SYBSsa9/lib/libdblib9_r.so.1
/opt/sybase/SYBSsa9/lib/libdbtool9_r.so.1
/opt/sybase/SYBSsa9/res/asa.cvf
/opt/sybase/SYBSsa9/res/dblgen9.res
/opt/sybase/shared/java/HelpManager11.jar
/opt/sybase/shared/java/JComponents142.jar
/opt/sybase/shared/java/SCEditor142.jar
/opt/sybase/shared/java/jsyblib142.jar
/opt/sybase/shared/sun/javahelp-1_1/jh.jar
```

Create the following symbolic links in */opt/SYBSsa9/lib*:

```
libdblib9_r.so -> libdblib9_r.so.1
libdbtool9_r.so -> libdbtool9_r.so.1
```

Sybase Central with the Adaptive Server Anywhere plug-in

Copy the following files from your reference installation:

```
/opt/sybase/SYBSsa9/bin/dbisql
/opt/sybase/SYBSsa9/java/asa.jar
/opt/sybase/SYBSsa9/java/asaplugin.jar
/opt/sybase/SYBSsa9/java/dbmaen9.jar
/opt/sybase/SYBSsa9/java/debugger.jar
/opt/sybase/SYBSsa9/java/isql.jar
/opt/sybase/SYBSsa9/java/jlogon.jar
/opt/sybase/SYBSsa9/java/jodbc.jar

/opt/sybase/SYBSsa9/java/xerces.jar
/opt/sybase/SYBSsa9/java/xml4j.jar
/opt/sybase/SYBSsa9/lib/libdblib9_r.so.1
/opt/sybase/SYBSsa9/lib/libdbput9_r.so.1
/opt/sybase/SYBSsa9/lib/libdbtool9_r.so.1
/opt/sybase/SYBSsa9/res/asa.cvf
/opt/sybase/SYBSsa9/res/dblgen9.res
/opt/sybase/shared/java/HelpManager11.jar

/opt/sybase/shared/java/JComponents142.jar
/opt/sybase/shared/java/SCEditor142.jar
/opt/sybase/shared/java/jsyblib142.jar
/opt/sybase/shared/sun/javahelp-1_1/jh.jar
/opt/sybase/shared/sybcentral43/jcchart400K.jar
/opt/sybase/shared/sybcentral43/scjview
/opt/sybase/shared/sybcentral43/scvwen43.jar
/opt/sybase/shared/sybcentral43/sybasecentral.jar
```

Create the following symbolic links in */opt/SYBSsa9/lib*:

```
libdblib9_r.so -> libdblib9_r.so.1
libdbput9_r.so -> libdbput9_r.so.1
libdbtool9_r.so -> libdbtool9_r.so.1
```

Create the following links in */opt/sybase/shared/sybcentral43*:

```
jre142 -> /opt/sybase/shared/jre_1.4.2_linux_sun_i386 (Linux)
jre142 -> /opt/sybase/shared/jre_1.4.2_solaris_sun_sparc
        (Solaris)
```

Sybase Central with the MobiLink plug-in

Copy the following files from your reference installation:

```
/opt/sybase/SYBSsa9/java/asa.jar
/opt/sybase/SYBSsa9/java/dbmaen9.jar
/opt/sybase/SYBSsa9/java/jlogon.jar
/opt/sybase/SYBSsa9/java/jodbc.jar
/opt/sybase/SYBSsa9/java/mlplugin.jar
/opt/sybase/SYBSsa9/lib/libdblib9_r.so.1
/opt/sybase/SYBSsa9/lib/libdbmlput9_r.so.1
/opt/sybase/SYBSsa9/lib/libdbtool9_r.so.1
```

```
/opt/sybase/shared/java/HelpManager11.jar
/opt/sybase/shared/java/JComponents142.jar
/opt/sybase/shared/java/SCEditor142.jar
/opt/sybase/shared/java/jsyblib142.jar
/opt/sybase/shared/sun/javahelp-1_1/jh.jar
/opt/sybase/shared/sybcentral43/jcchart400K.jar
/opt/sybase/shared/sybcentral43/scjview
/opt/sybase/shared/sybcentral43/scvwen43.jar
/opt/sybase/shared/sybcentral43/sybasecentral.jar
```

Copy this entire directory:

```
/opt/sybase/SYBSsa9/drivers
```

Create the following symbolic links in */opt/sybase/SYBSsa9/lib*:

```
libdblib9_r.so -> libdblib9_r.so.1
libdbmlput9_r.so -> libdbmlput9_r.so.1
libdbtool9_r.so -> libdbtool9_r.so.1
```

Create the following symbolic links in */opt/sybase/shared/sybcentral43*:

```
jre142 -> /opt/sybase/shared/jre_1.4.2_linux_sun_i386 (Linux)
jre142 -> /opt/sybase/shared/jre_1.4.2_solaris_sun_sparc
         (Solaris)
```

The Adaptive Server Anywhere Console

Copy the following files from your reference installation:

```
/opt/sybase/SYBSsa9/bin/dbconsole
/opt/sybase/SYBSsa9/java/DBConsole.jar
/opt/sybase/SYBSsa9/java/dbmaen9.jar
/opt/sybase/SYBSsa9/java/jlogon.jar
/opt/sybase/SYBSsa9/java/jodbc.jar
/opt/sybase/SYBSsa9/lib/libdblib9_r.so.1
/opt/sybase/SYBSsa9/lib/libdbtool9_r.so.1

/opt/sybase/SYBSsa9/res/asa.cvf
/opt/sybase/SYBSsa9/res/dblgen9.res
/opt/sybase/shared/java/HelpManager11.jar
/opt/sybase/shared/java/JComponents142.jar
/opt/sybase/shared/java/SCEditor142.jar
/opt/sybase/shared/java/jsyblib142.jar
/opt/sybase/shared/sun/javahelp-1_1/jh.jar
```

Create the following symbolic links in */opt/sybase/SYBSsa9/lib*:

```
libdblib9_r.so   -> libdblib9_r.so.1
libdbtool9_r.so  -> libdbtool9_r.so.1
```

International message and help files

All displayed text and online help for the administration tools have been translated from English into French, German, Japanese, and simplified Chinese. The resources for each language are in separate files. The English files contain **en** in the file names. French files have similar names, but use **fr**

551

instead of **en**. German file names contain **de**, Japanese file names contain **ja**, and Chinese files contain **zh**.

In the preceding procedures, only the English files were listed. If you want to install support for different languages, you have to add the resource and help files for those other languages. The translated files are as follows:

```
dbmaxx9.jar - SQL Anywhere Help Files
dbmaen9.jar   English
dbmafr9.jar   French
dbmade9.jar   German
dbmaja9.jar   Japanese
dbmazh9.jar   Chinese

dblgxx9.res - Adaptive Server Anywhere Messages Files
dblgen9.res   English
dblgfr9.res   French
dblgde9.res   German
dblgja9.res   Japanese
dblgzh9.res   Chinese

scvwen43.jar - Sybase Central Help Files
scvwen43.jar  English
scvwfr43.jar  French
scvwde43.jar  German
scvwja43.jar  Japanese
scvwzh43.jar  Chinese
```

These files are included with localized versions of SQL Anywhere Studio.

Step 3: Setting environment variables

To run the administration tools, a number of environment variables must be defined or modified. This is usually done in the *asa_config.sh* file, which is created by the SQL Anywhere Studio installer, but you have the flexibility to do this however is most appropriate for your application.

1. Set the PATH to include the following:

   ```
   /opt/Sybase/SYBsa9/bin
   ```

   If you are deploying Sybase Central, you must also add

   ```
   /opt/sybase/shared/sybcentral43
   ```

2. Set LD_LIBRARY_PATH to include the following:

   ```
   /opt/sybase/SYBsa9/jre142/lib/i386/client
   /opt/sybase/SYBsa9/jre142/lib/i386
   /opt/sybase/SYBsa9/jre142/lib/i386/native_threads
   ```

   If you are deploying the MobiLink plug-in, you must also include

   ```
   /opt/sybase/SYBsa9/drivers
   ```

3. Set the following environment variables:

```
ASANY9="/opt/sybase/SYBsa9"
ASANYSH="/opt/sybase/shared"
```

Step 4: Register the Sybase Central plug-ins

This step involves the configuration of Sybase Central. If you are not installing Sybase Central, you can skip it.

Sybase Central uses a configuration file that lists installed plug-ins. The file is called *.scRepository*. It resides in the same directory as *scjview* (*/opt/sybase/shared/sybcentral43*).

Your installer must create this file. Note that it contains full paths to a number of JAR files that may change depending on where the software is installed.

The following is a sample *.scRepository* file that contains information for the Adaptive Server Anywhere and MobiLink plug-ins. Some entries have been split across multiple lines for display purposes. When you create a *.scRepository* file, each entry must be entered on a single line.

```
SCRepositoryInfo/Version=4
#
Providers/asa90/Version=9.0.2.0000
Providers/asa90/UseClassloader=true
Providers/asa90/Classpath=\_opt\_sybase\_SYBSsa9\_java\_
          asaplugin.jar
Providers/asa90/Name=Adaptive Server Anywhere 9
Providers/asa90/AdditionalClasspath=
  \_opt\_sybase\_SYBSsa9\_java\_isql.jar:
  \_opt\_sybase\_SYBSsa9\_java\_asa.jar:
  \_opt\_sybase\_SYBSsa9\_java\_xerces.jar:
  \_opt\_sybase\_SYBSsa9\_java\_xml4j.jar:
  \_opt\_sybase\_SYBSsa9\_java\_jlong.jar:
  \_opt\_sybase\_SYBSsa9\_java\_debugger.jar:
  \_opt\_sybase\_SYBSsa9\_java\_jlogon.jar:
  \_opt\_sybase\_SYBSsa9\_java\_jodbc.jar:
  \_opt\_sybase\_shared\_java\_JComponents142.jar:
  \_opt\_sybase\_shared\_jConnect-5_5\_classes\_jconn2.jar
Providers/asa90/Provider=com.sybase.asa.plugin.ASAPlugin
Providers/asa90/ProviderId=asa90
Providers/asa90/InitialLoadOrder=0
```

```
#
Providers/mobilink90/Version=9.0.2.0000
Providers/mobilink90/UseClassloader=true
Providers/mobilink90/Classpath=\_opt\_sybase\_SYBSsa9\_java\_
        mlplugin.jar
Providers/mobilink90/Name=MobiLink Synchronization 9
Providers/mobilink90/AdditionalClasspath=
  \_opt\_sybase\_SYBSsa9\_java\_asa.jar:
  \_opt\_sybase\_SYBSsa9\_java\_jlogon.jar:
  \_opt\_sybase\_SYBSsa9\_java\_jodbc.jar:
  \_opt\_sybase\_shared\_jConnect-5_5\_classes\_jconn2.jar
Providers/mobilink90/Provider=com.sybase.mobilink.plugin.MLPlugi
        n
Providers/mobilink90/ProviderId=mobilink90
Providers/mobilink90/InitialLoadOrder=0
```

Notes

♦ Your installer should write out a file similar to this one. The only changes required are the fully-qualified paths to the JAR files in the Classpath and AdditionalClasspath lines.

♦ The AdditionalClasspath lines shown above have wrapped to take up additional lines. They must be on a single line in the *.scRepository* file.

♦ Forward slash characters (*/*) are represented with an escape sequence of _ in the *.scRepository* file.

♦ The first line indicates the version of the *.scRepository* file.

♦ The lines beginning with # are comments.

♦ The 8 lines following the first comment line are the information for the Adaptive Server Anywhere plug-in. These are followed by another comment line, and then the information for the MobiLink plug-in.

♦ The file should contain information only about the plug-ins you are deploying. For example, if you are not deploying the MobiLink plug-in, do not supply the last section of the file (the lines that start `Providers/mobilink90`).

For more information about deploying databases and database applications, see "Deploying Databases and Applications" on page 519.

Deploying database servers

You can deploy a database server by making the SQL Anywhere Studio Setup program available to your end-users. By selecting the proper option, each end-user is guaranteed of getting the files they need.

The simplest way to deploy a personal database server or a network database server is to use the supplied InstallShield objects. For more information, see "Using InstallShield for deployment" on page 526.

In order to run a database server, you need to install a set of files. The files are listed in the following table. All redistribution of these files is governed by the terms of your license agreement. You must confirm whether you have the right to redistribute the database server files before doing so.

Windows	UNIX	NetWare
dbeng9.exe	*dbeng9*	N/A
dbsrv9.exe	*dbsrv9*	*dbsrv9.nlm*
dbserv9.dll	*libdbserv9.so, libdbtasks9_r.so*	N/A
dblgen9.dll	*dblgen9.res*	*dblgen9.res*
dbjava9.dll [1]	*libdbjava9.so* [1]	*dbjava9.nlm* [1]
dbctrs9.dll	N/A	N/A
dbextf.dll [2]	*libdbextf.so* [2]	*dbextf.nlm* [2]
asajdbc.zip [1,3]	*asajdbc.zip* [1,3]	*asajdbc.zip* [1,3]
asajrt12.zip [1,3]	*asajrt12.zip* [1,3]	*asajrt12.zip* [1,3]
classes.zip [1,3]	*classes.zip* [1,3]	*classes.zip* [1,3]
dbmem.vxd [4]	N/A	N/A
dbunic9.dll	*libunic.so*	N/A
asa.cvf	*asa.cvf*	*asa.cvf*
charsets\directory	*charsets/ directory*	N/A

1. Required only if using Java in the database. For databases initialized using JDK 1.1, distribute asajdbc.zip. For databases initialized using JDK 1.2 or JDK 1.3, distribute asajrt13.zip.

2. Required only if using system extended stored procedures and functions

(xp_).

3. Install such that the CLASSPATH environment variable can locate classes in this file.

4. Required on Windows 95/98/Me if using dynamic cache sizing.

When deploying database servers on Windows NT/2000/XP, add the following registry key to ensure that messages written to the Event Log are formatted correctly:

```
HKEY_LOCAL_MACHINE\
    SYSTEM\
        CurrentControlSet\
            Services\
            Eventlog\
        Application\
        ASA 9.0\
            ODBC Data Sources
```

Within this key, add a REG_SZ value named EventMessageFile and assign it the data value of the fully qualified location of *dblgen.dll*, for example, *C:\Program Files\Sybase\SQL Anywhere 9\win32\dblgen9.dll*. The English DLL, *dblgen.dll*, can be specified regardless of the deployment language.

Notes
♦ Depending on your situation, you should choose whether to deploy the personal database server (*dbeng9*) or the network database server (*dbsrv9*).

♦ The Java DLL (*dbjava9.dll*) is required only if the database server is to use the Java in the Database functionality.

♦ The table does not include files needed to run utilities such as *dbbackup*.

☞ For information about deploying utilities, see "Deploying administration tools" on page 539.

♦ The zip files are required only for applications that use Java in the database, and must be installed into a location in the user's CLASSPATH environment variable.

Deploying databases

You deploy a database file by installing the database file onto your end user's disk.

As long as the database server shuts down cleanly, you do not need to deploy a transaction log file with your database file. When your end-user starts running the database, a new transaction log is created.

For SQL Remote applications, the database should be created in a properly synchronized state, in which case no transaction log is needed. You can use the Extraction utility for this purpose.

Deploying databases on read-only media

You can distribute databases on read-only media, such as a CD-ROM, as long as you run them in read-only mode or use a write file.

☞ For more information on running databases in read-only mode, see "-r server option" [*ASA Database Administration Guide,* page 154].

To enable changes to be made to Adaptive Server Anywhere databases distributed on read-only media such as a CD-ROM, you can use a **write file**. The write file records changes made to a read-only database file, and is located on a read/write storage media such as a hard disk.

In this case, the database file is placed on the CD-ROM, while the write file is placed on disk. The connection is made to the write file, which maintains a transaction log file on disk.

☞ For more information on write files, see "Working with write files (deprecated)" [*ASA Database Administration Guide,* page 294].

Deploying embedded database applications

This section provides information on deploying embedded database applications, where the application and the database both reside on the same machine.

An embedded database application includes the following:

♦ **Client application**   This includes the Adaptive Server Anywhere client requirements.

☞ For information on deploying client applications, see "Deploying client applications" on page 530.

♦ **Database server**   The Adaptive Server Anywhere personal database server.

☞ For information on deploying database servers, see "Deploying database servers" on page 555.

♦ **SQL Remote**   If your application uses SQL Remote replication, you must deploy the SQL Remote Message Agent.

♦ **The database**   You must deploy a database file holding the data the application uses.

Deploying personal servers

When you deploy an application that uses the personal server, you need to deploy both the client application components and the database server components.

The language resource library (*dblgen9.dll*) is shared between the client and the server. You need only one copy of this file.

It is recommended that you follow the Adaptive Server Anywhere installation behavior, and install the client and server files in the same directory.

Remember to provide the Java zip files and the Java DLL if your application takes advantage of Java in the Database.

Deploying database utilities

If you need to deploy database utilities (such as *dbbackup.exe*) along with your application, then you need the utility executable together with the following additional files:

Description	Windows	UNIX
Database tools library	*dbtool9.dll*	*libdbtools9.so*, *libdbtasks9.so*
Language resource library	*dblgen9.dll*	*dblgen9.res*
Connect dialog (dbisqlc only)	*dbcon9.dll*	

Notes

♦ The database tools are embedded SQL applications, and you must supply the files required for such applications, as listed in "Deploying embedded SQL clients" on page 536.

♦ For multi-threaded applications on UNIX, use *libdbtools9_r.so* and *libdbtasks9_r.so*.

Deploying SQL Remote

If you are deploying the SQL Remote Message Agent, you need to include the following files:

Description	Windows	UNIX
Message Agent	*dbremote.exe*	*dbremote*
Database tools library	*dbtool9.dll*	*libdbtools9.so*, *libdbtasks9.so*
Language resource library	*dblgen9.dll*	*dblgen9.res*
VIM message link library [1]	*dbvim9.dll*	
SMTP message link library [1]	*dbsmtp9.dll*	
FILE message link library [1]	*dbfile9.dll*	*libdbfile9.so*
FTP message link library [1]	*dbftp9.dll*	
MAPI message link library [1]	*dbmapi9.dll*	
Interface Library	*dblib9.dll*	

1 Only deploy the library for the message link you are using.

It is recommended that you follow the Adaptive Server Anywhere installation behavior, and install the SQL Remote files in the same directory as the Adaptive Server Anywhere files.

For multi-threaded applications on UNIX, use *libdbtools9_r.so* and *libdbtasks9_r.so*.

CHAPTER 19

SQL Preprocessor Error Messages

About this chapter

This chapter presents a list of all SQL preprocessor errors and warnings.

Contents

SQL Preprocessor error messages indexed by error message value

Message value	Message
2601	"subscript value %1 too large" on page 577
2602	"combined pointer and arrays not supported for host types" on page 570
2603	"only one dimensional arrays supported for char type" on page 576
2604	"VARCHAR type must have a length" on page 569
2605	"arrays of VARCHAR not supported" on page 569
2606	"VARCHAR host variables cannot be pointers" on page 569
2607	"initializer not allowed on VARCHAR host variable" on page 573
2608	"FIXCHAR type must have a length" on page 567
2609	"arrays of FIXCHAR not supported" on page 569
2610	"arrays of this type not supported" on page 570
2611	"precision must be specified for decimal type" on page 577
2612	"arrays of decimal not allowed" on page 570
2613	"Unknown hostvar type" on page 569
2614	"invalid integer" on page 574
2615	"'%1' host variable must be a C string type" on page 566
2617	"'%1' symbol already defined" on page 566
2618	"invalid type for SQL statement variable" on page 575
2619	"Cannot find include file '%1'" on page 566
2620	"host variable '%1' is unknown" on page 572
2621	"indicator variable '%1' is unknown" on page 573

Message value	Message
2622	"invalid type for indicator variable '%1'" on page 575
2623	"invalid host variable type on '%1'" on page 574
2625	"host variable '%1' has two different definitions" on page 572
2626	"statement '%1' not previously prepared" on page 577
2627	"cursor '%1' not previously declared" on page 570
2628	"unknown statement '%1'" on page 578
2629	"host variables not allowed for this cursor" on page 572
2630	"host variables specified twice - on declare and open" on page 572
2631	"must specify a host list or using clause on %1" on page 575
2633	"no INTO clause on SELECT statement" on page 576
2634	"incorrect SQL language usage – that is a '%1' extension" on page 573
2635	"incorrect Embedded SQL language usage – that is a '%1' extension" on page 573
2636	"incorrect Embedded SQL syntax" on page 573
2637	"missing ending quote of string" on page 575
2639	"token too long" on page 578
2640	"'%1' host variable must be an integer type" on page 566
2641	"must specify an SQLDA on a DESCRIBE" on page 576
2642	"Two SQLDAs specified of the same type (INTO or USING)" on page 568
2646	"cannot describe static cursors" on page 570
2647	"Macros cannot be redefined" on page 568

Message value	Message
2648	"Invalid array dimension" on page 568
2649	"invalid descriptor index" on page 574
2650	"invalid field for SET DESCRIPTOR" on page 574
2651	"field used more than once in SET DESCRIPTOR statement" on page 571
2652	"data value must be a host variable" on page 571
2660	"Into clause not allowed on declare cursor - ignored" on page 568
2661	"unrecognized SQL syntax" on page 579
2662	"unknown SQL function '%1'" on page 578
2663	"wrong number of parameters to SQL function '%1'" on page 579
2664	"static statement names will not work properly if used by 2 threads" on page 577
2665	"host variable '%1' has been redefined" on page 572
2666	"vendor extension" on page 579
2667	"intermediate SQL feature" on page 574
2668	"full SQL feature" on page 571
2669	"transact SQL extension" on page 578
2680	"no declare section and no INCLUDE SQLCA statement" on page 576
2681	"unable to open temporary file" on page 578
2682	"error reading temporary file" on page 571
2683	"error writing output file" on page 571
2690	"Inconsistent number of host variables for this cursor" on page 568
2691	"Inconsistent host variable types for this cursor" on page 567
2692	"Inconsistent indicator variables for this cursor" on page 567

Message value	Message
2693	"Feature not available with UltraLite" on page 567
2694	"no OPEN for cursor '%1'" on page 576
2695	"no FETCH or PUT for cursor '%1'" on page 576
2696	"Host variable '%1' is in use more than once with different indicators" on page 567
2697	"long binary/long varchar size limit is 65 535 for UltraLite" on page 575

SQLPP errors

This section lists messages generated by the SQL preprocessor. The messages may be errors or warnings, or either depending on which command-line options are set.

☞ For more information about the SQL Preprocessor and its command-line options, see "The SQL preprocessor" on page 203.

'%1' host variable must be a C string type

Message value	Message Type
2615	Error

Probable cause   A C string was required in an embedded SQL statement (for a cursor name, option name etc.) and the value supplied was not a C string.

'%1' host variable must be an integer type

Message value	Message Type
2640	Error

Probable cause   You have used a host variable that is not of integer type in a statement where only an integer type host variable is allowed.

'%1' symbol already defined

Message value	Message Type
2617	Error

Probable cause   You defined a host variable twice.

Cannot find include file '%1'

Message value	Message Type
2619	Error

Probable cause   The specified include file was not found. Note that the preprocessor will use the INCLUDE environment variable to search for include files.

FIXCHAR type must have a length

Message value	Message Type
2608	Error

Probable cause   You have used the DECL_FIXCHAR macro to declare a host variable of type FIXCHAR but have not specified a length.

Feature not available with UltraLite

Message value	Message Type
2693	Flag (warning or error)

Probable cause   You have used a feature that is not supported by UltraLite.

Host variable '%1' is in use more than once with different indicators

Message value	Message Type
2696	Error

Probable cause   You have used the same host variable multiple times with different indicator variables in the same statement. This is not supported.

Inconsistent host variable types for this cursor

Message value	Message Type
2691	Error

Probable cause   You have used a host variable with a different type or length than the type or length previously used with the cursor. Host variable types must be consistent for the cursor.

Inconsistent indicator variables for this cursor

Message value	Message Type
2692	Error

Probable cause   You have used an indicator variable when one was not previously used with the cursor, or you have not used an indicator variable when one was previously used with the cursor. Indicator variable usage must be consistent for the cursor.

Inconsistent number of host variables for this cursor

Message value	Message Type
2690	Error

Probable cause   You have used a different number of host variables than the number previously used with the cursor. The number of host variables must be consistent for the cursor.

Into clause not allowed on declare cursor - ignored

Message value	Message Type
2660	Warning

Probable cause   You have used an INTO clause on a DECLARE CURSOR statement. The INTO clause will be ignored.

Invalid array dimension

Message value	Message Type
2648	Error

Probable cause   The array dimension of the variable is negative.

Macros cannot be redefined

Message value	Message Type
2647	Error

Probable cause   A preprocessor macro has been defined twice, possibly in a header file.

Two SQLDAs specified of the same type (INTO or USING)

Message value	Message Type
2642	Error

Probable cause   You have specified two INTO DESCRIPTOR or two USING DESCRIPTOR clauses for this statement.

Unknown hostvar type

Message value	Message Type
2613	Error

Probable cause   You declared a host variable of a type not understood by the SQL preprocessor.

VARCHAR host variables cannot be pointers

Message value	Message Type
2606	Error

Probable cause   You have attempted to declare a host variable as a pointer to a VARCHAR or BINARY. This is not a legal host variable type.

VARCHAR type must have a length

Message value	Message Type
2604	Error

Probable cause   You have attempted to declare a VARCHAR or BINARY host variable using the DECL_VARCHAR or DECL_BINARY macro but have not specified a size for the array.

arrays of FIXCHAR not supported

Message value	Message Type
2609	Error

Probable cause   You have attempted to declare a host variable as an array of FIXCHAR arrays. This is not a legal host variable type.

arrays of VARCHAR not supported

Message value	Message Type
2605	Error

Probable cause   You have attempted to declare a host variable as an array of VARCHAR or BINARY. This is not a legal host variable type.

arrays of decimal not allowed

Message value	Message Type
2612	Error

Probable cause   You have attempted to declare a host variable as an array of DECIMAL. A decimal array is not a legal host variable type.

arrays of this type not supported

Message value	Message Type
2610	Error

Probable cause   You have attempted to declare a host variable array of a type that is not supported.

cannot describe static cursors

Message value	Message Type
2646	Error

Probable cause   You have described a static cursor. When describing a cursor, the cursor name must be specified in a host variable.

combined pointer and arrays not supported for host types

Message value	Message Type
2602	Error

Probable cause   You have used an array of pointers as a host variable. This is not legal.

cursor '%1' not previously declared

Message value	Message Type
2627	Error

Probable cause   An embedded SQL cursor name has been used (in a FETCH, OPEN, CLOSE etc.) without first being declared.

data value must be a host variable

Message value	Message Type
2652	Error

Probable cause   The variable used in the SET DESCRIPTOR statement hasn't been declared as a host variable.

error reading temporary file

Message value	Message Type
2682	Error

Probable cause   An error occurred while reading from a temporary file.

error writing output file

Message value	Message Type
2683	Error

Probable cause   An error occurred while writing to the output file.

field used more than once in SET DESCRIPTOR statement

Message value	Message Type
2651	Error

Probable cause   The same keyword has been used more than once inside a single SET DESCRIPTOR statement.

full SQL feature

Message value	Message Type
2668	Flag (warning or error)

Probable cause   You have used a full-SQL/92 feature and preprocessed with the -ee, -ei, -we or -wi flagging switch.

host variable '%1' has been redefined

Message value	Message Type
2665	Warning

Probable cause   You have redefined the same host variable with a different host type. As far as the preprocessor is concerned, host variables are global; two host variables with different types cannot have the same name.

host variable '%1' has two different definitions

Message value	Message Type
2625	Error

Probable cause   The same host variable name was defined with two different types within the same module. Note that host variable names are global to a C module.

host variable '%1' is unknown

Message value	Message Type
2620	Error

Probable cause   You have used a host variable in a statement and that host variable has not been declared in a declare section.

host variables not allowed for this cursor

Message value	Message Type
2629	Error

Probable cause   Host variables are not allowed on the declare statement for the specified cursor. If the cursor name is provided through a host variable, then you should use full dynamic SQL and prepare the statement. A prepared statement may have host variables in it.

host variables specified twice - on declare and open

Message value	Message Type
2630	Error

Probable cause   You have specified host variables for a cursor on both the declare and the

open statements. In the static case, you should specify the host variables on the declare statement. In the dynamic case, specify them on the open.

incorrect Embedded SQL language usage – that is a '%1' extension

Message value	Message Type
2635	Error

incorrect Embedded SQL syntax

Message value	Message Type
2636	Error

Probable cause   An embedded SQL specific statement (OPEN, DECLARE, FETCH etc.) has a syntax error.

incorrect SQL language usage – that is a '%1' extension

Message value	Message Type
2634	Error

indicator variable '%1' is unknown

Message value	Message Type
2621	Error

Probable cause   You have used a indicator variable in a statement and that indicator variable has not been declared in a declare section.

initializer not allowed on VARCHAR host variable

Message value	Message Type
2607	Error

Probable cause   You can not specify a C variable initializer for a host variable of type VARCHAR or BINARY. You must initialize this variable in regular C executable code.

intermediate SQL feature

Message value	Message Type
2667	Flag (warning or error)

Probable cause   You have used an intermediate-SQL/92 feature and preprocessed with the -ee or -we flagging switch.

invalid descriptor index

Message value	Message Type
2649	Error

Probable cause   You have allocated less than one variable with the ALLOCATE DESCRIPTOR statement.

invalid field for SET DESCRIPTOR

Message value	Message Type
2650	Error

Probable cause   An invalid or unknown keyword is present in a SET DESCRIPTOR statement. The keywords can only be TYPE, PRECISION, SCALE, LENGTH, INDICATOR, or DATA.

invalid host variable type on '%1'

Message value	Message Type
2623	Error

Probable cause   You have used a host variable that is not a string type in a place where the preprocessor was expecting a host variable of a string type.

invalid integer

Message value	Message Type
2614	Error

Probable cause   An integer was required in an embedded SQL statement (for a fetch offset, or a host variable array index, etc.) and the preprocessor was unable to convert what was supplied into an integer.

invalid type for indicator variable '%1'

Message value	Message Type
2622	Error

Probable cause   Indicator variables must be of type short int. You have used a variable of a different type as an indicator variable.

invalid type for SQL statement variable

Message value	Message Type
2618	Error

Probable cause   A host variable used as a statement identifier should be of type a_sql_statement_number. You attempted to use a host variable of some other type as a statement identifier.

long binary/long varchar size limit is 65 535 for UltraLite

Message value	Message Type
2697	Error

Probable cause   When using DECL_LONGBINARY or DECL_LONGVARCHAR with UltraLite, the maximum size for the array is 64kb.

missing ending quote of string

Message value	Message Type
2637	Error

Probable cause   You have specified a string constant in an embedded SQL statement, but there is no ending quote before the end of line or end of file.

must specify a host list or using clause on %1

Message value	Message Type
2631	Error

Probable cause   The specified statement requires host variables to be specified either in a host variable list or from an SQLDA.

must specify an SQLDA on a DESCRIBE

Message value	Message Type
2641	Error

no FETCH or PUT for cursor '%1'

Message value	Message Type
2695	Error

Probable cause   A cursor is declared and opened, but is never used.

no INTO clause on SELECT statement

Message value	Message Type
2633	Error

Probable cause   You specified an embedded static SELECT statement but you did not specify an INTO clause for the results.

no OPEN for cursor '%1'

Message value	Message Type
2694	Error

Probable cause   A cursor is declared, and possibly used, but is never opened.

no declare section and no INCLUDE SQLCA statement

Message value	Message Type
2680	Error

Probable cause   The EXEC SQL INCLUDE SQLCA statement is missing from the source file.

only one dimensional arrays supported for char type

Message value	Message Type
2603	Error

Probable cause   You have attempted to declare a host variable as an array of character arrays. This is not a legal host variable type.

precision must be specified for decimal type

Message value	Message Type
2611	Error

Probable cause   You must specify the precision when declaring a packed decimal host variable using the DECL_DECIMAL macro. The scale is optional.

statement '%1' not previously prepared

Message value	Message Type
2626	Error

Probable cause   An embedded SQL statement name has been used (EXECUTE) without first being prepared.

static statement names will not work properly if used by 2 threads

Message value	Message Type
2664	Warning

Probable cause   You have used a static statement name and preprocessed with the -r re-entrancy switch. Static statement names cause static variables to be generated that are filled in by the database. If two threads use the same statement, contention arises over this variable. Use a local host variable as the statement identifier instead of a static name.

subscript value %1 too large

Message value	Message Type
2601	Error

Probable cause   You have attempted to index a host variable that is an array with a value too large for the array.

token too long

Message value	Message Type
2639	Error

Probable cause The SQL preprocessor has a maximum token length of 2K. Any token longer than 2K will produce this error. For constant strings in embedded SQL commands (the main place this error shows up) use string concatenation to make a longer string.

transact SQL extension

Message value	Message Type
2669	Flag (warning or error)

Probable cause You have used a Sybase Transact SQL feature that is not defined by SQL/92 and preprocessed with the -ee, -ei, -ef, -we, -wi or -wf flagging switch.

unable to open temporary file

Message value	Message Type
2681	Error

Probable cause An error occurred while attempting to open a temporary file.

unknown SQL function '%1'

Message value	Message Type
2662	Warning

Probable cause You have used a SQL function that is unknown to the preprocessor and will probably cause an error when the statement is sent to the database engine.

unknown statement '%1'

Message value	Message Type
2628	Error

Probable cause You attempted to drop an embedded SQL statement that doesn't exist.

unrecognized SQL syntax

Message value	Message Type
2661	Warning

Probable cause   You have used a SQL statement that will probably cause a syntax error when the statement is sent to the database engine.

vendor extension

Message value	Message Type
2666	Flag (warning or error)

Probable cause   You have used an Adaptive Server Anywhere feature that is not defined by SQL/92 and preprocessed with the -ee, -ei, -ef, -we, -wi or -wf flagging switch.

wrong number of parameters to SQL function '%1'

Message value	Message Type
2663	Warning

Probable cause   You have used a SQL function with the wrong number of parameters. This will likely cause an error when the statement is sent to the database engine.

Index

A

K

L

N

name SQLDA field

P

T

W

Z