Adabas SQL Gateway Embedded SQL

Table of Contents

Chapter 1 - Adabas SQL Gateway Embedded SQL

Introduction to Adabas SQL Gateway Embedded SQL

Adabas SQL Gateway Embedded SQL is a precompiler that enables any application written in C/C++ or COBOL and hosted on Windows, MVS (Started Task), Linux x86/390, Sun/Solaris, HPUX, or AIX to access any data source that CONNX supports. The precompiler takes embedded SQL commands and expands them into native language protocol statements that the CONNX Remote Call Interface (RCI) can understand from within the application source code. Once it is compiled and linked with the RCI, the application can communicate with the CONNX JDBC Server running on any Windows platform.

The CONNX JDBC Server will be available on most UNIX platforms in upcoming versions of CONNX.

Getting Started

Before starting, verify that the following items appear in your ../../Precompiler install directory:

Folder Name	Subfolder Name	File Name	Description
GUI		instrci.exe	Installation GUI for Unix and Mainframe systems
Mainframe	**FTP/MVS/Started Task**	ace3gl.obj	Object file necessary for linking MVS objects that access the Precompiler run-time.
		acedsa	Object file necessary for completion of the Precompiler binary on MVS.
		aceint	Pre-linked ACEINT utility for MVS.
		acepcc	Pre-linked C/C++ Precompiler for MVS.
		acepccob	Pre-linked COBOL Precompiler for MVS
		ftp.dat	FTP script for MVS install
		rciclnt	Prelinked precompiler run-time.
	JCL/MVS/Started Task	aceint	JCL procedure to run the ACEINT utility.
		cmempcob	JCL to compiler, link, and run the C example.
		lnace	JCL to compile, link, and run the COBOL example.
		pcacecc	JCL to link the Precompilers.
		pcacecob	JCL procedure to run the COBOL Precompiler.
		rnacecc	Sample JCL to run the ACEINT utility.

		rnacecob	JCL to precompile the COBOL sample source.
		rnaceint	JCL to precompile the C sample source.
Unix		aix.5_32_rci.tar.z	AIX 32-bit precompiler and examples.
		aix.5_64_rci.tar.z	AIX 64-bit precompiler and examples.
		ftp.dat	FTP script for UNIX platforms.
		hpux11_32_rci.tar.z	HPUX 32-bit precompiler and examples.
		lnx86_32_rci.tar.z	Linux 32-bit precompiler and examples.
		lnx390_32_rci.tar.z	Linux 32-bit precompiler and examples.
		sunos_32_rci.tar.z	Sun/Solaris 32-bit precompiler and examples.
		sunos_64_rci.tar.z	Sun/Solaris 64-bit precompiler and examples.
Windows		_employees.c	Simple single-threaded example in C.
		_multithread.c	Multithread example in C.
		aceint.exe	ACEINT utility.
		acepcc.exe	Precompiler for C/C++.
		acepccob.exe	Precompiler for COBOL
		build_test.bat	Batch file that precompiles, builds and runs the example.
		compile.bat	Batch file that compiles the example.
		employees.ddl	DDL table cluster

			descriptions for the Employees table.
		employees.fdt	FDT table description for the Employees table.
		pre_compile.bat	Batch file that precompiles the example.
		rciclnt.lib	LIB file for the Remote Call Interface.
		readme.txt	Description of how to get started with Adabas SQL Gateway Embedded SQL in text file format.
		SQLGatewayEmbeddedSQL.chm	This Windows help file.

The following files are installed in your SYSTEM32 directory:

RCICLNT.dll	The Remote Call Interface.

Windows Installation

Quick-Start Guide

This section describes how to precompile, build, and execute the Employees.exe example; contains a description of the preliminary requirements; and explains how to run the example; for the Windows platform.

These are the minimum preliminary requirements:

- Install the latest version of the Adabas SQL Gateway.
- Create an Adabas database with a sample EMPLOYEES table that is identical in content to the default sample Adabas EMPLOYEES table. The table may be created within ADADBA using the Employees.fdt provided in the PRECOMPILER/WINDOWS install directory.
- Install either Visual Studio 6 or 7 and make sure it is available.

To quick-start the Adabas SQL Gateway Embedded SQL Employees.exe example file on a Windows machine:

1. Create a new CDD using the DDL import option and the Employees.ddl provided in the PRECOMPILER/WINDOWS install directory or use an existing CDD containing the EMPLOYEES Adabas sample table.
2. Verify that the CDD security allows Read/Write Queries.
3. Create a new DSN by click **Start,** pointing to **Programs,** pointing to **CONNX,** and then clicking on the **DSNRegistry** tool.
   Click the **Add** button to create a new DSN.

   For more information, see "To add a new data source name for the JDBC driver" found in the Adabas SQL Gateway help file, available online and within the Adabas SQL Gateway product.
4. Edit the _Employees.c source file so that all connection information is correct.
5. Set the following environment variables within the build_test.bat file:

```
ADADIR = <Root of your Adabas install>
ADAVERS = <version of your Adabas installation, for example, V331>
SAG_COMMON = <Path to the Software AG common DLL repository>
VS_PATH = <Path to bin directory of Visual Studio>
```

4. Inspect the pre_compile.bat and compile.bat files and edit if necessary to ensure that the sample will build in your environment.

5. Start the CONNX JDBC Server as described in "Starting the JDBC Server" in the Adabas SQL Gateway help file.

6. Start the Adabas database.

7. Type the following at the command line, to precompile, build, and run the example:

```
build_test
```

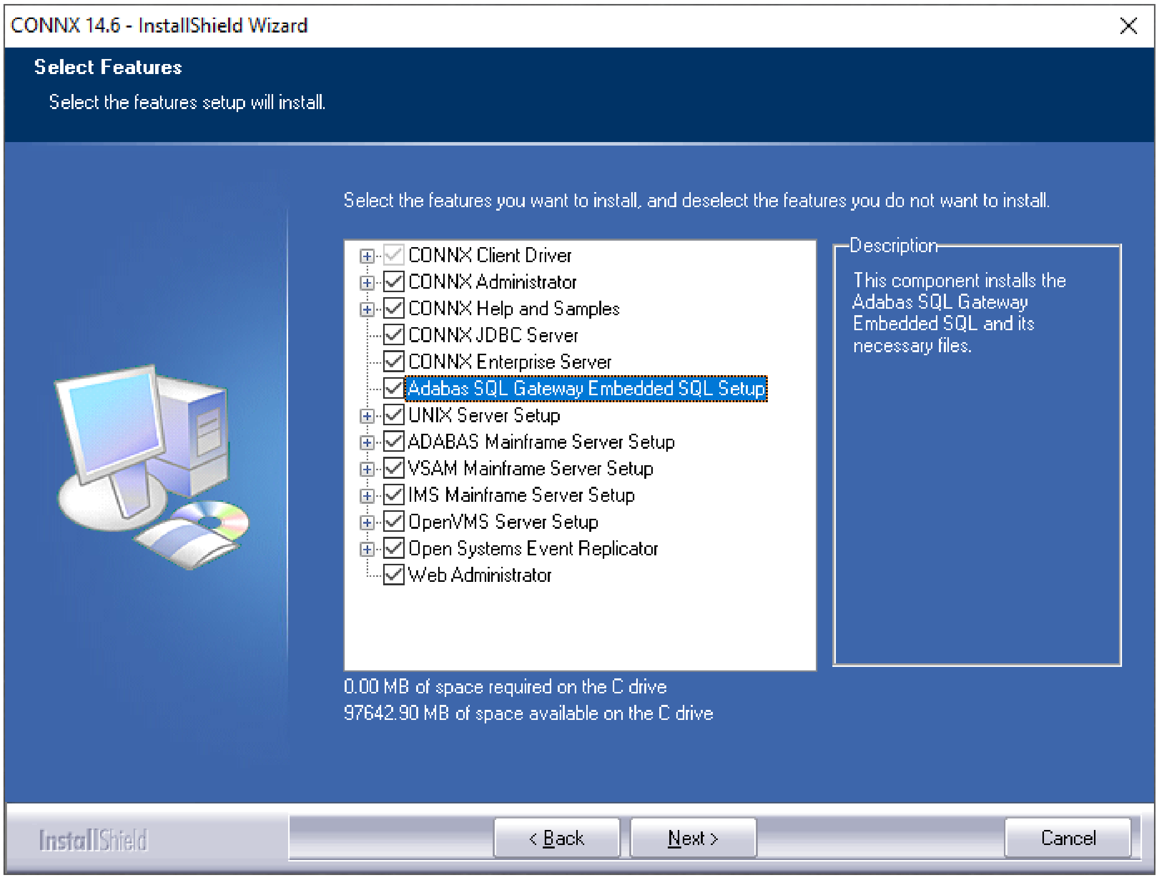
Important: *The example writes to and deletes from the database. If all of the tests that do write/delete succeed, then the database should have returned to its original state.*

z/OS Installation

Installing Adabas SQL Gateway Embedded SQL - z/OS

1. Select a computer on which the CONNX Server Installation component is installed. Click the **Start** button, and then point to **All Programs** and then to **CONNX Driver** and then click **Adabas SQL Gateway Embedded SQL Setup.**

   If you selected the Adabas SQL Gateway or the Adabas Mainframe Server Components, and the Adabas SQL Gateway Embedded SQL option in the Select Components or Database Modules dialog boxes during installation, the **Adabas SQL Gateway Embedded SQL Setup** dialog box appears during installation.



CONNX 14.6 - InstallShield Wizard
Select Features
Select the features setup will install.
InstallShield
< Back
Next >
Cancel

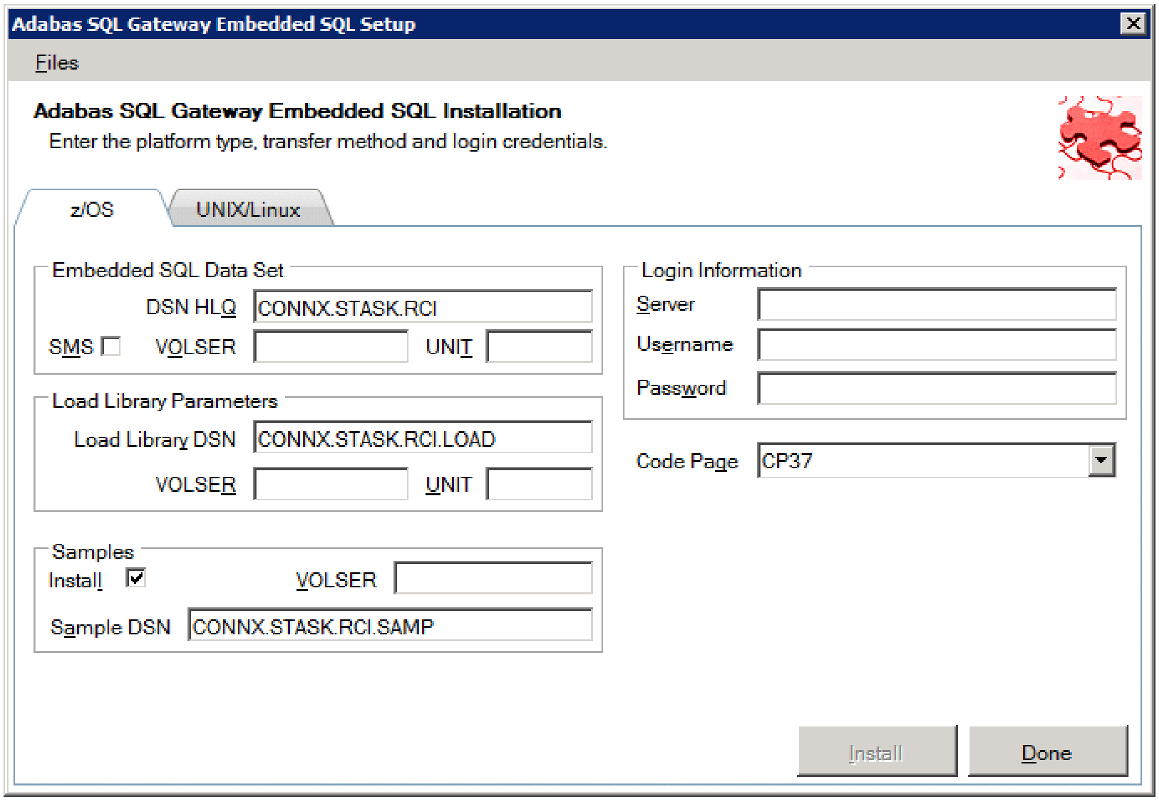
Select the features you want to install, and deselect the features you do not want to install.

- ☑ CONNX Client Driver
- ☑ CONNX Administrator
- ☑ CONNX Help and Samples
- ☑ CONNX JDBC Server
- ☑ CONNX Enterprise Server
- ☑ Adabas SQL Gateway Embedded SQL Setup
- ☑ UNIX Server Setup
- ☑ ADABAS Mainframe Server Setup
- ☑ VSAM Mainframe Server Setup
- ☑ IMS Mainframe Server Setup
- ☑ OpenVMS Server Setup
- ☑ Open Systems Event Replicator
- ☑ Web Administrator

Description

This component installs the Adabas SQL Gateway Embedded SQL and its necessary files.

0.00 MB of space required on the C drive
97642.90 MB of space available on the C drive



Adabas SQL Gateway Embedded SQL Setup

Adabas SQL Gateway Embedded SQL Installation

Enter the platform type, transfer method and login credentials.

Embedded SQL Data Set

DSN HLQ: CONNX.STASK.RCI

SMS ☐ VOLSER UNIT

Load Library Parameters

Load Library DSN: CONNX.STASK.RCI.LOAD

VOLSER UNIT

Samples

Install ☑ VOLSER

Sample DSN: CONNX.STASK.RCI.SAMP

Login Information

Server

Username

Password

Code Page: CP37

2. In **Login**, enter the server's TCP/IP symbolic host name or dotted numeric address, a privileged user account name, and a user account password (for example, **MVS1, CONNX,** and **Password).** The password appears as *******.

   Note: All fields are required.

3. Under **Code Page,** enter the code page the zOS system is using**.**
4. See the tables in the following topics for a detailed explanation of the information required under **Required and Optional Fields**.
5. Click the Install button. This action starts an FTP session and copies the selected CONNX components from the client PC to the target host. Copying alert and FTP windows are displayed.
6. Allow several minutes for the file transfer process between the CONNX administrator computer directory C:\CONNX32\Precompiler and the target host.
7. Click the **Close** button. The Precompiler Setup dialog box closes.
8. Once the FTP successfully completes, you are ready to install the selected Adabas SQL Gateway Embedded components on the host. Log on to TSO on the target system. Under ISPF, navigate to the data set identified by the DSN HLQ parameter in **Required and Optional Fields**. . Verify that the following data sets exist:

Dataset	Purpose
CNTL	Location of the installation JCL and Utility procedure JCL
OBJ	Location of Adabas SQL Gateway Embedded SQL object files
SAMP	Location of utility JCL for running the CNTL procedures and for compiling the C and COBOL sample files

	[Available only if Samples were selected for installation]

9.
10. The SAMP data set contains useful utility JCL:

Member	Purpose
CMEMPCC	Sample JCL to compile, link and run the C example. This JCL is specific to SAS/C.
CMEMPCOB	Sample JCL to compile, link and run the COBOL example.
PCCCSMP	Sample C source with embedded SQL.
PCCOBSMP	Sample COBOL source with embedded SQL.
RNACECC	Sample JCL to run the C/C++ pre-compiler.
RNACECOB	Sample JCL to run the COBOL pre-compiler.
RNACEINT	Sample JCL to run ACEINT with dynamic SQL commands.

11.
12. Browse the CNTL data set and verify that the following members exist:

Member	Purpose
INST	Installation script
ACEINT	JCL Procedure for executing the ACEINT utility
PCACECC	JCL Procedure for executing the C/C++ Precompiler
PCACECOB	JCL Procedure for executing the COBOL Precompiler

13. Edit the job card for the INST member and submit. The installation is a success if all the return codes are 4 or less .

Required and Optional Fields and Files - z/OS

Required and Optional Fields

Field in Install GUI	Term	Description
Login	**Server**	A symbolic or dotted numeric TCP/IP address, for example: MVS or 123.123.123.123
	User ID	Your 1-8 character VSE logon ID. All characters convert from lowercase to uppercase.
	Password	Your 1-8 character VSE password. All characters convert from lowercase to uppercase.
Code Page	Code Page the zOS system is using	
Precompiler Data Set	**DSN HLQ**	A one- or multi-part high-level data set name, which is used to create the CONNX installation data sets on the target host. In the current example, the DSN HLQ is CONNX.STASK.
	Unit	Optional. The DASD device type (3380, 3390, etc.) of the above VOLSER. Omit if the VOLSER field is blank.
	VOLSER	The DASD volume serial on which the CONNX.Adabas sequential and partitioned data sets are created. Optional: The DASD volume serial on which the CONNX.ADABAS sequential and partitioned data sets are created.
Load Library Parameters	**Load Lib. DSN**	Location of the Software AG CICS High Performance Stub Routine. The default for the Adabas SQL Gateway Embedded SQL is CONNX.STASK.LOAD
	Unit	Optional. The DASD device type (3380, 3390, etc.) of the above VOLSER. Omit if the VOLSER field is blank.
	VOLSER	The DASD volume serial on which the new load library is created. Specify a VOLSER and a UNIT to create the PDS on a specific volume, or leave both fields blank to accept the system defaults. If linking into an existing PDS, leave the VOLSER and UNIT text boxes blank.
Samples	**Install**	Select the check box to install the Precompiler sample files.
	VOLSER	The DASD volume serial on which the new CONNX load library is created. Specify a VOLSER and a UNIT to create the PDS on a specific volume, or leave both fields blank to accept the system defaults. If linking into an existing PDS, leave the VOLSER and UNIT text boxes blank.
	Samples DNS	Location of the Adabas SQL Gateway Embedded SQL Sample files. The default location is CONNX.STASK.SAMP.

Required and Optional Files

System codepage files (for advanced users only)

The Adabas SQL Gateway Embedded SQL library default codepage is CP37.

If the **Code Page** dropdoown list contains a better codepage, substitute its value for CP37.

The following codepages are available:

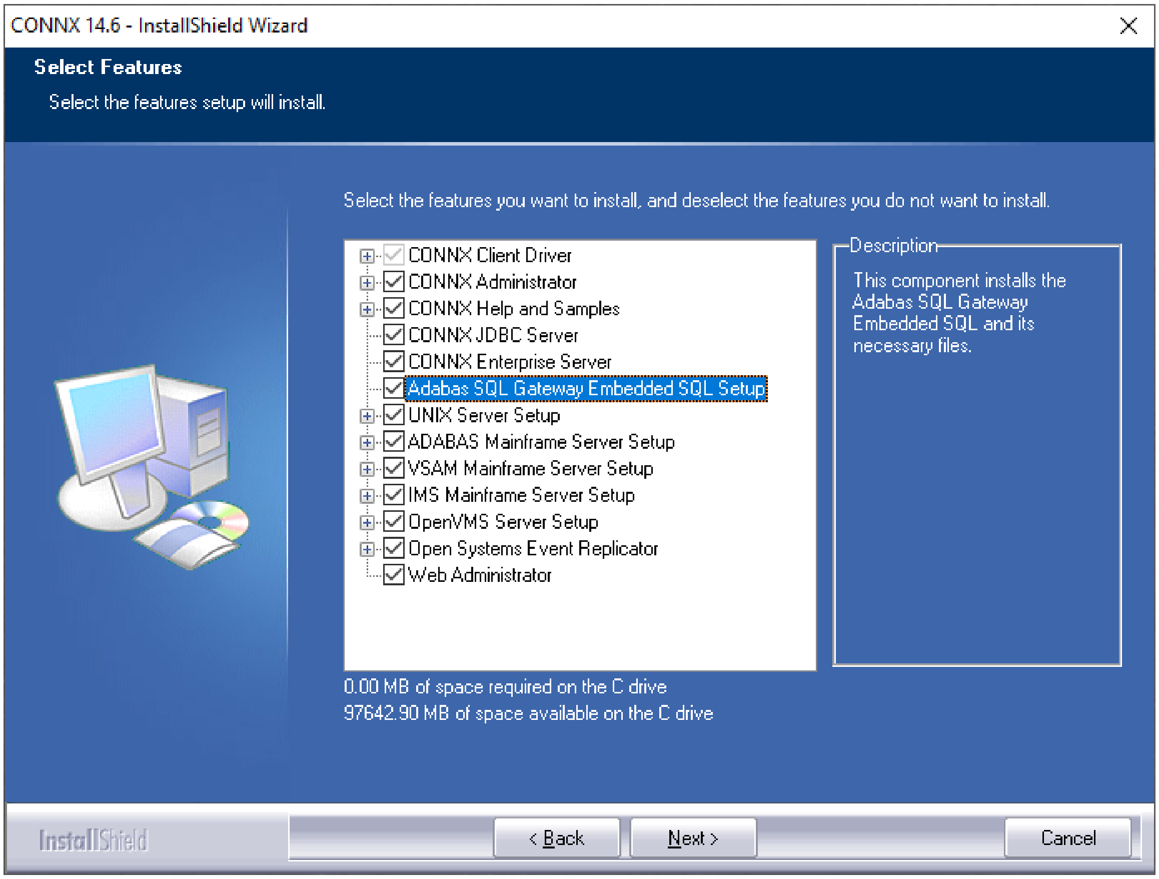
File	Codepage
CP37	IBM273_EBCDIC_EN_US
CP273	IBM273_EBCDIC_DE
CP277	IBM277_EBCDIC_DK_NO
CP278	IBM278_EBCDIC_FI
CP280	IBM280_EBCDIC_IT
CP284	IBM284_EBCDIC_ES
CP285	IBM285_EBCDIC_GB
CP290	IBM290_EBCDIC_JP_Kana
CP297	IBM297_EBCDIC_FR
CP420	IBM420_EBCDIC_Arabic_1
CP424	IBM424_EBCDIC_Hebrew
CP500	IBM500_EBCDIC_International
CP871	IBM871_EBCDIC_Icelandic
CP1140	IBM1140_EBCDIC_EN_US_w_Euro
CP1141	IBM1141_EBCDIC_DE_w_Euro
CP1142	IBM1142_EBCDIC_DK_NO_w_Euro
CP1143	IBM1143_EBCDIC_FI_w_Euro
CP1144	IBM1144_EBCDIC_IT_w_Euro
CP1145	IBM1145_EBCDIC_ES_w_Euro
CP1146	IBM1146_EBCDIC_GB_w_Euro
CP1147	IBM1147_EBCDIC_FR_w_Euro
CP1148	IBM1148_EBCDIC_International_w_Euro
CP1149	IBM1149_EBCDIC_Icelandic_w_Euro
CP1153	IBM1153_EBCDIC_Latin_2_w_Euro
CP1154	IBM1154_EBCDIC_Cyrillic_Multilingual

Unix and Linux Installation

Installing Adabas SQL Gateway Embedded SQL - Linux + Unix

1. Select a computer on which the CONNX Server Installation component is installed. Click the **Start** button, and then point to **All Programs** and then to **CONNX Driver** and then click **Adabas SQL Gateway Embedded SQL Setup.**

   If you selected the Adabas SQL Gateway or the Adabas Mainframe Server Components, and the Adabas SQL Gateway Embedded SQL Setup option in the Select Features dialog box during installation, the **Adabas SQL Gateway Embedded SQL Setup** dialog box appears during installation.



CONNX 14.6 - InstallShield Wizard

Select Features

Select the features setup will install.

Select the features you want to install, and deselect the features you do not want to install.

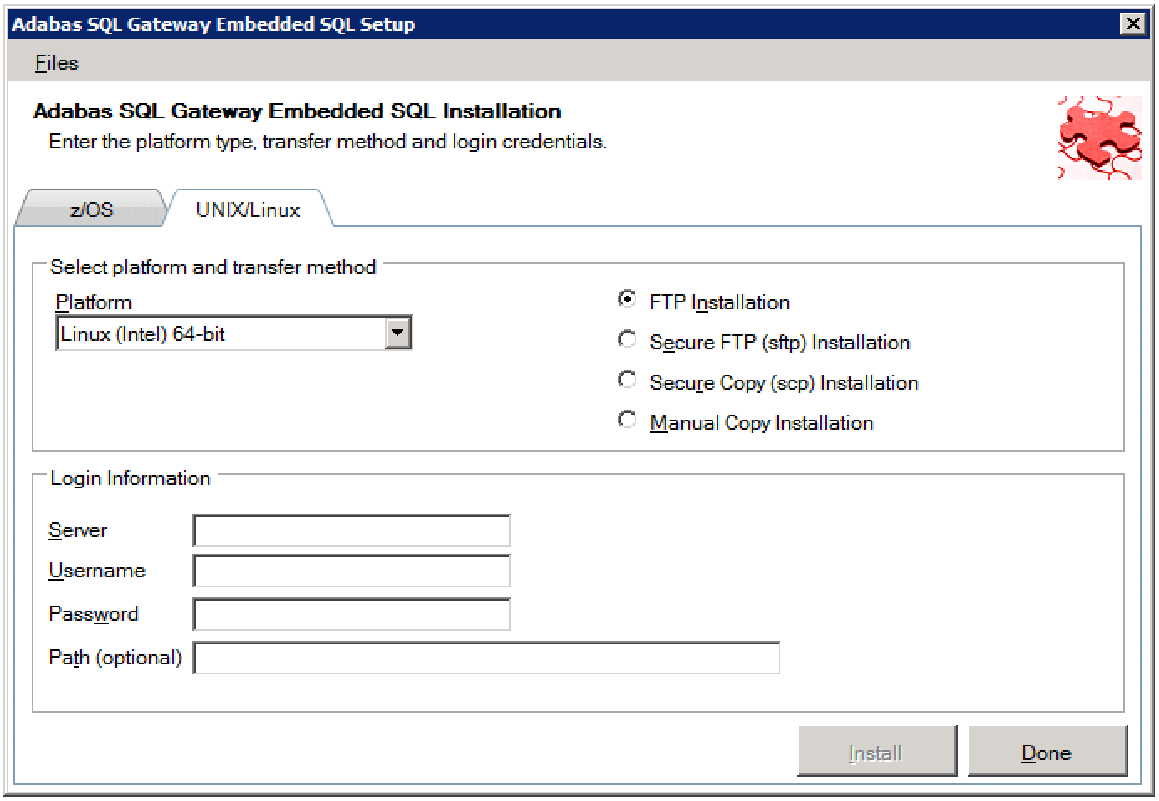
- ☑ CONNX Client Driver
- ☑ CONNX Administrator
- ☑ CONNX Help and Samples
- ☑ CONNX JDBC Server
- ☑ CONNX Enterprise Server
- ☑ Adabas SQL Gateway Embedded SQL Setup
- ☑ UNIX Server Setup
- ☑ ADABAS Mainframe Server Setup
- ☑ VSAM Mainframe Server Setup
- ☑ IMS Mainframe Server Setup
- ☑ OpenVMS Server Setup
- ☑ Open Systems Event Replicator
- ☑ Web Administrator

Description

This component installs the Adabas SQL Gateway Embedded SQL and its necessary files.

0.00 MB of space required on the C drive

97642.90 MB of space available on the C drive



Adabas SQL Gateway Embedded SQL Setup

Files

Adabas SQL Gateway Embedded SQL Installation

Enter the platform type, transfer method and login credentials.

z/OS | UNIX/Linux

Select platform and transfer method

Platform: Linux (Intel) 64-bit

● FTP Installation

○ Secure FTP (sftp) Installation

○ Secure Copy (scp) Installation

○ Manual Copy Installation

Login Information

Server

Username

Password

Path (optional)

2. If the target system has an FTP, SFTP or SCP server enabled, skip to step 4

3. If the target system does not have an FTP, SFTP or SCP server enabled, select the Manual Copy Installation option and click the Begin Client Installation button. This option will create the necessary install files in the PRECOMPILER\UNIX\TEMPINST subdirectory of the CONNX installation directory. Move these files via an alternate copy method to the Unix server and then proceed to step 11 below.
   .

4. Select an operating system from the **Platform** list box and the desired transfer method (FTP, SFTP or SCP).

5. In the **Login** area, enter the **TCP/IP symbolic host name or dotted numeric address** for your system, a privileged user account name, and a user account password in the login text boxes (for example, **Linux, CONNX,** and **Password).** The password appears as ********.

   Note: All fields are required.

6. In **Path**, enter the directory location in which you want to install the files. The location can either be a fully qualified path from the root of the file system or a path relative to the default directory of the same User ID you specified in the Login area. If the directory information is left blank then the location will be the User ID default home directory .

7. Click **Install** . This action starts an FTP session and copies the selected CONNX components from the client PC to the target host. Copying alert and FTP windows are displayed.

8. Allow several minutes for the file transfer process between the CONNX administrator computer directory C:\CONNX32\Precompiler and the target host.

9. Click **Done**. The Precompiler Setup dialog box closes.

10. Once the FTP successfully completes, you are ready to install the selected Adabas SQL Gateway Embedded components.

11. Log on to a terminal session on the target system.

12. Change the directory to the location specified in the **Path** location.

13. Run the following command:

```
./installrci
```

14. If there are no error messages the installation was successful. The **Precompiler Install Location** contains a LibRCI_* directory with the following files:

File	Purpose
ACEINT	Dynamic SQL utility
ACEPCC	C/C++ pre-compiler
ACEPCCOB	CCOBOL pre-compiler
BOOK_ORDERS.ddl	Dynamic Definition Language specification for sample files
BOOK_ORDERS.fdt	ADABAS FDT table structure definition file for samples
BOOKS.ddl	Dynamic Definition Language specification for sample files
BOOKS.fdt	ADABAS FDT table structure definition file for samples
EMPLOYEES.ddl	Dynamic Definition Language specification for sample files
EMPLOYEES.fdt	ADABAS FDT table structure definition file for samples
librciclnt_32.so	RCICLNT run-time library (this must be on the search path for pre-compiled executables and ACEAPI applications)
Makefile	Make file for samples
precompile	Pre-compiler helper script for samples
_Employees.c	Example C source with embedded SQL statements (single-threaded)
_MultiThread.c	Example C source with embedded SQL statements (multi-threaded)

Chapter 2 - Threading SQL Applications

Threading SQL Applications

Introduction

SQL client applications access Adabas SQL Gateway Embedded SQL using supplied client support libraries that are linked with the SQL application. These client support libraries also provide support for threaded SQL applications. This section explains what this means, and how to write such an application.

An SQL application can have one or more users. In the case of a commercial application that is accessible across a network (internet or intranet), the normal situation requires that the application service the needs of many users. So how does an SQL application manage all these users, or keep track of what they are doing?

Additionally, an SQL application may have multiple threads executing in parallel, in order to perform the task efficiently and quickly. The number of users trying to access an application at any given time can be greater than or less than the number of threads available to the application. How does an application manage its threading resources in order to manage the workload from the user population most efficiently?

Adabas SQL Gateway Embedded SQL provides client support for threaded SQL applications by implementing the concept of the SQL Context for managing users in a threaded environment.

SQL Context

An SQL Context is the collection of contextual information relating to database connections, cursors and diagnostics that results from the execution of SQL statements by one (logical) user.

To illustrate this, consider a user of an SQL application. The user may connect to one database in order to compile a list of books on a particular subject. Then the user may connect to a second database to check whether a specific book is currently in stock in the warehouse. In making these database queries, the user has acquired an SQL Context that consists of two database connections plus information on the SQL cursors that were used to perform the searches. In addition, any diagnostic information such as the success or failure of the SQL queries will form part of the SQL Context.

So the SQL Context is the result of Embedded SQL statements that have been executed by a user, and it also forms the basis for the execution of further statements in the future.

SQL Language Support for Embedded SQL Contexts

Embedded SQL provides language support for Embedded SQL Contexts by implementing the following Embedded SQL statements:

```
EXEC SQL ALLOCATE  SQL CONTEXT AS :hv;
```

and

```
EXEC  SQL DEALLOCATE CONTEXT AS :hv;
```

These statements instruct the Embedded SQL as to which SQL Context is in effect at any given time.

The `ALLOCATE SQL CONTEXT` statement identifies the SQL Context that is to be used for subsequent SQL statements. The host variable `:hv` must contain the address of a `SAGContext` structure (a data type defined by the client support libraries). Simple textual scoping rules apply in regards to Embedded SQL Contexts: the closest preceding `ALLOCATE SQL CONTEXT` statement for any given SQL statement defines the SQL Context that is to be used in the resulting call to the database.

Note: *The `ALLOCATE SQL CONTEXT` call itself allocates no memory; it merely specifies which SQL Context is active for subsequent SQL statements. The application must allocate whatever memory is required for the `SAGContext` structures that it uses. It is also essential that all `SAGContext` structures be initialized to zero before use.*

The `DEALLOCATECONTEXT` statement signifies that the SQL Context referenced by the host variable `:hv` is no longer required, and that the resources associated with it can be released. (Note that an SQL Context can only be deallocated when all of its database connections have been disconnected.)

The use of these statements is illustrated in SQL Threading models in Adabas SQL Gateway Embedded SQL.

SQL Threading Models in Adabas SQL Gateway Embedded SQL Clients

There are several different threading models in which SQL Contexts can be used. They are known as:

- Single Threading
- Bound Threading
- Free Threading

The Adabas SQL Gateway client support libraries support all three threading models, which are described below.

How these threading models can be implemented is described in section Implementing the Threading Models.

Single Threading

This is the trivial case in which an application has only one thread and one SQL Context. This model is limited in its ability to serve multiple users in that each request cannot assume any previous SQL Context. If such a context is required, then it must be recreated for each request, which is inefficient in terms of processing resources.

Bound Threading

In this model, an application has multiple threads and an equal number of SQL Contexts, each thread having one SQL Context which is "bound" to it. Each thread can use only the SQL Context that is bound to it.

The Bound Threading model is simply the multi-threaded generalization of the Single Threading model and, therefore, suffers from the same drawback. If the Bound Threading model is used to service multiple users, then each thread must either be locked for a specific user for the duration of the user's request in order to protect the SQL Context, or the thread must recreate the SQL Context at the start of each request.

However, if the application requirements are such that efficiency is not a priority, then the Bound Threading model can offer a useful compromise between the inefficiency of the Single Threading model and the extra programming effort required for the Free Threading model.

Free Threading

This is the most flexible threading model, in which the application has multiple threads and multiple SQL Contexts. However, the SQL Contexts are not bound to a specific thread - any thread can use any SQL Context. Consequently, the number of SQL Contexts can be greater than the number of threads. It is then possible to allocate one SQL Context for each logical user of the application, for example, for an internet-accessible application, one SQL Context for each web browser that is accessing the application.

This allows for efficient applications where the stream of requests from the user population is processed by multiple worker threads in a Multiple Server/Single Queue (MSSQ) configuration. MSSQ has the advantage of automatic load balancing between the worker threads, making it more likely that requests will be processed in the shortest possible time.

In this model, the SQL application is responsible for controlling which thread can access which SQL Context. As a consequence of this, it is possible (although not desirable) for more than one thread to simultaneously attempt to use the same SQL Context. In order to protect the integrity of SQL Contexts, the client support libraries of the Adabas SQL Gateway implement a locking mechanism to ensure that only one thread can use an SQL Context at any time.

This locking mechanism is largely transparent to the application writer. However, applications should ideally implement their own locking strategy to control access to the SQL Contexts or at least be able to cope with the situation that a thread finds an SQL Context to be locked already by another thread.

Implementing the Threading Models

The three threading models can be achieved as follows:

- The Single Threading model is in operation when an application has only one thread and uses no `ALLOCATE/DEALLOCATE SQLCONTEXT` statements. A single SQL Context is allocated transparently by the client support libraries.
- The Bound Threading model is in operation when an application has multiple threads and uses no `ALLOCATE/DEALLOCATE SQLCONTEXT` statements. Multiple SQL Contexts (one per thread) are allocated transparently by the client support libraries. Each SQL Context is bound to one thread.
- The Free Threading model is in operation when an application has multiple threads and uses the `ALLOCATE/DEALLOCATE SQLCONTEXT` statements. The SQL application is responsible for allocating all SQL Contexts, and for managing their use by the threads. Any thread can use any SQL Context, as permitted by the application.

Note: *It also possible to have a single-threaded application that uses multiple SQL Contexts; this is a variation of the Free Threading Model.*

Example 1: Simple Use of ALLOCATE/DEALLOCATE SQLCONTEXT

In order to aid clarity, the first example is a Single Threaded application that simply connects to a database and disconnects. The purpose is to demonstrate the basic use of the `ALLOCATE SQLCONTEXT` and `DEALLOCATE SQLCONTEXT` statements. Subsequent examples will demonstrate multi-threaded use of SQL Contexts.

```
#include <stdio.h>
void main(void)
{
    exec sql begin declare section;
    int         SQLCODE;
    exec sql end declare section;
    SAGContext  sqlCtx;

    /* Declare the SQL context. */
    memset(&sqlCtx, 0, sizeof(SAGContext));
    exec sql allocate sqlcontext as :&sqlCtx

    exec sql connect to 'testDB' user 'DBA' password 'dba';
    printf("connect returned SQLCODE %i\n", SQLCODE);
    if (SQLCODE != 0) exit(1);
    exec sql disconnect;
    printf("disconnect returned SQLCODE %i\n", SQLCODE);
    if (SQLCODE != 0) exit(2);
    /* Deallocate the SQL context. */
    exec sql deallocate sqlcontext as :&sqlCtx

    exit(0);
}
```

In this example, one SQL Context is defined, initialized and identified in an `ALLOCATE SQLCONTEXT` statement. The program connects to a database and immediately disconnects. The SQL Context is then deallocated.

Things to note about this example:

- Storage for the SQL Context (`sqlCtx`) is allocated by the user application, in this case by defining the context as a dynamic variable.

- It is very important that the SQL Context is set to zero before use. The application will fail if this is not done.
- The type of the host variable that is supplied in the `ALLOCATE SQLCONTEXT` statement must evaluate to `(SAGContext *)`, hence `:&sqlCtx`.
- All connections must be closed before the `DEALLOCATE SQLCONTEXT` statement is executed. Otherwise, an error will be returned to the user.
- In this example, the `ALLOCATE SQLCONTEXT` and `DEALLOCATE SQLCONTEXT` statements are actually not needed. The application would function identically if they were removed, since the client support libraries will transparently allocate one SQL Context for the single thread. Their purpose in this example is simply to illustrate their use.

Example 2: Multiple SQL Contexts, Multiple Threads

The second example illustrates an online book sales application that uses the Free Threading model.

In this simplified example, there are three threads in the application which serve a customer population of ten. Each customer requests a book search, buys one book and logs off. Each work thread has a main loop, in which it gets some work from a queue and does the work.

Therefore, any work packet could potentially be processed by any thread. There is no pre-ordained sequence of which thread processes which work packet - it is simply a matter of which thread becomes available first.

The ten customers are represented by an array of the structure `AppContext`. The `AppContext` structure encapsulates all the application data pertaining to one user. Note that this contains the SQL Context structure, as well as other application status data. The work queue is simulated by a statically initialized array of the structure `WorkPacket`.

The `main()` function starts three threads which all run the `workThread()` function and waits for them to finish. The main loop of `workThread()` gets work from the `commandQueue` array and - depending on the `commandCode` in the `WorkPacket` - calls `listBooks()`, `buyBook()` or `logoffUser()`.

The example uses threading and locking functions that are declared in *kbthreads.h*. These are not listed in order to aid clarity.

See further notes after this example.

```
#include <stdio.h>
#include <stdlib.h>
#include <kbthreads.c>
static int debug_print = 1;
#define TESTAPP_DBNAME      "testDB"
#define TESTAPP_USERNAME    "DBA"
#define TESTAPP_PASSWORD    "dba"
#define NUM_THREADS     (3)     /* Number of threads to start. */
#define NUM_CONTEXTS    (10)    /* Number of contexts in use. */
enum CommandCodes { LISTBOOKS, BUYBOOK, LOGOFF, TERMINATE, num_cmds };
/* Thread count & mutex. */
mutex_t thread_count_mutex = NULL ;
int     thread_count = 0 ;
/* Application Context (includes SQL Context). */
typedef struct {
    int         userID;
    int         appFlags;
    mutex_t     appContext_mutex;
    SAGContext  sqlContext;
} AppContext;
```

```
#define APPFLAG_CONNECTED   (1 << 0)
#define APPFLAG_WORK_DONE   (1 << 1)
AppContext userTable[] = {
    { 1000, 0, 0, 0 }, { 1001, 0, 0, 0 }, { 1002, 0, 0, 0 },
    { 1003, 0, 0, 0 }, { 1004, 0, 0, 0 }, { 1005, 0, 0, 0 },
    { 1006, 0, 0, 0 }, { 1007, 0, 0, 0 }, { 1008, 0, 0, 0 },
    { 1009, 0, 0, 0 }
};
/* Command Queue & mutex. */
typedef struct {
    int     appContext;
    int     commandCode;
    char    userArg[20];
    char    results[10][100];
} WorkPacket;
WorkPacket commandQueue[] = {
    { 1, LISTBOOKS, "computers" },      { 2, LISTBOOKS, "languages" },
    { 0, LISTBOOKS, "politics" },       { 2, BUYBOOK,   "0-002-90009-0" },
    { 0, BUYBOOK,   "0-003-90009-0" },  { 1, BUYBOOK,   "0-001-90009-0" },
    { 0, LOGOFF },                      { 1, LOGOFF },
    { 2, LOGOFF },                      { 3, LISTBOOKS, "travel" },
    { 3, BUYBOOK,   "0-004-90009-0" },  { 3, LOGOFF },
    { 4, LISTBOOKS, "sport" },          { 4, BUYBOOK,   "0-005-90009-0" },
    { 5, LISTBOOKS, "cinema" },         { 5, BUYBOOK,   "0-006-90009-0" },
    { 6, LISTBOOKS, "health" },         { 6, BUYBOOK,   "0-007-90009-0" },
    { 6, LOGOFF },                      { 9, LISTBOOKS, "humour" },
    { 7, LISTBOOKS, "poetry" },         { 8, LISTBOOKS, "history" },
    { 8, BUYBOOK,   "0-010-90009-0" },  { 7, BUYBOOK,   "0-009-90009-0" },
    { 9, BUYBOOK,   "0-008-90009-0" },  { 7, LOGOFF },
    { 5, LOGOFF },                      { 4, LOGOFF },
    { 9, LOGOFF },                      { 8, LOGOFF },
    { 0, TERMINATE }
};
mutex_t commandQueue_mutex = NULL;
int     cqIndex = 0;
int     cqLength = 31;
exec sql begin declare section;
    char testDb[20];
    char testUser[20];
    char testPass[20];
exec sql end declare section;
int sqlError(SQL_INTEGER    sqlcodeOrig,
             char *         stmt,
             AppContext *   pAppContext,
             int            appCtxNo)
{
    exec sql begin declare section;
    int             SQLCODE;
    int             i;
    int             errCnt = 0;
    char            errTxt[128];
    exec sql end declare section;
    SAGContext *    pSqlCtx = &pAppContext->sqlContext;
    if (sqlcodeOrig < 0)
    {
        /* Error */
        printf("C%i: %s: returns SQLCODE=%i\n", appCtxNo, stmt, sqlcodeOrig);
        exec sql allocate sqlcontext as :pSqlCtx;
        exec sql get diagnostics :errCnt = number;
        for (i = 1; i <= errCnt; i++)
        {
            memset(errTxt, 0, 128);
            exec sql get diagnostics exception :i :errTxt = MESSAGE_TEXT;
            printf("C%i: %s: SQLCODE=%i,  %s\n",
                   appCtxNo, stmt, sqlcodeOrig, errTxt);
        }
        /* Tidy-up work/connections. */
        if (pAppContext->appFlags & APPFLAG_WORK_DONE)
        {
            pAppContext->appFlags &= ~APPFLAG_WORK_DONE;
            exec sql rollback work;
            sqlError(SQLCODE, "rollback work", pAppContext, appCtxNo);
        }
        if (pAppContext->appFlags & APPFLAG_CONNECTED)
        {
            pAppContext->appFlags &= ~APPFLAG_CONNECTED;
            exec sql disconnect;
            sqlError(SQLCODE, "disconnect", pAppContext, appCtxNo);
        }
        return(-1);
    }
    else
    {
        return(0);
    }
}
int listBooks(WorkPacket * pWork, int threadNo, int workPacketNo)
{
    exec sql begin declare section;
    int             SQLCODE = 0;
    char            userChoice[20] = { 0 };
    char            theISBN[20] = { 0 };
    char            theTitle[60] = { 0 };
    int             thePrice = 0;
    exec sql end declare section;
    int             rowCount;
    AppContext *    pAppCtx = &userTable[pWork->appContext];
    SAGContext *    pSqlCtx = &pAppCtx->sqlContext;
    /* Set up the SQL context host variable. */
    exec sql allocate sqlcontext as :pSqlCtx;
    if (!(pAppCtx->appFlags & APPFLAG_CONNECTED))
    {
        exec sql connect to :testDb user :testUser password :testPass;
        if (sqlError(SQLCODE, "connect", pAppCtx, pWork->appContext))
        {
            return(-1);
        }
        else
        {
            pAppCtx->appFlags |= APPFLAG_CONNECTED;
        }
    }
    strcpy(userChoice, pWork->userArg);
    exec sql declare listCur cursor for
```

```
        select isbn, title, price from books where category = :userChoice;
    exec sql open listCur;
    if (sqlError(SQLCODE, "open", pAppCtx, pWork->appContext))
    {
        return(-1);
    }
    for (SQLCODE = 0, rowCount = 0; SQLCODE == 0; rowCount++)
    {
        exec sql fetch listCur into :theISBN, :theTitle, :thePrice;
        if (SQLCODE == 100)
        {
            break;
        }
        else if (sqlError(SQLCODE, "fetch", pAppCtx, pWork->appContext))
        {
            return(-1);
        }
        else
        {
            sprintf(pWork->results[rowCount],
                    "%s,%s,%i\n", theISBN, theTitle, thePrice);
        }
    }
    exec sql close listCur;
    if (sqlError(SQLCODE, "close", pAppCtx, pWork->appContext))
    {
        return(-1);
    }
    if (debug_print)
    {
        printf(" listBooks(): T%i C%i WP%i - success\n",
               threadNo, pWork->appContext, workPacketNo);
    }
    return (0);
}
int buyBook(WorkPacket * pWork, int threadNo, int workPacketNo)
{
    exec sql begin declare section;
    int             SQLCODE = 0;
    int             thisOrdnum = 0;
    int             thisUser = 0;
    char            theISBN[20] = { 0 };
    exec sql end declare section;
    AppContext *    pAppCtx = &userTable[pWork->appContext];
    SAGContext *    pSqlCtx = &pAppCtx->sqlContext;
    /* Set up the SQL context host variable. */
    exec sql allocate sqlcontext as :pSqlCtx;
    thisOrdnum = workPacketNo;
    thisUser = pAppCtx->userID;
    strcpy(theISBN, pWork->userArg);
    exec sql insert into orders (ordnum, userid, isbn)
                values (:thisOrdnum, :thisUser, :theISBN);
    if (sqlError(SQLCODE, "insert", pAppCtx, pWork->appContext))
    {
        return(-1);
    }
    exec sql commit work;
    if (sqlError(SQLCODE, "insert", pAppCtx, pWork->appContext))
    {
        return(-1);
    }
    if (debug_print)
    {
        printf("   buyBook(): T%i C%i WP%i - success\n",
               threadNo, pWork->appContext, workPacketNo);
    }
    return (0);
}
int logoffUser(WorkPacket * pWork, int threadNo, int workPacketNo)
{
    exec sql begin declare section;
    int             SQLCODE = 0;
    exec sql end declare section;
    AppContext *    pAppCtx = &userTable[pWork->appContext];
    SAGContext *    pSqlCtx = &pAppCtx->sqlContext;
    /* Set up the SQL context host variable. */
    exec sql allocate sqlcontext as :pSqlCtx;
    exec sql disconnect current;
    if (sqlError(SQLCODE, "disconnect", pAppCtx, pWork->appContext))
    {
        return(-1);
    }
    else
    {
        pAppCtx->appFlags &= ~APPFLAG_CONNECTED;
        if (debug_print)
        {
            printf("logoffUser(): T%i C%i WP%i - success\n",
                   threadNo, pWork->appContext, workPacketNo);
        }
        return (0);
    }
}
static DWORD WINAPI workThread(LPVOID threadArg)
{
    int             threadNo = (int) threadArg;
    int             retval = 0;
    WorkPacket *    pWork;
    AppContext *    pAppCtx;
    int             workCode;
    int             workPacketNo;
    Sleep(THREAD_INIT_SLEEP_TIME) ;
    do
    {
        if (!lock_mutex(commandQueue_mutex, INFINITE))
        {
            printf("FAILURE: workThread(%i): lock_mutex(cq) FAILS\n",
threadNo);
        }
        workPacketNo = cqIndex;
        if (cqIndex < (cqLength - 1))
        {
            cqIndex++;
        }
        pWork = &commandQueue[workPacketNo];
        workCode = pWork->commandCode;
        if (workCode != TERMINATE)
        {
            pAppCtx = &userTable[pWork->appContext];
            if (!lock_mutex(pAppCtx->appContext_mutex, INFINITE))
            {
                printf("FAILURE: workThread(%i): lock_mutex(aq %i) FAILS\n",
                       threadNo, pWork->appContext);
            }
        }
        /* Safe to unlock command queue now. */
        if (!unlock_mutex(commandQueue_mutex))
        {
            printf("FAILURE: workThread(%i): unlock_mutex(cq) FAILS\n",
                   threadNo);
        }
        switch (workCode)
        {
            case LISTBOOKS:
                retval = listBooks(pWork, threadNo, workPacketNo);
                break;
            case BUYBOOK:
                retval = buyBook(pWork, threadNo, workPacketNo);
                break;
            case LOGOFF:
                retval = logoffUser(pWork, threadNo, workPacketNo);
                break;
            case TERMINATE:
                printf(" <TERMINATE>: T%i C%i WP%i - success\n",
                       threadNo, pWork->appContext, workPacketNo);
        }
        if (workCode != TERMINATE)
        {
            if (!unlock_mutex(pAppCtx->appContext_mutex))
            {
                printf("FAILURE: workThread(%i): unlock_mutex(aq %i)
FAILS\n",
                       threadNo, pWork->appContext);
            }
        }
    } while ((workCode != TERMINATE) && (retval == 0));
    /* Reduce thread_count. */
    if (!lock_mutex(thread_count_mutex, INFINITE))
    {
        printf("FAILURE: workThread(%i): lock_mutex(tc) FAILS\n", threadNo);
    }
    thread_count--;
    if (!unlock_mutex(thread_count_mutex))
    {
        printf("FAILURE: workThread(%i): unlock_mutex(tc) FAILS\n",
threadNo);
    }
    exit_thread(EXIT_THREAD_CODE);
    return(0);
}
int main(int argc, char ** argv)
```

```
{
    int             threadNo, ctxCount, old_count = 0;
    bool_t          finished = FALSE;
    thread_t        thread_h[NUM_THREADS];
    SAGContext *    pSqlCtx = 0;
    exec sql begin declare section;
    int             SQLCODE;
    exec sql end declare section;
    /* Set up the db, user & passwd. */
    strcpy(testDb, TESTAPP_DBNAME);
    strcpy(testUser, TESTAPP_USERNAME);
    strcpy(testPass, TESTAPP_PASSWORD);
    /* Create thread-count and command queue mutex locks. */
    if (!create_mutex(FALSE, &thread_count_mutex)
        || !create_mutex(FALSE, &commandQueue_mutex))
    {
        printf("Can't create TC/CQ mutex lock\n");
        exit(ABORT_THREAD_CODE);
    }
    /* Create mutex locks for application contexts. */
    for (ctxCount = 0; ctxCount < 10; ctxCount++)
    {
        if (!create_mutex(FALSE, &userTable[ctxCount].appContext_mutex))
        {
            printf("Can't create AC mutex lock\n");
            exit(ABORT_THREAD_CODE);
        }
    }
    /* Start the worker threads. */
    for (threadNo = 0; threadNo < NUM_THREADS; threadNo++)
    {
        if (!lock_mutex(thread_count_mutex, INFINITE))
        {
            printf("FAILURE: main: lock_mutex 1 FAILS\n");
        }
        thread_count++;
        create_thread(workThread, (void *)threadNo, &thread_h[threadNo]);
        if (!unlock_mutex(thread_count_mutex))
        {
            printf("FAILURE: main: unlock_mutex 1 FAILS\n");
        }
    }
    /* Wait for threads to finish. */
    while (!finished)
    {
        if (!lock_mutex(thread_count_mutex, INFINITE))
        {
            printf("FAILURE: main: lock_mutex 2 FAILS\n");
        }
        if (thread_count != old_count)
        {
            old_count = thread_count;
            printf("%d threads running\n", thread_count);
        }
        finished = (thread_count <= 0);
        if (!unlock_mutex(thread_count_mutex))
        {
            printf("FAILURE: main: unlock_mutex 2 FAILS\n");
        }
        if (!finished)
        {
            Sleep(TIMEOUT_SLEEP);
        }
    }
    /* Deallocate all SQL contexts. */
    exec sql allocate sqlcontext as :pSqlCtx;
    for (ctxCount = 0; ctxCount < 10; ctxCount++)
    {
        pSqlCtx = &userTable[ctxCount].sqlContext;
        exec sql deallocate sqlcontext as :pSqlCtx;
    }
    for (threadNo = 0; threadNo < NUM_THREADS; threadNo++)
    {
        CloseHandle(thread_h[threadNo]);
    }
    printf("Finished.\n");
}
```

Things to note:

- The initialization of the `userTable` ensures that the `SAGContext` structure (the SQL Context) is initialized to zeroes before it is used. It is essential that an SQL application does this.
- The `AppContext` structure contains a lock mechanism (`appContext_mutex`) which is used to serialize access on a customer's SQL Context. This avoids errors being returned by the client support libraries, which will detect whether an SQL Context is already in use. Applications are responsible for managing access to SQL Contexts. They must either ensure that only one thread can access an SQL Context at a time (as this example does), or be prepared to handle errors from the SQL interface.
- SQL diagnostic information is held in the SQL Context. Therefore, the error reporting function `sqlError()` also specifies which SQL Context it is using.
- Note that all the SQL Contexts are deallocated by the main thread after all the work threads have finished. It would also have been possible to deallocate the active context after the disconnect statement in the `logoffUser()` function. The application can deallocate an SQL Context whenever it has finished with it, providing it contains no active SQL connections.

Description of Client Configuration

Adabas SQL Gateway Embedded SQL precompiler requires access to the RCI run-time library (RCICLNT.dll). ACEPCC is a console based application and is therefore invoked from the command line. Outside of the directory where ACEPCC is installed (See: Chapter 1 - Installation Instructions: Precompiler) the ACEPCC executable must be on the system search path or the full path to the executable must be specified on the command line. ACEPCC does not require a development environment to function, however, a development environment for ANSI C or C++ must be available in order to compile and execute the Precompiler output.

Server Configuration

Description of Server Configuration

The server for the RCI is defined as the Windows machine running an instance of the CONNX RCI/JDBC Server. In order for the Precompiler or the RCI run-time to function, an instance of the RCI/JDBC server service must be running and available. For details on the installation and configuration of the RCI/JDBC server, please refer to the CONNX documentation, available online, on the CONNX CD-ROM, and within the CONNX Adabas SQL Gateway product.

CDD / JDBC Server

To access data from the RCI, the schema definition for the tables referred to in the user's embedded SQL must be defined in a CONNX Data Dictionary (CDD).

For more information on how to import into and configure a CDD please refer to the CONNX documentation, available online, on the CONNX CD-ROM, and within the CONNX application.

Once the CDD has been defined, the CDD must be made available to the RCI/JDBC server by invoking the CONNX DSNRegistry tool to create a logical name, known as a Data Source Name (DSN) that points to the CDD. The CDD DSN should be registered on the same machine on which the CONNX RCI/JDBC Server is installed. The CONNX RCI/JDBC Server checks the Windows registry on that machine to locate the DSNs.

Note: A DSN for use with the RCI is not the same as an ODBC DSN. If you intend to use both ODBC and the RCI to access databases, you must create a DSN for both access methods. A DSN registered for the RCI is also compatible with CONNX JDBC.

Client / Server Communication

Description of Client / Server Communication

The CONNX RCI/JDBC Server component is a Microsoft Windows server component that communicates with the RCI run-time. It is a Windows executable that opens a socket and listens for new connections. When it accepts a new connection, it creates a new thread that is dedicated towards communicating to that client. Installing the CONNX RCI/JDBC Server component on every machine is an optional task since only one server is required for communication with all CONNX RCI/JDBC client machines.

Host / Port

At any time during the execution of the precompiler or the RCI run-time, if a Host / Port combination is required, the information requested is the HOST address of the RCI/JDBC server and the PORT that the server is listening on. The default PORT for the RCI/JDBC server is 7500. For more information on how to configure the RCI/JDBC server, please refer to the CONNX documentation, available online, on the CONNX CD-ROM, and within the CONNX application.

SQL Statements

SQL statements are not part of the host language but are embedded in an application written in the host language. The compilation of such a program consists of two phases; the precompilation of the SQL statements contained in the application program followed by the compilation of the actual program itself.

The SQL statements must be invisible to the host language compiler during the compilation phase. In fact, the embedded SQL statements are commented out by the precompiler and are replaced by statements generated into the application program in a form that corresponds to the requirements of the host language.

The precompiler must be able to identify all embedded SQL statements. Therefore, all SQL statements are delimited by special SQL delimiters. It is not possible to have more than one SQL statement between one set of delimiters.

The SQL Starting Delimiter

The starting delimiter consists of a sequence of two words:

```
EXEC  SQL
```

These words must be separated by one or more whitespace characters. They may be separated by one or more lines or blanks, and may be in either upper case or lower case depending on what the host language permits.

In ANSI mode, the two keywords must be in upper case and must be separated by blanks (not lines).

The SQL Statement Body

Once the starting delimiter has been specified, the statement itself must be provided. It must be separated from the starting delimiter by at least one whitespace character and may be specified on the same line or on a following line to the starting delimiter. The statement may be specified in either upper case or lower case and may be split over several lines. Each keyword or token must be separated by at least one whitespace character and may not be split over two or more lines. Keywords may be written in upper case or lower case depending on the host language. In ANSI mode, keywords must be written in upper case only.

The SQL Communications Area (SQLCA)

Any application program needs to be able to check the success or failure of any particular SQL statement once it has been executed. At least one special host variable structure needs to be declared in the program, so that there is always one in scope for each SQL statement. For this purpose, a host variable structure, called SQL communication area (SQLCA) is used. Adabas SQL Gateway Embedded SQL updates certain fields of the structure depending on the nature of the particular SQL statement and the outcome of its execution. The application program can verify the successful execution of an SQL statement by inspecting the contents of the sqlcode element of the SQLCA.

Declaring the SQLCA

As stated above, the SQLCA is a special type of host variable structure. In order to ensure that the structure has the correct format, the application program should use the definition of the SQLCA provided by the Adabas SQL Gateway Embedded SQL. To facilitate this, the following SQL statement should be embedded in the application program:

```
INCLUDE SQLCA;
```

Executing this statement has the effect of generating an appropriate SQLCA definition and declaration at the point where it is specified. Thus, the SQLCA obeys the rules of scoping set by the host language relative to the position of the INCLUDE SQLCA statement.

Application programs can explicitly declare an SQLCA without using the INCLUDE statement. It is then the responsibility of the programmer to ensure that the structure is correctly defined and declared. Failure to do so may lead to unpredictable results.

Using the SQLCA

Once the SQL statement execution has completed, the application program should check the SQLCODE field of the SQLCA. The program logic should then be in a position to deal with any eventuality. This may be done for every SQL statement. However, by using the precompiler directive WHENEVER, such coding can be generated automatically.

Currently, not all fields in the SQLCA are used.

Note: The following static statements do not result in any update of the SQLCA:

DECLARE CURSOR

BEGIN DECLARE SECTION

END DECLARE SECTION

WHENEVER

INCLUDE

Field	Description
sqlcaid	An eight-byte character string containing the constant SQLCA. This field serves mainly as an eye-catcher for easy memory dump interpretation.
sqlcabc	A four-byte integer variable containing the length in bytes of the SQLCA. It normally contains the value 136.
sqlcode	A four-byte integer variable containing the status of the executed SQL command. The standard defines three categories of results.
	zero The command has been successfully executed. (There may have been warning messages) negative An error has occurred. The negative number indicates the nature of the error.

	Adabas SQL Gateway Embedded SQL allows the installation to define its own error values. Thus, compatibility with different SQL DBMSs can be achieved. (The ANSI/ISO standard does not specify which negative values should be used with a particular error status). When a negative code is returned, the SQLERROR condition of the WHENEVER statement is activated. positive The command executed successfully, but an exceptional condition occurred. +100 This value is returned to indicate that the command was successfully executed but processed no rows. It is used in conjunction with the following commands: DELETE FETCH INSERT SELECT UPDATE
sqlerrm	A variable containing two fields holding the actual values to replace the variables contained in error messages.
	sqlerrml A two-byte integer field. This field is currently not used. sqlerrmc A character string of variable length which may not exceed 70 characters. This field is currently not used. The string contains one or more actual values for the variables of the associated error messages. As many error messages contain no text variables, this field is not always filled. Each value in the string is terminated by one byte containing the hex value FF.
sqlerrp	An eight-byte character variable. This field is currently not used.
sqlerrd1 - 6	A group of six integer fields, each four bytes in length.
	sqlerrd1 Currently not used. sqlerrd2 Currently not used. sqlerrd3 Specifies how many rows were processed by the SQL statement. sqlerrd4 Currently not used. sqlerrd5 Currently not used. sqlerrd6 Currently not used.
sqlwarn0 - 7	A group of eight character variables, each one byte in length. The default content is blank. This field is currently not used.
	sqlwarn0 Currently not used. sqlwarn1 Currently not used. sqlwarn2 Currently not used. sqlwarn3 Currently not used. sqlwarn4 Currently not used. sqlwarn5 Currently not used. sqlwarn6 Currently not used. sqlwarn7 Currently not used.
sqlext	An eight-byte character string. This field is currently not used.

Program Structure

To develop correct SQL application programs, it is important to understand the difference between the physical order of the SQL statements in a program and the order of their execution.

The Adabas SQL Gateway Embedded SQL scans the source application program for SQL statements and effectively skips any host language statements or commands. The Adabas SQL Gateway Embedded SQL has no understanding of the underlying logic of the application or the context of any particular SQL statement. All it can actually understand is an isolated collection of SQL statements.

Under the following circumstances, the physical order of the statements is relevant and does not have anything to do with the actual order in which the statements will be executed.

When running in ANSI compatibility mode, any statements which reference a cursor must physically follow the associated DECLARE CURSOR statement. In Adabas SQL Gateway Embedded SQL mode, this restriction does not exist and so the physical order of such statements is irrelevant.

In Adabas SQL Gateway Embedded SQL mode, although the DECLARE and OPEN statements must be in the same source file, other associated statements need not be. For more details, refer to the section on Static SQL.

The physical ordering of other statements can now follow freely as long as any host variables accessed within an SQL statement have been declared physically prior to usage and an SQLCA is in scope for each statement.

Positioning the SQL Statement

Almost all SQL statements may be positioned anywhere within an application program where a host language statement is permitted. This is, because in general, SQL statements are replaced with appropriate generated host language statements by the precompiler. The rules governing the positioning of host language statements also apply to the embedding of SQL statements. The positioning of the SQL statement must also conform to the context of the logic of the application program. As long as each SQL statement is individually delimited, where the host language permits, more than one SQL statement may be positioned on a single line.

The following statements are exceptions to the above rules:

BEGIN DECLARE SECTION
END DECLARE SECTION
INCLUDE
	These statements can only be positioned where host language declarations are allowed.

WHENEVER **DECLARE CURSOR (static)**	These statements can be placed anywhere depending on the desired control flow.

SQL Commands and Grammar

A complete description of CONNX-supported SQL Grammar can be found in the CONNX User Reference Guide.

Transaction Logic

Database modifications are performed using the transaction concept. A transaction consists of the following statements:

1. INSERT, DELETE or UPDATE statements. Such statements define the changes which are to be applied to the database.
2. A COMMIT statement which causes the changes to be applied to the database. Successful execution of a COMMIT statement causes the transaction to be closed.

A ROLLBACK statement can be used to back out any changes made to the database by the current (open) transaction.

Any rows in the database which are modified are placed in hold status until the transaction is completed. This prevents any conflicting modification by other users who must wait until the row is released at the completion of the transaction.

Transactions Containing Different Types of Statements

The execution of DDL and DCL statements may be mixed in the same transaction, but may not be mixed with the execution of DML statements.

The mixing of DML and DDL/DCL statement execution within one transaction will be detected, the violating statement execution will be rejected, and an error message will be issued. The current transaction status will not be affected. For example, in a transaction with only DDL/DCL statements, a DML statement will be considered a violating statement and vice versa.

Transaction neutral statements (PREPARE, EXECUTE, EXECUTE IMMEDIATE and DESCRIBE) may be mixed with all other statements in one transaction. They may be contained in a DDL transaction, a DCL transaction, a mixed DDL and DCL transaction and also in a DML transaction.

Static SQL

Introduction

Static SQL refers to a particular type of application where SQL statements are fixed or static. This is as opposed to dynamic SQL where the actual statement to be executed against a database is created at run time. Thus static SQL statements are embedded SQL statements that do not vary during the execution of the application program.

Strictly speaking, embedded statements like the PREPARE statement are also static. They are embedded in an application program but they enable the use of dynamic SQL statements. Such statements are described in the section Dynamic SQL later in this documentation.

The following types of statements can be used:

- **DDL Statements**
  Data Definition Language statements define database structures. An example of a DDL statement is the CREATE TABLE statement.
- **DML Statements**
  Data Manipulation Language statements perform operations on the data in the database. An example of a DML statement is the SELECT statement.
- **DCL Statements**
  Data Control Language statements control access to data. An example of a DCL statement is the GRANT statement.

Manipulating Data

The DML (Data Manipulation Language) component of SQL provides the following functionality:

- populates the data structures with actual data,
- enables the retrieval of data from the data structures,
- updates the data by either changing or removing values.

Two distinct concepts are provided for data manipulation, non-cursor and cursor operations.

Non-cursor-based Statements

Statements which are not based on a cursor are not associated with other statements in any way.

Data is generally retrieved by using the SELECT statement. An INTO clause and an appropriate host variable list must be provided in order to receive the returned data. This mechanism, therefore, does not facilitate the retrieval of more than one row. An embedded static SELECT statement may only generate one row, otherwise an exception condition will occur during run time. It is the programmer's responsibility to ensure that the SELECT statement really does only return a single row. It is not possible for this to be checked in any way during the compilation of the statement.

Note: *In Interactive SQL or Dynamic SQL there is no such restriction on the number of rows which can be retrieved by a SELECT statement.*

The host variable list should match the derived columns list in every aspect. There is a one-to- one correspondence between a derived column and a host variable. The basic type of the host variable must match that of the corresponding derived column. The relative number of items should be the same but need not be. If there are insufficient host variables then data will be lost. If there are too many then the contents of the extra host variables will be undefined after the statement has completed. In either case a compiler warning is issued. Should an error occur during the execution of the query, the values in the host variables are undefined.

Inserting Single Rows

Data is inserted into a table using the INSERT statement. The data is inserted on a row by row basis. The source of the data can either be literals or host variables (that is non-SQL derived data) or from a subquery (that is SQL-derived data). When specified using non-SQL data, only one row may be inserted for one execution of the statement. If a subquery is used then as many rows as the subquery provides are inserted for one execution of the statement. The subquery may not access the target table. There is no corresponding cursor-based INSERT statement.

Updating Rows

Data can be updated by using the UPDATE statement. The data is updated on a row by row basis as identified by the search expression. Therefore, more than one row can be updated at a time.

Deleting Rows

Rows of data can be removed from the table by using the DELETE statement. The data is deleted on a row by row basis as identified by the search condition and, therefore, many rows can be deleted at once. If no search condition is specified, then all rows are identified and the table is cleared of all its data.

Level 1 or level 2 tables cannot be the target of DELETE statements. Data from such tables can only be removed by deleting the associated level 0 row. In such a case the referencing level 1 and level 2 rows are deleted automatically with the level 0 row. This is analogous to a DELETE CASCADE in referential integrity terminology.

Cursor-based Statements

As described in the previous section, non-cursor-based statements are not suitable for accessing and updating more than one row in a table. This requires the use of cursors as described below.

Declaring and Opening a Cursor

A cursor is declared in a DECLARE CURSOR statement along with the underlying query expression. which defines a resultant table. The cursor is used as a pointer to a particular row of this table. At runtime, a static DECLARE CURSOR statement has no effect, it is only a declaration for the SQL compiler.

When the cursor is opened by an OPEN CURSOR statement, the runtime system establishes the resultant table with the cursor pointing to the first row.

Other SQL compilers require that a DECLARE CURSOR statement is followed by the OPEN CURSOR statement. This is because the information contained in the DECLARE CURSOR statement has to be attached to the OPEN statement. Adabas SQL Gateway does not have this restriction. As the static DECLARE CURSOR statement has no effect at runtime, the logical order is not relevant.

Once the cursor has been opened, it can be used by other statements.

Retrieving Data Using a Cursor

Data is retrieved from the resultant table using the FETCH statement. This statement specifies the cursor to be used and a target buffer list which is similar to that of the single row SELECT statement. The target buffer list must match the projection list of the query expression. Each time the FETCH is executed, it moves the cursor to one row and copies the values of the derived columns into the corresponding host variables of the target buffer list. For a newly opened cursor, the first row of the resultant table will be retrieved and the values will be made available to the application program in the host variables. The cursor now points to the first row. Each execution of the FETCH statement results in successive rows of the resultant table being retrieved.

Once all the rows have been fetched, the cursor is said to be exhausted and points past the last row. After the last row has been fetched, the next and any subsequent FETCH statement will result in a return code of +100 being issued by the runtime system. This needs to be checked by the application program either explicitly or by specifying a WHENEVER statement with the NOT FOUND option. Once a cursor is exhausted, it should be closed.

The current row, as determined by the cursor's position in the resultant table, can be updated by using a positioned UPDATE statement. The use of such a statement does not affect the position of the cursor. Only one row, the current one, can be updated by using this statement. However, by embedding the statement in the same loop as the FETCH statement, each row of the resultant table can be updated successively. For this reason, Adabas SQL Gateway permits the use of a FETCH statement without having to specify a target buffer.

Similarly, the current row can be removed from the underlying base table by executing a positioned DELETE statement against the cursor. After execution of the DELETE statement, the row no longer exists, but a FETCH statement must still be executed in order to position the cursor to the next row.

Both the positioned UPDATE and DELETE statements are only valid if it is determined during compilation that the cursor can be updated as specified in SQL Statements.

Closing a Cursor

A cursor can be closed at any time. Normally, it is closed once all rows have been fetched and the cursor is positioned past the last row. Closing the cursor means that the resultant table is discarded along with any internal resources required for the cursor's processing. A cursor is also closed implicitly if a COMMIT or ROLLBACK statement is executed without the KEEPING ALL option.

Programming Logic for Cursor Usage

In general, the FETCH, the positioned UPDATE or DELETE, and the CLOSE statements should appear in the same compilation unit as the associated DECLARE CURSOR statement. This is because all the necessary checks to see if the statement is valid can be performed at compile time. Adabas SQL Gateway, however, does permit such statements to be located in another compilation unit. This is intended to aid the modular design of the application. However, it should be noted that the necessary compile time checks are performed at runtime and may result in a loss of performance.

An application program may contain many DECLARE CURSOR statements but each cursor identifier must be unique. For each DECLARE CURSOR statement there must be at least one OPEN CURSOR statement, and it must be in the same compilation unit. As long as it is not compiled under ANSI compatibility mode, the OPEN CURSOR statement need not follow the DECLARE CURSOR statement. There may also be many instances of the FETCH, positioned UPDATE or DELETE and the CLOSE CURSOR statements. As long as a cursor is closed, either explicitly or implicitly, it may be opened as many times as required.

Closing the cursor does not commit any changes made to the underlying base table. However, these changes are visible to the user, once the cursor is re-opened during the current transaction.

Dynamic SQL

Introduction to Dynamic SQL

The principle difference between static and dynamic SQL statements is the point in time when the SQL statements are constructed and compiled.

Static SQL Statements

A static SQL statement is embedded in a host language program and must be precompiled. The type of statement, the tables, views, columns referenced, and search conditions cannot be changed at runtime. Although host variables can be used to provide search values at runtime, the general content of the statement, for example, the derived column list, the search conditions, cannot be changed at runtime.

Dynamic SQL Statements

A dynamic SQL statement is constructed at runtime. The statement, including the tables, views and columns referenced, and the search conditions, is compiled at runtime.

Dynamic SQL usage requires the use of the following statements:

PREPARE, EXECUTE, EXECUTE IMMEDIATE and DESCRIBE.

There are also some SQL statements which are normally used as static SQL statements, but have an extended functionality for dynamic SQL:

DECLARE CURSOR, OPEN and FETCH.

Note: *The above statements are embedded in the application program like any other static SQL statement. However, they enable the use of dynamic SQL statements which are not embedded in the application program.*

There are various methods of using dynamic SQL statements, mainly depending on the type of SQL statement (SELECT or NON-SELECT statements) and the degree of flexibility required.

General Aspects

Dynamic SQL Principles

The processing of a dynamic SQL statement consists of the following steps:

- A string containing the SQL statement is created. The application program has complete control over the contents of the string and therefore, the SQL statement is dynamic in nature.
- After the SQL statement has been constructed, it must be passed to Adabas SQL Gateway Embedded SQL for compilation. This is done either using a PREPARE statement or an EXECUTE IMMEDIATE statement. The compiled form of the dynamic SQL statement is called the prepared statement.
- If the statement was processed using an EXECUTE IMMEDIATE statement, it is not only compiled but also executed at the same time. The prepared form of the statement is not retained.
- If the statement was processed using a PREPARE statement, the prepared statement can be executed using an EXECUTE statement, or by using cursor processing, as many times as required.
- It may be that additional information about the prepared statement is required before it can be executed, for example, for statements with an unknown derived column list. This information can be retrieved from Adabas SQL Gateway Embedded SQL through an SQL descriptor area (SQLDA) using a DESCRIBE statement. An SQLDA can also be used to resolve host variable markers. Such information must be obtained prior to execution.

After the dynamic SQL statement has been prepared, it may be executed more than once, by using the same statement identifier. A prepared statement remains available until the completion of the current session. A prepared statement can be deallocated using a DEALLOCATE PREPARE statement.

Dynamic versus Static SQL - Considerations

The choice between static and dynamic SQL is a choice of flexibility versus complexity and performance. It is easier to code static SQL statements than to construct SQL statements dynamically. In most cases, it will be possible to use static SQL, but there are some applications where the use of dynamic SQL is unavoidable. If the number of static SQL statements that would be required for a certain application exceeds a manageable amount, dynamic SQL may be the solution.

In principle, the question to be answered is:

- Is it possible to define all necessary SQL statements in my application and will this be a manageable and feasible amount of coding?

  If the answer is no, then dynamic SQL needs to be considered.

Once dynamic SQL has been identified as a viable possibility, the particular variation or degree of complexity of dynamic SQL must be decided upon. The following questions to be answered are:

- Must the program contain dynamically constructed SELECT statements or not?
- If SELECT statements are dynamically constructed, does the derived column list vary dynamically?
- Are host variable markers to be used, and if so, does the number and type of host variable markers vary dynamically per prepared statement?

Another point of consideration, when deciding whether to use dynamic SQL, is the issue of performance. Compiling SQL statements at runtime has an influence on the overall performance of the execution of that SQL statement. The compilation of an SQL statement also includes the access of information stored in the catalog, like table and column descriptions. These descriptions are buffered, but the possibility exists that additional database requests need to be issued.

The consequence of the fact that a dynamic SQL statement is compiled at runtime is also that the statement is compiled with more current information concerning the existence of indices and other optimization information.

Using dynamic SQL, also means that syntactical and semantical errors are only detected during runtime. This means that the PREPARE and EXECUTE IMMEDIATE statements may return an SQLCODE indicating a syntactical or semantical error.

Limitations

The following SQL statements cannot be used as dynamic SQL statements, that is, they cannot be prepared or executed:

BEGIN DECLARE

CLOSE

DEALLOCATE PREPARE

DECLARE

DESCRIBE

DISCONNECT

END DECLARE

EXECUTE

EXECUTE IMMEDIATE

FETCH

INCLUDE

OPEN

PREPARE

WHENEVER

Dynamic SQL may require the use of addresses and pointers within the application program. It may also require dynamically obtained memory.

Non-Select Statements

The simplest form of dynamic SQL programs do not contain SELECT statements. In such a case, there is no resultant table and no data has to be passed back to the application program.

There are two ways to execute a NON-SELECT SQL statement dynamically; using the EXECUTE IMMEDIATE statement, or using PREPARE and EXECUTE statements.

Using EXECUTE IMMEDIATE

The EXECUTE IMMEDIATE statement has a single parameter which must be a character string which contains the dynamic SQL statement. The string has to be constructed properly by the application program. The dynamic SQL statement is then compiled and executed immediately. The compiled form of the SQL statement (the prepared statement) is discarded after execution.

Example:

```
EXEC SQL
EXECUTE IMMEDIATE :dyn_sql_statement;
```

where dyn_sql_statement is a character string containing the dynamic SQL statement.

All SQL statements which can be prepared can be used in an EXECUTE IMMEDIATE statement except for the SELECT statement.

Notes: *If the string representing the dynamic SQL statement cannot be compiled, the SQLCODE will indicate this error after execution of the EXECUTE IMMEDIATE statement.*

It is not possible to use host variable markers in the dynamic SQL statement when using the EXECUTE IMMEDIATE statement.

Using PREPARE and EXECUTE

The compilation and execution of the dynamic SQL statement can be split into the statements PREPARE and EXECUTE. The dynamic SQL statement is contained in a string which is constructed by the application program. The PREPARE statement initiates the compilation of the dynamic SQL statement, and the EXECUTE statement executes it.

The result of a PREPARE statement is a statement ready for execution. This prepared statement is identified by an SQL statement identifier which is either set by the user as a fixed identifier or is generated by Adabas SQL Gateway Embedded SQL when a host variable has been specified. The prepared statement is kept for later execution. If it is intended that the statement identifier is to be generated by Adabas SQL Gateway Embedded SQL, it is necessary to initialize the variable with blanks or an empty string prior to execution. Otherwise, Adabas SQL Gateway Embedded SQL will use the actual (non-blank) value of the variable. The same statement identifier must then be specified with the EXECUTE statement.

Example:

```
EXEC SQL
PREPARE statement_id FROM :dyn_sql_statement;
EXEC SQL
EXECUTE STATEMENT_ID;
```

where dyn_sql_statement is a character string containing the dynamic SQL statement.

All SQL statements except those mentioned earlier under Limitations can be prepared by the PREPARE statement. Only NON-SELECT statements can be executed by the EXECUTE statement.

It is possible for the dynamic SQL statement to contain host variable markers. For more information, see Using Host Variable Markers later in this section.

Note: *If the string representing the dynamic SQL statement cannot be compiled, the SQLCODE will indicate this error upon return from the PREPARE statement.*

Summary

A program which issues dynamic NON-SELECT statements must include the following steps:

1. Construct the dynamic SQL statement.
   The dynamic SQL statement must be constructed as a character string. The process of creating this string is application-dependent. It may be that the user enters the SQL statement or part thereof directly from a terminal, or that the application program dynamically builds the statement based on other sources of information.
2. PREPARE and EXECUTE the dynamic SQL statement.
   Either EXECUTE IMMEDIATE or PREPARE and EXECUTE can be used to execute the dynamic SQL statement. Variable input values, as specified by a host variable marker "?" may have to be provided by specifying an USING clause and specifying an SQLDA in an EXECUTE statement.
3. Check the result.
   All errors are returned to the application program using SQLCODE in the SQLCA. These errors must be handled like any other error situation.

Select Statements

SELECT statements can only be executed dynamically by using a separate PREPARE statement and the dynamic cursor logic. The statements DECLARE CURSOR, OPEN, FETCH and CLOSE must be used.

There are two ways to execute a SELECT statement dynamically. The method to be used depends on the characteristics of the SELECT statements to be processed:

- If the derived column list of the SELECT statement has a constant format, that is, the number of elements in the resultant table and their data types remain constant, the fixed derived column list method can be used.
- If the derived column list varies, the varying derived column list method must be used. In the latter case, an SQL descriptor area (SQLDA) is required.

Note: *It is not possible to execute dynamically a single-row SELECT.*

Fixed Derived Column List Method

Dynamic SELECT statements with a fixed derived column list produce resultant tables which have a fixed layout, that is, the number of columns is the same and the data type of each column is fixed and known at the time the application program is precompiled.

The fixed derived column list method assumes that the dynamically created SELECT statements have a fixed derived column list, so that a normal FETCH statement can be used to retrieve the rows of the resultant table. This FETCH statement requires that the columns of the resultant table are each assigned to specific hardcoded host variables. As these host variables have to be known at precompilation time, the layout of the derived column list must be determined at the same time. All other clauses of the SELECT statement, the FROM clause, the WHERE clause, etc., can vary dynamically every time the statement is prepared. This means that the fixed derived column list method can be used in those cases where the result and format of the query is known, but the search criteria can vary to such a degree that the rest of the query needs to be constructed dynamically at runtime.

The fixed derived column list method consists of a number of steps:

PREPARE

The entire SELECT statement must be constructed in a host variable which is passed to Adabas SQL Gateway Embedded SQL as a parameter of a PREPARE statement. The application needs to ensure that the resulting format of the query cannot vary dynamically.

Example:

```
EXEC SQL
PREPARE statement_id FROM :dyn_sql_statement;
```

where dyn_sql_statement is a character string containing the dynamic SQL statement.

As an Adabas SQL Gateway Embedded SQL extension, a host variable may be used to identify a statement. If so, Adabas SQL Gateway Embedded SQL returns a unique value in this variable which must have been initialized with blanks upon return from the PREPARE statement. This value is then used for all subsequent references to the prepared statement.

Example:

```
EXEC SQL
PREPARE :statement_id FROM :dyn_sql_statement;
```

DECLARE

The prepared statement must then be associated with a cursor. This can either be achieved explicitly by means of a dynamic DECLARE CURSOR statement or implicitly by an OPEN statement. The dynamic DECLARE CURSOR statement is similar to the static DECLARE, but instead of specifying the SELECT statement, it specifies the statement identifier as defined in the PREPARE statement, thus associating the prepared SELECT statement with the cursor. Such a

DECLARE statement may also be executed prior to the associated PREPARE statement or may be omitted altogether, if the associated OPEN statement specifies the SQL statement identifier instead.

Example:

```
EXEC SQL
DECLARE ABC CURSOR FOR statement_id;
```

Note: *Alternatively, an Adabas SQL Gateway Embedded SQL extension allows a host variable to be used to identify the cursor. This host variable must be initialized with a suitable value by the application program before use.*

Example:

```
EXEC SQL
DECLARE :cursor_name CURSOR FOR STATEMENT_ID;
```

Note: *If in the original PREPARE statement, a host variable was used to express the statement identifier, then a host variable containing the same assigned value must be used here in order to identify the statement. If used at all, the DECLARE statement must be executed after the PREPARE statement.*

Example:

```
EXEC SQL
DECLARE ABC CURSOR FOR :statement_id;
```

It can be seen that the dynamic DECLARE CURSOR statement differs from its normal static counterpart in that during runtime the statement is of significance, that is, the prepared statement is associated to the particular cursor. The order of execution is important in a dynamic SQL application. Once the PREPARE and then the DECLARE CURSOR statements have been successfully executed, other cursor associated statements can be executed in the normal way, except that the cursor may need to be expressed as a host variable. The normal OPEN, FETCH, CLOSE logic is still applicable.

OPEN

The cursor associated with the dynamic SELECT statement is opened by means of an OPEN statement. Note that the cursor name may be expressed as a host variable.

Example:

```
EXEC SQL
OPEN ABC;
```

If the SELECT statement contains host variable markers, the parameters can be submitted by the USING clause or the USING DESCRIPTOR clause. For more information, see Using Host Variable Markers later in this section.

Example:

```
EXEC SQL
OPEN ABC USING :hv1, :hv2;
or using an SQL descriptor area:
EXEC SQL
```

```
OPEN ABC USING DESCRIPTOR :input_sql_da;
```

In addition, an SQL statement identifier can be specified in case the DECLARE CURSOR statement has been omitted.

Example:

```
EXEC SQL
OPEN ABC CURSOR FOR :statement_id;
```

FETCH

As the format of the derived column list of the dynamic SELECT statement is constant, the FETCH statement can be identical to the static case. For each one of the columns in the resultant table, a host variable needs to be specified which is of a compatible data type.

Notes:

1. *Although the format of the derived column list does not vary dynamically, it is still not visible to the Adabas SQL Gateway Embedded SQL. Therefore, the compiler cannot actually check the validity of the FETCH statement and in particular its target buffer list. Naturally, at run time, such checks are performed.*
2. *An attempt to fetch a derived column of type binary, using a dynamically prepared SELECT statement and a FETCH statement which is identical to the static counterpart, will always result in an error condition. This is because, upon precompiling the FETCH statement, the fact that a character host variable is going to be used for the retrieval of a derived column of type binary is not foreseeable. If a derived column of type binary is to be retrieved using a dynamically prepared select statement, even if has a fixed derived column list, then a FETCH statement which uses a descriptor area must be used.*

Example:

```
EXEC SQL
FETCH ABC INTO :hv1, :hv3;
```

CLOSE

The closing of the cursor is identical to the static case. By executing the CLOSE statement, all resources reserved by the cursor are released.

Example:

```
EXEC SQL
CLOSE ABC;
```

Likewise, once closed, the cursor may simply be re-opened again.

Summary

A program which issues dynamic fixed derived column list SELECT statements must include the following steps:

1. Construct the dynamic SELECT statement. The statement is constructed as a character string in a similar fashion to NON-SELECT dynamic statements. However, the derived column list must remain fixed and its format must have been determined at compile time.
2. PREPARE the dynamic SQL statement.
3. Optionally, DECLARE a cursor for the prepared statement using a dynamic DECLARE CURSOR statement.
4. OPEN the cursor in a similar way to a normal static cursor.
5. Variable input values, as specified by a host variable marker "?" may have to be provided by using an USING clause appended to the OPEN statement and specifying an SQLDA.
6. FETCH from the cursor as required until all rows have been processed.

7. CLOSE the cursor.

Varying Derived Column List Method

Dynamic SELECT statements with a varying derived column list are SELECT statements which produce resultant tables which have differing formats, that is, the format of the resultant table is specified dynamically and may vary from instance to instance.

This method is more complicated than the one of using a fixed derived column list but is only required if indeed the format of the possible resultant tables can vary. Otherwise the fixed derived column list method may be used. In order to be able to use the varying list method, the application program must be able to acquire dynamic storage and be able to manipulate pointers or addresses. This obviously limits the use of this method to those host languages which provide these facilities, or specially written subroutines are needed.

The application program needs to get information about the layout of the resultant table for a varying derived column list statement as target buffers must be dynamically provided. Adabas SQL Gateway Embedded SQL provides special functions to aid the application program in this task. This information is passed to the program using an SQL descriptor area or an SQLDA.

PREPARE

The SELECT statement must be constructed in a host variable which is passed on to Adabas SQL Gateway Embedded SQL as a parameter to a PREPARE statement.

Example:

```
EXEC SQL
PREPARE statement_id FROM :dyn_sql_statement;
```

Alternatively, an Adabas SQL Gateway Embedded SQL extension allows a host variable to be used to identify the statement. If so, Adabas SQL Gateway Embedded SQL returns a unique value in this variable which must have been initialized with blanks upon return from the PREPARE statement. This value is then to be used for all subsequent references to the prepared statement.

Example:

```
EXEC SQL
PREPARE :statement_id FROM :dyn_sql_statement;
```

DECLARE

The prepared statement must then be associated with a cursor. This can either be achieved explicitly by means of a dynamic DECLARE CURSOR statement or implicitly by an OPEN statement. The dynamic DECLARE CURSOR statement is similar to the static DECLARE, but instead of specifying the SELECT statement, it specifies the statement identifier as defined in the PREPARE statement, thus associating the prepared SELECT statement with the cursor. Such a DECLARE statement may also be executed prior to the associated PREPARE statement or may be omitted altogether, if the associated OPEN statement specifies the SQL statement identifier instead.

Example:

```
EXEC SQL
DECLARE ABC CURSOR FOR statement_id;
```

Alternatively, an Adabas SQL Gateway Embedded SQL extension allows a host variable to be used to identify the cursor. This host variable must be initialized with a suitable value by the application program before use.

Example:

```
EXEC SQL
DECLARE :cursor_name CURSOR FOR statement_id;
```

Note: *If in the original PREPARE statement, a host variable was used to express the statement identifier, then a host variable containing the same assigned value must be used here in order to identify the statement. If used at all, the DECLARE statement must be executed after the PREPARE statement.*

Example:

```
EXEC SQL
DECLARE ABC CURSOR FOR :statement_id;
```

DESCRIBE

A description of the resulting format of the query may now be retrieved from Adabas SQL Gateway Embedded SQL. This is done using an SQLDA and a DESCRIBE statement. The functionality of the DESCRIBE statement can also be achieved by using an INTO clause in the PREPARE statement.

Example:

```
EXEC SQL
DESCRIBE STATEMENT_ID INTO :output_sqlda;
```

After successful execution of the DESCRIBE statement, the SQLDA contains detailed information concerning the resulting format of the SELECT statement.

The total number of columns and the particular type of each column will be supplied. The application program must act on this information by supplying dynamically an appropriate target buffer for each of the columns. The address of each target buffer must be written into the SQLDA. In addition, an associated indicator value may have to be assigned.

OPEN

The cursor associated with the dynamic SELECT statement is opened by means of an OPEN statement. The cursor name may be expressed as a host variable.

Example:

```
EXEC SQL
OPEN ABC;
```

If the SELECT statement contained host variable markers, the parameters can be submitted by the USING clause or the USING DESCRIPTOR clause. For more information, see Using Host Variable Markers later in this section.

Example:

```
EXEC SQL
OPEN ABC USING :hv1, :hv2;
```

or using an SQL input descriptor area:

```
EXEC SQL
OPEN ABC USING DESCRIPTOR :input_sqlda;
```

In addition, an SQL statement identifier can be specified in case the DECLARE CURSOR statement has been omitted.

Example:

```
EXEC SQL
OPEN ABC CURSOR FOR :statement_id
    USING DESCRIPTOR :input_sqlda;
```

FETCH

The FETCH statement must be executed in conjunction with the SQLDA that has been constructed for this particular dynamic SELECT statement. The resulting values are copied into the locations specified in the corresponding column description in the SQLDA. Note that Adabas SQL Gateway Embedded SQL can only assume that such locations are of sufficient size to accommodate the returned data. It is the responsibility of the application program to provide such locations. Using a DESCRIBE statement greatly simplifies this task.

Example:

```
EXEC SQL
FETCH ABC USING DESCRIPTOR :output_sqlda;
```

CLOSE

The CLOSE statement causes all resources reserved by the cursor to be released.

Example:

```
EXEC SQL
CLOSE ABC;
```

Once closed, the cursor may be re-opened again within the current transaction.

Summary

A program which issues dynamic varying derived column list SELECT statements must include the following steps:

1. Construct the dynamic SELECT statement. The statement is constructed as a character string in a similar fashion to NON-SELECT dynamic statements.
2. PREPARE the dynamic SQL statement.
3. Allocate and build an appropriate SQLDA. This may be done using a DESCRIBE statement. Assign appropriate target buffers.
4. Optionally, DECLARE a cursor for the prepared statement using a dynamic DECLARE CURSOR statement.
5. OPEN the cursor in a similar way to a normal static cursor.
6. Variable input values, as specified by a host variable marker "?" may have to be provided by supplying a USING clause appended to the OPEN statement specifying an SQLDA.
7. FETCH from the cursor as required until all rows have been processed. The output SQLDA must be specified in order to receive retrieved data.
8. CLOSE the cursor.

Using Host Variable Markers

A dynamic SQL statement cannot contain host variables directly. It is, however, possible to provide a dynamic SQL statement after it has been prepared with value parameters at execution time. The dynamic statement must then contain a host variable marker for every host variable. A host variable marker is represented by a question mark (?).

Example:

```
EXEC SQL
PREPARE statement_id FROM "DELETE FROM CRUISE WHERE CRUISE_ID = ?";
EXEC SQL
EXECUTE STATEMENT_ID USING :cruise_id;
```

The dynamic DELETE statement contains one host variable marker, so the USING clause in the EXECUTE statement contains one host variable. The host variable cruise_id is used to provide a parameter for the prepared DELETE statement. It is as if the following static SQL statements were executed:

```
EXEC SQL
DELETE FROM CRUISE WHERE CRUISE_ID = :cruise_id;
```

The host program can re-execute repeatedly the prepared statement by supplying a fresh value in the host variable with each iteration.

Restrictions

In principle, a host variable marker may appear everywhere in a statement where a host variable may appear. Because of the nature of dynamic SQL, however, there are certain restrictions. The following rules apply:

- a host variable marker is not allowed to appear in a derived column list
- only one operand of a diadic arithmetic operator or comparison operator may be a host variable marker, for example, ? = ? or ? * ? is not allowed.
- the first two operands of a BETWEEN or IN operator cannot be host variable markers, for example, ? IN (?,...) is not allowed. However, 5 + ? IN (?,...) is allowed.

The reason for these restrictions is that at the time the dynamic SQL statement is compiled, the data type of each one of the host variable markers needs to be determined. In the cases described above this cannot be done.

Different Methods

As with SELECT statements, there are different methods to deal with host variable markers. One method can be applied in situations where the number of host variable markers is constant and their type is known and also constant. Another method must be applied if the number of host variable markers varies. Both methods are described in the following sections.

Constant Number of Host Variable Markers

When the number and data types of the host variable markers are constant and known at compilation time in a dynamic SQL statement, a matching set of host variables can be defined to be used to provide values prior to the execution of the prepared dynamic statement. These host variables can be specified in the USING clause of either an EXECUTE or an OPEN statement.

NON-SELECT Statements

For NON-SELECT statements, the host variables used to resolve the host variable markers must be specified in the USING clause on the EXECUTE statement. The host variables in the USING clause must be specified in the same order as the host variable markers were specified in the dynamic SQL statement.

Example:

```
EXEC SQL
PREPARE statement_id FROM "INSERT INTO CRUISE VALUES (?,?,?,?,?)";
EXEC SQL
EXECUTE statement_id USING :hv1,:hv2,:hv3,:hv4,:hv5;
```

SELECT Statements

For SELECT statements, the host variables used to resolve the host variable markers must be specified in the USING clause appended to the OPEN statement. The host variables in the USING clause must be specified in the same order as the host variable markers were specified in the dynamic SQL statement.

Example:

```
EXEC SQL
PREPARE statement_id FROM "SELECT CRUISE_ID FROM CRUISE WHERE CRUISE_ID =
?";
EXEC SQL
DECLARE ABC CURSOR FOR statement_id;
EXEC SQL
OPEN ABC USING :hv1;
```

Varying Number of Host Variable Markers

When the number and data types of the host variable markers varies with each dynamically prepared statement and/or their data type cannot be pre-determined, it is not possible to define a matching set of host variables to provide values prior to the execution of the prepared statement.

In that case, the application program needs to get information about the host variable markers in a prepared statement dynamically. The application program can either do this itself by analyzing the dynamic SQL statement, or Adabas SQL Gateway Embedded SQL can provide this information in an SQL descriptor area using a PREPARE or DESCRIBE statement.

Upon return from an appropriate PREPARE or DESCRIBE statement, Adabas SQL Gateway Embedded SQL will have filled the SQLDA with information about each one of the host variable markers. This information can then be used by the application program to allocate and assign host variables for each one of the host variable markers. Note that it is possible at this stage to change the data type description of a host variable in the SQLDA. Be aware that this may lead to runtime errors if the data type of a host variable is changed to one that is incompatible with the one established by Adabas SQL Gateway Embedded SQL.

Note: *Such an input SQLDA is a separate instance of an output SQLDA but has the same structure.*

NON-SELECT Statements

For NON-SELECT statements, the input SQLDA must be supplied with the EXECUTE statement. The host variables described in the SQLDA must be specified in the same order as the host variable markers were specified in the dynamic SQL statement.

Example:

```
EXEC SQL
PREPARE statement_id FROM "INSERT INTO CRUISE VALUES (?,?,?,?,?)";
EXEC SQL
```

```
DESCRIBE statement_id INTO INPUT :input_sqlda;
EXEC SQL
EXECUTE statement_id USING DESCRIPTOR :input_sqlda;
```

SELECT Statements

For SELECT statements, the input SQLDA must be supplied with the OPEN statement. The host variables in the input SQLDA must be specified in the same order as the host variable markers were specified in the dynamic SQL statement.

Example:

```
EXEC SQL
PREPARE statement_id
   FROM "SELECT CRUISE_ID FROM CRUISE WHERE CRUISE_ID = ?";
EXEC SQL
DESCRIBE statement_id INTO INPUT :input_sqlda;
EXEC SQL
DECLARE ABC CURSOR FOR statement_id;
EXEC SQL
OPEN ABC USING DESCRIPTOR :input_sqlda;
```

Summary

A program which issues dynamic statements which contain host variable markers must perform the following steps:

1. Construct the dynamic SQL statement. The dynamic SQL statement must be constructed as a character string, which will contain host variable markers (?).
2. Prepare the dynamic SQL statement. The dynamic SELECT statement always has to be prepared using a PREPARE statement.
3. Establish information about the host variable markers. If the host variable markers are constant in number and data type, host variables may be applied statically. Otherwise an INTO input clause in the PREPARE or DESCRIBE statement must be used in order to obtain the information about the host variable markers. Variables must then be allocated dynamically.
1. Either assign values to any static host variables
2. Or load the input SQLDA. If the host variables are assigned dynamically, the SQLDA has to be supplied with information about them. The host variables themselves must have appropriate values assigned to them

4. Execute the dynamic SQL statement. A USING clause containing either references to the static host variables or the input SQLDA is appended to either the EXECUTE statement or the OPEN statement as required.

QL Descriptor Area (SQLDA)

General Information

An SQL descriptor area is used as a communication area between an application program and Adabas SQL Gateway Embedded SQL for dynamic SQL. It is used for communicating information between Adabas SQL Gateway Embedded SQL and the application program in both directions.

The information for a dynamic SQL statement that can be retrieved from Adabas SQL Gateway Embedded SQL by an application program using an SQLDA originates from either of two sources:

OUTPUT SQLDA

The derived column list of a dynamic SELECT statement. The application program can retrieve information about the layout of the resulting format of a SELECT statement. The information comprises a list of elements where each element describes the corresponding derived column. An SQLDA describing this type of information is called an output SQLDA. The information is assigned to the output SQLDA by either an extended PREPARE statement (example 1a) or a separate DESCRIBE statement (example 1b). The keyword OUTPUT is the default and therefore optional.

Example 1a:

```
EXEC SQL
PREPARE statement_id INTO :output_sqlda FROM dyn_sql_statement;
```

Example 1b:

```
EXEC SQL
PREPARE statement_id FROM :dyn_sql_statement;
EXEC SQL
DESCRIBE statement_id INTO OUTPUT :output_sqlda;
```

INPUT SQLDA

The host variable markers in a dynamic SQL statement.

The application program can retrieve information about all host variable markers used in a dynamic SQL statement. The information comprises a list of elements where each element describes the corresponding host variable marker. An SQLDA describing this type of information is called an input SQLDA. The information is assigned to the input SQLDA by either an extended PREPARE statement (example 2a) or a separate DESCRIBE statement (example 2b). The keyword INPUT is mandatory.

Example 2a:

```
EXEC SQL
PREPARE statement_id INTO INPUT :input_sqlda FROM dyn_sql_statement;
```

Example 2b:

```
EXEC SQL
PREPARE statement_id FROM :dyn_sql_statement;
EXEC SQL
DESCRIBE statement_id INTO INPUT :input_sqlda;
```

Both input and output SQLDAs can be specified in the same PREPARE and DESCRIBE statements if desired. However, one SQLDA cannot be used for both an input and an output SQLDA simultaneously.

Once Adabas SQL Gateway Embedded SQL has filled an SQLDA with this information, the application program must provide a host variable reference for each element. This must be done prior to the execution of the prepared statement.

Corresponding to the two types of SQLDAs, two types of host variable references must be supplied.

- **Target host variables for receiving resultant data**

The elements of an output SQLDA associated with a prepared SELECT statement each describe the expected format of the data to be received. The application program must assign to each element a suitable host variable which is capable of receiving the expected data. Adabas SQL Gateway Embedded SQL can now determine where to copy the resulting data by means of the pointer reference in each element. Such an output SQLDA is only used in conjunction with a FETCH statement.

Example 1:

```
EXEC SQL
FETCH ABC USING DESCRIPTOR :output_sqlda;
```

- **Host variables as host variable marker replacements**

The elements of an input SQLDA each describe the expected format of any additional parameters required by the prepared statement as represented by host variable markers. The application program must assign a suitable host variable to each element of the input SQLDA and each host variable must be loaded with the desired value before execution of the prepared statement. Such an input SQLDA is used in conjunction with either an OPEN statement (example 2a) or an EXECUTE statement (example 2b).

Example 2a:

```
EXEC SQL
OPEN ABC USING DESCRIPTOR :input_sqlda;
```

Example 2 b:

```
EXEC SQL
EXECUTE statement_id USING DESCRIPTOR :input_sqlda;
```

The SQLDA Structure

Exactly the same structure is used for input and output SQLDA. It consists of two distinct parts:

- A header containing general information about the prepared statement,
- A consecutive list of elements corresponding to fields in the derived column list or the host variable markers.

The structure consists of the four fields of the SQLDA header immediately followed by as many occurrences of the sqlvar structure as stated in the sqln field.

Field	Description
sqldaid	An eight-byte character string containing the constant SQLDA which serves as an eye catcher for easier memory dump interpretation.
sqldabc	A four-byte integer field of type SAG_INTEGER containing the total length of the SQLDA in bytes, that is, the length of the header plus the length of the variable descriptor elements multiplied by the number of available elements (sqln).

sqln	A two-byte integer field containing the total number of variable descriptor elements allocated in the SQLDA.	
sqld	A two-byte integer field containing the total number of variable descriptor elements returned during the execution of a DESCRIBE statement.	
sqlvar	An array containing sqln variable descriptor elements.	
	sqltype	A four-byte integer field of type SAGTYPE containing the data type of the required/specified host variable and whether or not there is an INDICATOR variable present.
	sqllen	A four-byte integer field of type SAG_INTEGER containing the length of the required/specified host variable. The interpretation of this field depends on the data type.
	reserved	A four-byte integer field of type SAG_INTEGER required for internal purposes.
	internal	A two-byte integer field required for internal purposes.
	sqlindlen	A two-byte integer field containing the length of the host variable acting as an indicator value, if one is required
	sqlindtype	A four-byte integer field containing the data type of the host variable acting as an indicator value, if one is required

	sqlind	A structure of type SAGPointer containing the pointer to the host variable acting as an indicator value, if one is required.
	sqldata	A structure of type SAGPointer containing the pointer to the host variable which is to receive or which contains the data.
	sqlname	A structure of type SAGCOLUMN containing the derived column name of the resulting column, the length of the column name and the column type.

The sqlname field is only relevant for an output SQLDA and only in the particular case of the corresponding derived column having a derived column label.

The sqltype field is set by Adabas SQL Gateway Embedded SQL to reflect the particular type of the required field.

The sqllen field is also set by Adabas SQL Gateway Embedded SQL depending on the value assigned to the sqltype field. This field specifies the required size of the host variable.

The sqltype field also specifies whether or not a indicator variable is required or is supplied. This is shown by the type value being negated.

Declaring an SQLDA

The SQLDA is a special type of host variable structure. To ensure that the structure has the correct format, the application program should use the definition of the SQLDA provided by Adabas SQL Gateway Embedded SQL. To facilitate this, an SQL statement like the following one should be embedded in the application.

```
EXEC SQL
INCLUDE SQLDA AS sqlda_ptr;
```

This statement has the effect of generating a declaration of a variable sqlda_ptr at the point where it is specified. This variable can then be used as a pointer to a descriptor area.

Allocating an SQLDA

When using an SQLDA to retrieve descriptive information from Adabas SQL Gateway Embedded SQL either for input or output purposes, the application program normally does not know the number of variable descriptions required. The application program, however, has to allocate an SQLDA of a certain dimension before the PREPARE or DESCRIBE statements can be issued. In general, there are two techniques which can be used:

- The application program allocates an SQLDA of maximum size required for the maximum possible number of derived column list elements or host variable markers. This might cause a significant waste of storage if the maximum has to be set very high.
- The application program allocates an SQLDA of minimum size. The dimension of the SQLDA is determined by the sqln element in the SQLDA header. If the number of derived column list elements or host variable markers exceeds this number, Adabas SQL Gateway Embedded SQL will refrain from attempting to provide information on the remaining elements or markers. Adabas SQL Gateway Embedded SQL, however, does return the correct number of elements in the sqld element of the SQLDA. The application program can then use this number to allocate a new SQLDA of sufficient size and re-issue the PREPARE or DESCRIBE statement. The application program must explicitly have an SQLDA declaration such that the resulting structure is in scope for all SQL statements which access it. Such a declaration does not need to be in a BEGIN DECLARE SECTION.

Determining the Type of SQL statement

Although the SQLDA does not return explicitly the SQL statement type, enough information is returned in the SQLDA for the application program to determine whether or not the dynamic statement is a SELECT statement. If the field sqln is 0, the statement did not contain a derived column list and must therefore be a NON-SELECT statement.

Host Variable Specification

Host variables serve as a data exchange medium between Adabas SQL Gateway Embedded SQL and the application program written in a host language. When used in an SQL statement, a host variable specification has one of the following purposes:

- to identify a variable in the host language program which is to receive a value(s) from Adabas SQL Gateway Embedded SQL.
- to identify a variable in the host language program which is to pass a value(s) to Adabas SQL Gateway Embedded SQL.

A host variable is a single variable or structure declared in the host program.

A host variable identifier is used to identify a single host variable or structure from within an SQL statement.

A host variable specification consists of a host variable identifier and an associated optional INDICATOR variable and defines either a single variable, a structure, or an element in a structure.

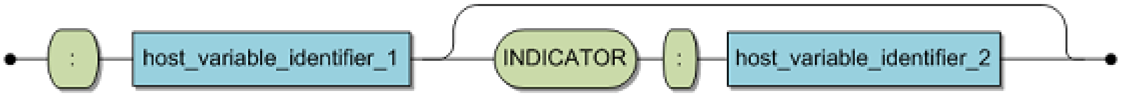
This section contains the following topics:

- Single Variables
- INDICATOR Variables
- Host Variable Markers
- Host Structures

Single Variables

The identified single host variable may actually be a single element within a host variable structure. Such a reference is not permitted in ANSI compatibility mode.

A single host variable is identified by a host variable identifier which has the following syntax:



host variable identifier 1	Identifies a single variable which is assigned any value but the NULL value.
host variable identifier 2	Identifies an INDICATOR variable, see INDICATOR Variables below.

Example:

Select the price of the cruise with a cruise ID of 5064 into a host variable.

```
SELECT cruise_price
   INTO :host_variable1
      FROM cruise
      WHERE cruise_id=5064;
```

INDICATOR Variables

An INDICATOR variable can serve as one of two purposes:

- Signifies the presence of a NULL value in a host variable assignment. If the NULL value is to be assigned to a target host variable specification then an accompanying INDICATOR variable must be present and is assigned a negative value to signify the NULL value. If the NULL value is to be assigned and the INDICATOR variable is missing, then a runtime error will occur.

The INDICATOR variable must be of a numeric data type with the exception of double precision, real and floating point data types. It must be of the appropriate data type for the host language.

Example:

Select the cancellation date of Contract 2025 into a host variable. (The column 'date_cancellation' could contain NULL values)

```
SELECT date_cancellation
   INTO :host_variable1 INDICATOR :host_variable2
      FROM contract
         WHERE contract_id=2025 ;
```

- Signifies that truncation has occurred in a host variable assignment. If truncation occurred during the assignment of a character string to a host variable, then the INDICATOR variable will show the total number of characters in the originating source prior to truncation.

SUMMARY:

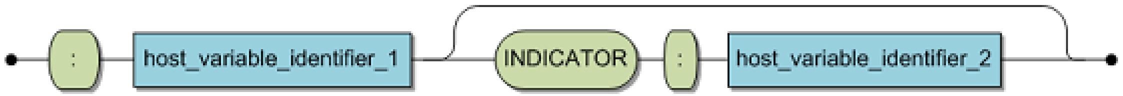
Indicator Value	Meaning	Host Variable Name
<0	indicates NULL value	undefined
=0	indicates non-NULL value	actual value
>0	number of characters	actual value in originating source

Host Variable Markers

A dynamic SQL statement can not contain host variables directly. It is, however, possible to provide a dynamic SQL statement after it has been prepared with value parameters at execution time. The dynamic statement must then contain a host variable marker for every host variable specification. A host variable marker is represented by a question mark (?) in the statement's source text. For details, see the section on Dynamic SQL.

Host Structures

A host structure is a C structure or a COBOL group that is referenced in an SQL statement. The exact rules to which a host structure must conform are described in the host language sections of the Programming Guide.



host variable identifier 1	Identifies a host structure. It can only be specified in the INTO clause of a single row SELECT or FETCH statement. A reference to a host structure is equivalent to a reference to each element in that structure.
	Each element of the host structure identified by host variable identifier 1 is a host variable which is assigned a value, if that value is not the NULL value.
host variable	An INDICATOR structure. An INDICATOR structure is a host

identifier 2	structure consisting of elements each identifying an INDICATOR variable.
	Each element of the INDICATOR structure identified by host variable identifier 2 identifies an INDICATOR variable, see also INDICATOR Variables in this section.

The i th element in the host structure indicated by host variable identifier 2 is the INDICATOR variable for the i th element in the host structure indicated by host variable identifier 1.

Note: Pointer expressions will be supported in the next release version.

Assume the number of elements in the host structure identified by host variable identifier 1 is m and the number of elements in the host structure identified by host variable identifier 2 is n:

- If m > n, then the last m-n elements in the host structure identified by host variable identifier 1 do not have an INDICATOR variable.
- If m < n, then the last n-m elements in the host structure identified by host variable identifier 2 are ignored.

Examples:

If two host structures have been declared, one for actual returned values and one for indicator values, and the variables 'struct1' and 'indicator1' identify these structures respectively, then the following syntax shows how values from a derived column list are entered into host variables (assuming that the host structures match the derived columns).

```
SELECT cruise_identifier,start_date,cruise_price
   INTO :struct1 INDICATOR :indicator1
      FROM cruise;
```

The following example inserts a resulting value from a query into a particular 'Element' of a defined structure. 'struct1' is a structure identifier that contains an element identified by 'price_element' and 'indicator1' is a structure identifier that contains the element identified by 'price_ind'.

```
SELECT cruise_price
   INTO :struct1.price_element INDICATOR :indicator1.price_ind
      FROM cruise;
```

SQL Statements in C

The SQL Terminating Delimiter

The end of a SQL statement is determined explicitly by a terminating delimiter. For C, this is a semicolon (;).

Host Language	Terminating Delimiter
C	semicolon (;)

Comments within an SQL Statement

Two types of comments are supported, host language comments and SQL comments:

- Host language comments obey the rules determined by the host language.

  Host language comments may be positioned anywhere within an SQL statement where a whitespace character can appear. For examples of host language comments, refer to the host language dependent sections later in this documentation.

  Note: Host language comments are not permitted between the following keywords:

  ```
  EXEC SQL
  BEGIN DECLARE SECTION
  END DECLARE SECTION
  ```

- SQL comments are character string preceded by two minus characters (- -).

  SQL comments may be positioned anywhere within the SQL statement body where a whitespace character can appear. All characters following this starting delimiter until the end of the line are interpreted as part of the comment.

If a host languages does not permit nested comments, modify the host language comment delimiters within a statement so the actual SQL statement is commented out; this will prevent nested comments.

Host Variables

C host variables used in SQL statements must be declared within the SQL BEGIN DECLARE SECTION and END DECLARE SECTION statements. Adabas SQL Gateway Embedded SQL allows the use of single host variables and host variable structures.

Host Variable Declaration

C structures are named sets of C single host variables and must conform to the ANSI Standard (X3.159-1989) for C. The use of C structures within SQL statements is an Adabas SQL Gateway Embedded SQL extension and not part of the ANSI SQL Standard. It is not possible to use a union of host variables. The use of the enum type is also not possible.

Binary Data Type

There is no intrinsic binary data type in the C host language. In order to retrieve and supply binary data using host variables, the intrinsic C data type of character must be mapped to the Adabas SQL Gateway Embedded SQL data type of binary. A pseudo type has been introduced using an Adabas SQL Gateway Embedded SQL macro. Therefore, host variables which are to be used for binary data type transportation, must be declared as SAGBINARY. The Adabas SQL Gateway Embedded SQL will then associate such variables with the type binary. Such variables can, therefore, only be used as binary host variables. The type declaration, is indeed a C macro, provided by the Adabas SQL Gateway Embedded SQL. A SAGBINARY variable is declared as follows :

```
EXEC SQL BEGIN DECLARE SECTION ;
SAGBINARY ( hv_name, x ) ;
EXEC SQL END DECLARE SECTION ;
```

where hv_name is the name of the host variable and x is the length in bits of the variable.

The macro is resolved as follows :

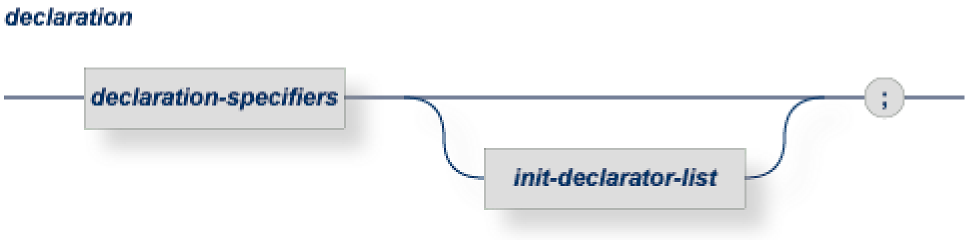
```
char hv_name [ y ] ;
```

where y is the required length in bytes.

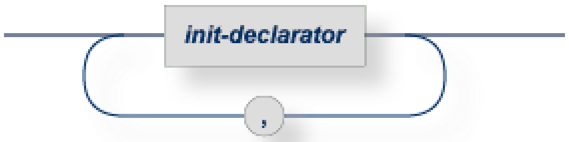
Binary host variables are not subject to byte swapping, nor are they subject to any ASCII/ EBCDIC conversion. No string terminator is to be provided for binary host variables. Therefore, the direct binary contents of a host variable will be received by Adabas SQL Gateway Embedded SQL, with each element of the character array representing a full 8 bytes.

The pre-compiler must run prior to the C macro processor. No corresponding functionality is supported for the C language when using dynamic SQL and the SQLDA.

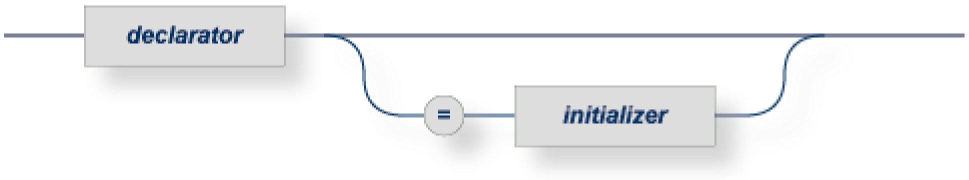
Syntax



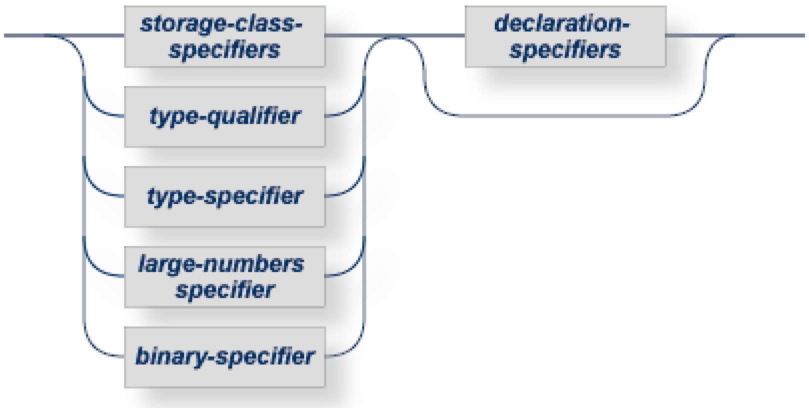
init-declarator-list



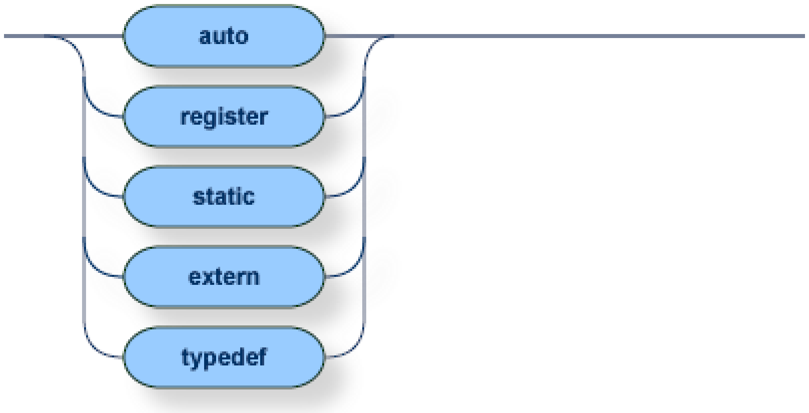
init-declarator



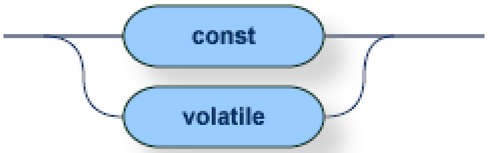
declaration specifiers



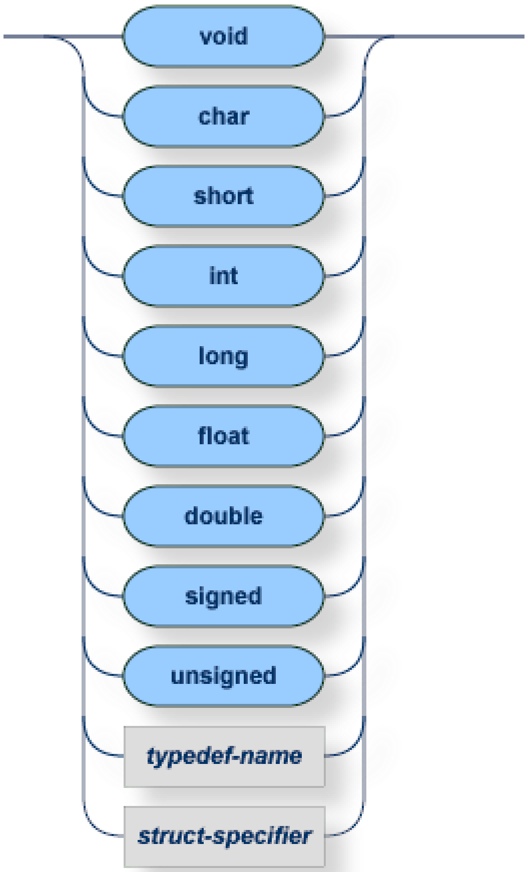
storage-class-specifier



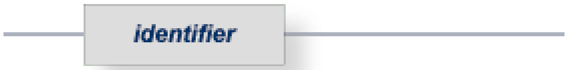
type-qualifier



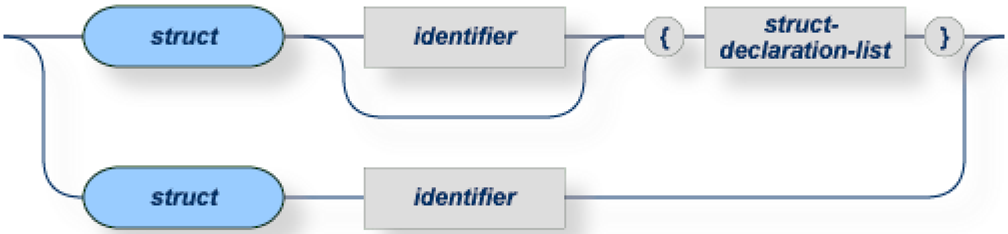
storage-class-specifier



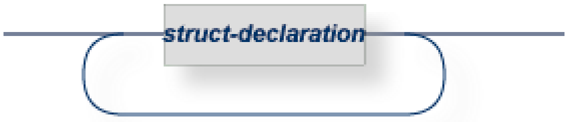
typedef-name



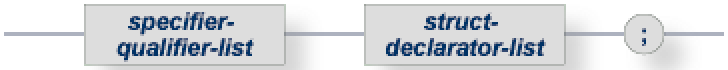
init-declarator



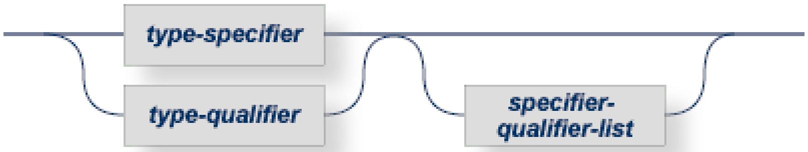
struct-declaration-list



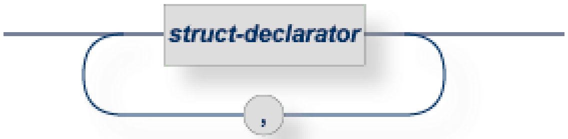
struct-declaration



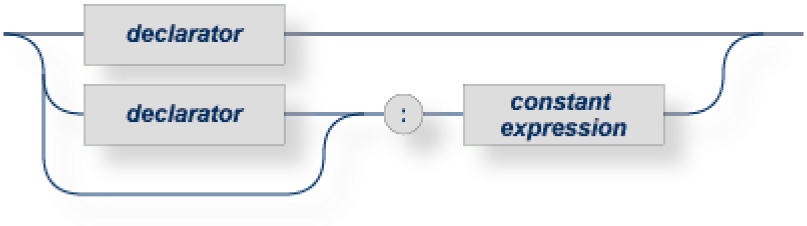
specifier-qualifier-list



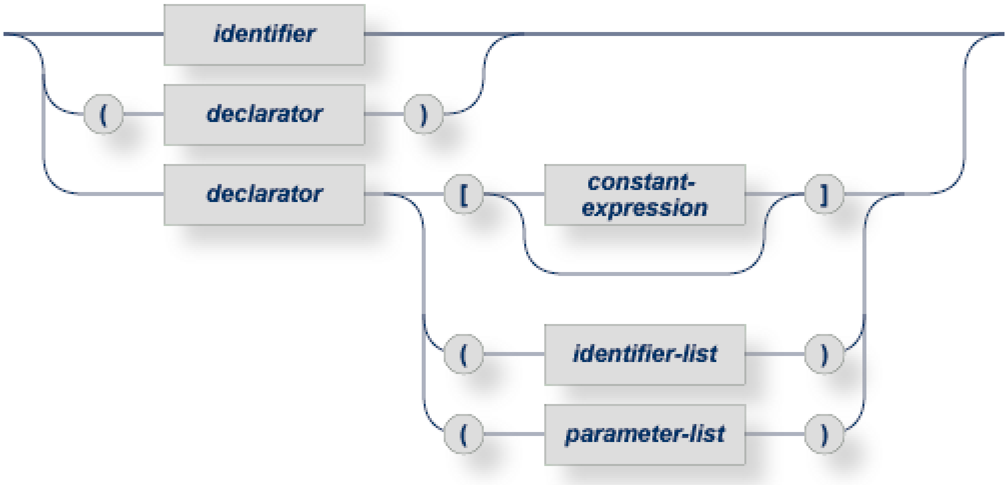
struct-declarator-list



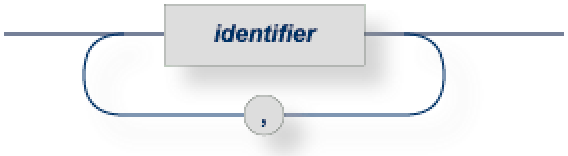
struct-declarator



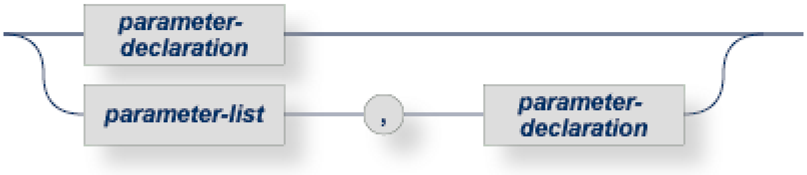
declarator



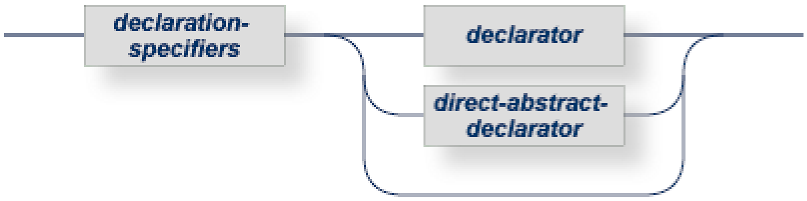
identifier-list



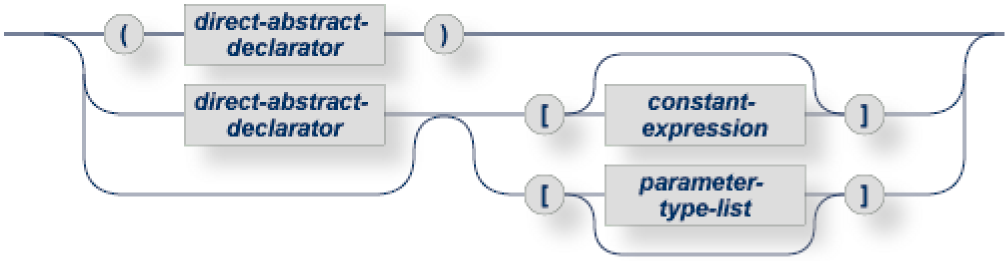
parameter-list



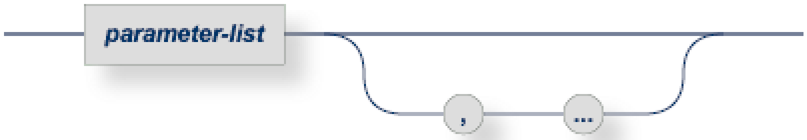
parameter-declaration



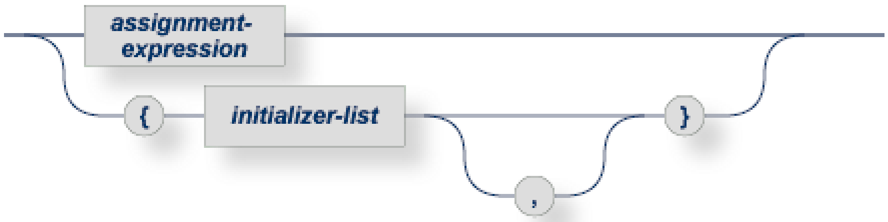
direct-abstract-declarator



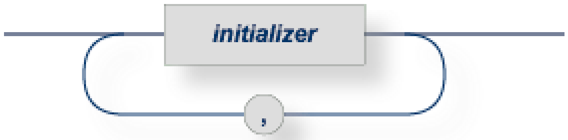
parameter-type-list



initializer



initializer-list



Within embedded SQL statements the C naming qualification rules for structure and array elements are as defined in the ANSI standard (X3.159-1989) for C. There may be any number of SQL BEGIN DECLARE SECTIONs.

Ambiguous References and Multiple Declarations

A declaration that appears more than once with the same identifier is called a multiple declaration. If a host variable refers to such a multiple declaration, and the different declarations are of different types, an error occurs. Otherwise the host variable is accepted.

Data Type Conversion

The following table shows the conversion of C data types to SQL data types:

C Data Types	SQL Data Types
char (array)	CHARACTER
long (signed or unsigned)	INTEGER
int (signed or unsigned)	INTEGER
float	REAL
double	DOUBLE PRECISION
short (signed or unsigned)	SMALLINT
long long	BIGINT

For more details on SQL data types and their usage in SQL statements refer to the section on **Common Elements** in this online help file.

The following table shows the conversion of SQL data types to C data types:

SQL Data Types	C Data Types
CHAR (more than one character)	char (array)
BINARY	SAGBINARY (char array)
INTEGER	long
SMALLINT	short
REAL	float
DOUBLE PRECISION	double
FLOAT	float
DECIMAL	float, double or long
NUMERIC	float, double or long
BIGINT	long long

Embedding SQL Statements in C

General Rules in C

SQL Statement Delimiters

SQL statements are delimited by the prefix EXEC SQL and a semicolon (;) which acts as the terminator. The prefix may be written in upper or lower case letters.

SQL Statement Placement

SQL statements may be specified wherever a C statement may be specified within a C function block, as the Adabas SQL Gateway Embedded SQL compiler replaces them with generated C statements. Included C source code must not contain any SQL statements nor any host variable declaration for use in SQL statements. Similar restriction apply to C macro bodies.

The INCLUDE SQLCA statement may be positioned anywhere a C variable declaration could be positioned. As this statement results in a declaration of an SQLCA structure, it must be positioned to be in scope for any statement using this SQLCA declaration. The C scoping rules apply.

The SQL WHENEVER statement may be coded anywhere in the C program.

Comments

SQL statements may contain C comments wherever a blank is permitted, C++ comments, or SQL comments which extend to the end of the current source line. Comments are not allowed in strings and may not be nested.

C Example:

```
EXEC SQL WHENEVER SQLERROR
/*                  CONTINUE */
                    GOTO HANDLE-ERROR;
```

C++ Example:

```
EXEC SQL WHENEVER SQLERROR
// CONTINUE */
```

SQL Example:

```
EXEC SQL WHENEVER SQLERROR
                -- CONTINUE
                   GOTO HANDLE-ERROR;
```

Error Handling in C

Textual error messages associated with each error number may be retrieved using the SQL statement GET DIAGNOSTICS and GET DIAGNOSTICS EXCEPTION. These can be declared in a function which is called in the event of a non-zero SQLCODE being returned.

The C program must declare a character variable to receive the error text, an integer variable containing the length of the error text, an integer variable containing the total number of error conditions available and an integer variable containing the current error condition. These 4 items must be declared in a DECLARE SECTION which is in scope whenever the above SQL statements are called.

A programming example using GET DIAGNOSTICS / GET DIAGNOSTICS EXCEPTION looks like this:

```
/* --------------------------------------------------- */

void doERROR()
{
  EXEC SQL BEGIN DECLARE SECTION;
  char errBuf[ 513 ];  /* Error Text Buffer           */
  int  errLen = 513;   /* Length of Error Text Buffer */
  int  conditionCount; /* Count of error conditions   */
  int  errNumber;      /* Current error condition     */
  EXEC SQL END DECLARE SECTION;
/* Obtain the count of error conditions to be returned in
   conditionCount */
  EXEC SQL
    GET DIAGNOSTICS :conditionCount = NUMBER;
/* Obtain each error condition text conditionCount times */
/* The error condition text will be returned in errBuf   */
  for( errNumber = 1;
       errNumber <= conditionCount;
       errNumber++)
  {
    EXEC SQL
      GET DIAGNOSTICS EXCEPTION :errNumber
                                :errBuf = MESSAGE_TEXT,
                                :errLen = MESSAGE_LENGTH;
    printf( "ERR MSG: %s\n\n", errBuf );
  }
}
```

where:

conditionCount is the count of error conditions available, returned by GET DIAGNOSTICS

errNumber is the current error condition.

errBuf is the target buffer, null-terminated on return from GET DIAGNOSTICS EXCEPTION

errLen is the length of the target buffer errBuf.

SQL Communication Area (SQLCA)

The SQLCA provides the programmer with comprehensive information about the success or failure of each SQL command.

The following is the declaration of the SQLCA structure in C:

```
struct sqlca
 { unsigned char sqlcaid [8];   /* eye catcher 'sqlca'       */
   SAG_INTEGER   sqlcabc;       /* size of SQLCA in bytes    */
   SAG_INTEGER   sqlcode;       /* SQL return code           */
   short         sqlerrml;      /* length of error message   */
   unsigned char sqlerrmc [70]; /* error message             */
   unsigned char sqlerrp [8];   /* internal error info       */
   SAG_INTEGER   sqlerrd [6];   /* internal error info       */
   unsigned char sqlwarn [8];   /* warning flags             */
   unsigned char sqlext [8];    /* reserved                  */
 };
```

SQL Descriptor Area (SQLDA)

The SQLDA provides the programmer with comprehensive information about each resulting column of a dynamic SELECT statement.

The following is the declaration of the SQLDA structure in C:

```
struct sqlda
 {
   unsigned char sqldaid [8];   /* eye catcher: 'SQLDA'      */
   SAG_INTEGER   sqldabc;       /* size of sqlda in bytes    */
   short         sqln;          /* #sqlvar elements allocated*/
   short         sqld;          /* #sqlvar elements returned */
   struct sqlvar
   { SAGTYPE     sqltype;       /* datatype of variable      */
     SAG_INTEGER sqllen;        /* length of variable        */
     SAG_INTEGER reserved;      /* reserved                  */
     short       internal;      /* reserved                  */
     short       sqlindlen;     /* length of indicator       */
     SAG_INTEGER sqlindtype;    /* datatype of indicator     */
     SAGPointer  sqlind;        /* pointer to indicator      */
     SAGPointer  sqldata;       /* pointer to data           */
     SAGCOLUMN   sqlname;       /* name of the column or HV  */
   } sqlvar [1];
 };
#define SQLDASIZE(n) (sizeof(struct sqlda)+(n-1)*sizeof(struct sqlvar))
```

SQL Data Type	C Definitions for Data Types Returned or Set in sqltype and sqlindtype	Value
UNKNOWN	SQL_TYP_UNKNOWN	0
CHAR	SQL_TYP_CHAR	1
NUMERIC	SQL_TYP_NUMERIC	2
DECIMAL	SQL_TYP_DECIMAL	3
INTEGER	SQL_TYP_INTEGER	4
SMALLINT	SQL_TYP_SMALLINT	5
FLOAT	SQL_TYP_FLOAT	6
LARGEINT	SQL_TYP_LARGEINT	9
VARCHAR	SQL_TYP_VARCHAR	12
NUMERIC SIGNED LEADING	SQL_TYP_NUMERIC_LD	20
NUMERIC SIGNED TRAILING	SQL_TYP_NUMERIC_TR	21
NUMERIC SIGNED LEADING SEPARATE	SQL_TYP_NUMERIC_SLD	22
NUMERIC SIGNED LEADING TRAILING	SQL_TYP_NUMERIC_STR	23
NULLABLE_CHAR	SQL_TYP_NCHAR	-1
NULLABLE NUMERIC	SQL_TYP_NNUMERIC	-2
NULLABLE DECIMAL	SQL_TYP_NDECIMAL	-3
NULLABLE INTEGER	SQL_TYP_NINTEGER	-4
NULLABLE SMALLINT	SQL_TYP_NSMALLINT	-5

NULLABLE FLOAT	SQL_TYP_NFLOAT	-6
NULLABLE LARGEINT	SQL_TYP_NLARGEINT	-9
NULLABLE VARCHAR	SQL_TYP_NVARCHAR	-12
NULLABLE NUMERIC SIGNED LEADING	SQL_TYP_NNUMERIC_LD	-20
NULLABLE NUMERIC SIGNED TRAILING	SQL_TYP_NNUMERIC_TR	-21
NULLABLE NUMERIC SIGN LEADING	SQL_TYP_NNUMERIC_SLD	-22
NULLABLE NUMERIC SIGN TRAILING	SQL_TYP_NNUMERIC_STR	-23
BINARY	SQL_TYP_BINARY	-51
NULLABLE BINARY	SQL_TYP_NBINARY	-52
NATURAL DATE	SQL_TYP_NATDATE	-53
NULLABLE NATURAL DATE	SQL_TYP_NNATDATE	-54
NATURAL TIME	SQL_TYP_NATTIME	-55
NULLABLE NATURAL TIME	SQL_TYP_NNATTIME	-56
SQLDA	SQL_TYP_SQLDA	-57
NATURAL TIMESTAMP	SQL_TYP_NATTIMESTAMP	-58
NULLABLE NATURAL TIMESTAMP	SQL_TYP_NNATTIMESTAMP	-59

SQL Descriptor Area (SQLDA)

Encoding and using the SQLLEN field

```
SQLDA - Content of SQLLEN
------------------------
The sqllen field of the sqlvar contains different values based on the
sqltype of the column.
For all datatypes except DECIMAL and NUMERIC the value contained in
sqllen is the actual length of the column.
The sqllen for DECIMAL and NUMERIC types, contains the length, precision
(identical to length) and scale. This can be decoded into length,
precision and scale variables by using the macro SQLDAGET_PREC_SCALE_LEN.
The sqllen can be encoded with length, precision and scale by using the
macro SQLDAPUT_PREC_SCALE_LEN.
The following SQLTYPES require decoding or encoding of the sqllen :
  SQL_TYP_DECIMAL
  SQL_TYP_NUMERIC
  SQL_TYP_NUMERIC_LD
  SQL_TYP_NUMERIC_TR
  SQL_TYP_NUMERIC_SLD
  SQL_TYP_NUMERIC_STR
  SQL_TYP_NUMERIC_BINARY
  SQL_TYP_NUMERIC_BIN_BE
  SQL_TYP_NDECIMAL
  SQL_TYP_NNUMERIC
  SQL_TYP_NNUMERIC_LD
  SQL_TYP_NNUMERIC_TR
  SQL_TYP_NNUMERIC_SLD
  SQL_TYP_NNUMERIC_STR
  SQL_TYP_NNUMERIC_BINARY
  SQL_TYP_NNUMERIC_BIN_BE
The sqllen field in the sqlvar should remain encoded and not be modified
therefore, if the length of the column is required, when allocating
storage for example, the decoded values sqlprec or sqllength should be
used and not the sqllen itself.
SAMPLE USAGE:
-------------
/*
  Show how to decode sqllen for DECIMAL and NUMERIC type columns
  The SQLDAGET_PREC_SCALE_LEN and SQLDAPUT_PREC_SCALE_LEN macros are
  generated automatically by the Precompiler.
*/
```

```
  int   sqlprec;   /* Precision */
  int   sqlscale;  /* Scale     */
  int   sqllength; /* Length    */
  if ( Sqlda->sqlvar[n].sqltype == SQL_TYP_NDECIMAL
                              || SQL_TYP_NNUMERIC
                              || SQL_TYP_NNUMERIC_LD
                              || SQL_TYP_NNUMERIC_TR
                              || SQL_TYP_NNUMERIC_SLD
                              || SQL_TYP_NNUMERIC_STR
                              || SQL_TYP_NNUMERIC_BINARY
                              || SQL_TYP_NNUMERIC_BIN_BE)
  {
    SQLDAGET_PREC_SCALE_LEN(Sqlda->sqlvar[n].sqllen,
                        sqlprec,
                        sqlscale,
                        sqllength);
  }
/*
  Allocate storage using sqllength
*/
  Sqlda->sqlvar[n].sqldata.host_address = calloc(1, sqllength);
```

Windows

Invocation and Precompiler Options - (Windows - C)

The general format of the call to the C precompiler is:

```
acepcc [<precompiler options>] <source file>
```

The C source file can have any extension. If no extension is supplied the default extension 'pcc' will be used. If the source file cannot be located, an error will be returned.

Options

Every option begins with a minus sign (-). The names of the options are not case-sensitive.

The following C precompiler options are available:

Option and Option Name	Description	Default
-a catalog name	If a table name exists multiple times in the CDD and is not fully qualified with CATALOG.SCHEMA.TABLE, an 'ambiguous table reference' error will occur. The default catalog name can be supplied with '-a <catalog_name>'. The catalog name must exist in the CDD.	
-b suppress trailing blanks	Suppress character string trailing blanks. Character string Host-Variables used for input or output are padded with trailing blanks as required. Trailing blanks can be suppressed with '-b'. This is especially useful for applications written in C/C++. where it is normal practice for character strings to be NULL terminated without trailing blanks padding.	no
-c compatibility mode	The Precompiler executes in extended ACE mode. ANSI mode can be enabled with '-c ANSI'. This option affects messages only. In ANSI mode, SQL syntax which does not adhere to ANSI standards will be flagged as a warning. Code generation is not affected. Values : ACE / ANSI	ace
-e error listing file	Error, Warning and Statistics messages are written to STDOUT.	<basename>.plc (Windows and Open

	These messages can be routed to an output file with '-e <error listing file>'. The <error listing file> name must consist of at least a basename and optionally an extension. If no extension is supplied, the default extension 'pcl' will be used. On z/OS platforms, a DDNAME must be used instead of a filename, using the format : -e //ddn:ddname The DDNAME assignment must be made to a physical file. It must not be assigned to // DD SYSOUT=	Systems) DD=SYSTERM (z/OS)
-f code output file	The <code output file> name must consist of at least a basename and optionally an extension. If no extension is supplied, the default extension 'cbl' will be used.	<basename>.c
-k keep the generated output file in case of compilation errors	The <code output file> is deleted after compilation errors. The <code output file> can be kept with '-k'.	no
-l generate line information	Line information in the form : /* line nn "<source file name>" */ is generated into the <code output file> with '-l'. This can be useful when you want to know the line number of the associated SQL Statement in the original C/C++ source file.	no
-m migration check of SQL statements only	The Precompiler executes in code generation mode. When migrating or checking existing C/C++ or COBOL programs, it can be useful to have a listing of the SQL statements only. These are output to STDOUT. No output file is generated. The migration check can be enabled with '-m'.	no
-n Adabas SQL Gateway server name	This option is REQUIRED Server <server_name> to which the session is to connect. <server_name> need not be enclosed in quotes.	no default
-o output code mode	Output code mode is either 32 or 64-Bit The C/C++ Precompiler generates 32-Bit code when invoked on 32-Bit systems (-o 32) and 64-Bit code when	32 (when invoked on 32-Bit systems) 64 (when invoked on 64-Bit systems)

	invoked on 64-Bit systems (-o 64). Generation of code suitable for a different environment can be enabled with '-o 32' or '-o 64'. Values : 32 / 64	
-q no SQL validation	The Precompiler performs SQL syntax validation at precompile time. SQL validation can be disabled with '-q'.	no
-r length of output line	The Precompiler generates code output files with a line length of 72. The output line length can be increased to the length defined in <output line length>. The output line length cannot be defined shorter than the default length for the appropriate Host Language. If this is attempted, the output line length will be reset to the appropriate Host Language default and a warning message issued. Values : >= 72	72
-s schema name	The SCHEMA name used by the Precompiler is derived from <user_name> as supplied with the '-u' option. The default schema name can be overridden with '-s <schema_name>'. The schema name must exist in the CDD.	user_name
-u user name [, password]	This option is REQUIRED User <user_name> and Password <password> to be connected to the Server defined with '-n'. The <password> parameter is optional, however, this must be supplied if the user being connected requires a password.	no default
-w working directory	Input and output files are searched for or stored in the current directory. This can be overridden with '-w <working directory>'. If no filename is given for a file to be generated, the basename of the precompiler source file will be taken, with the appropriate extension.	current directory
-z suppress warnings	Warning messages are generated by the Precompiler. Warning messages can be suppressed	no

	with '-z'. This option affects WARNING messages only. ERROR messages cannot be suppressed.	
-@ file containing command line options and parameters	Command line options and parameters saved in a file can be optionally input to the Precompiler with '-@ <command line options and parameters file>'. The options and parameters in the <command line options and parameters file> can be combined with other command line options when invoking the Precompiler. On z/OS platforms, a DDNAME must be used instead of a filename, using the format : -@ //ddn:ddname The DDNAME assignment must be made to a physical file. It must not be assigned to // DD *	no default

Note :
The minimum required options are server name (-n) and user name (-u). All other options are optional.

Filename Conventions

- The C precompiler source file name must consist of at least a basename and optionally an extension. If no extension is supplied the default extension 'pcc' will be used. If the source file cannot be located, an error will be returned.
- The C precompiler generated output file name must consist of at least a basename and optionally an extension. If no extension is supplied the default extension 'c' will be used.
- The (optional) C precompiler listing file name must consist of at least a basename and optionally an extension. If no extension is supplied the default extension 'pcl' will be used.
- If no directory is given, then the file will be searched or stored in the current directory. If a working directory is set (W option), then all generated files will be stored there. This setting can be overridden by a directory indication within a certain filename. If no filename is given for a file to be generated, then the basename of the precompiler source file will be taken, with the appropriate extension.

Examples:

```
acepcc -nDD=MYDSN -uMYUSER precapp.pcc
```

All generated files will be stored in the current directory. The code output file (C source) is assigned the name precapp.c.

```
acepcc -nDD=MYDSN,GATEWAY=REMOTEHOST,PORT=7500 -uMYUSER precapp.pcc
```

All generated files will be stored in the current directory. The code output file (C source) is assigned the name precapp.c.

A remote connection will be made to the CONNX JDBC Server, listening on PORT 7500.

```
acepcc -nDD=MYDSN -uMYUSER -wC:\TEMP -fD:\CSRC\MYSRC.CPP precapp.pcc
```

This precompiler call stores all generated files in c:\temp except the generated C source. This will be stored in d:\csrc with the name mysrc.cpp. The name of the source file of the precompiler is precapp.pcc.

Libraries

To build an executable program (or a dynamic link library), only the rciclnt.lib must be linked to the objects which are the result of preceding executions of a C compiler.

Example:

```
cl precapp.c c:\connx32\precompiler\rciclnt.lib
```

Unix

Invocation and Precompiler Options (Unix - C)

The general format of the call to the C precompiler is:

```
ACEPCC [<precompiler options>] <source file>
```

The C source file can have any extension. If no extension is supplied the default extension 'pcc' is used. If the source file cannot be located, an error is returned.

Options

Every option begins with a minus sign (-). The names of the options are not case-sensitive.

The following C precompiler options are available:

Option and Option Name	Description	Default
-a catalog name	If a table name exists multiple times in the CDD and is not fully qualified with CATALOG.SCHEMA.TABLE, an 'ambiguous table reference' error will occur. The default catalog name can be supplied with '-a <catalog_name>'. The catalog name must exist in the CDD.	
-b suppress trailing blanks	Suppress character string trailing blanks. Character string Host-Variables used for input or output are padded with trailing blanks as required. Trailing blanks can be suppressed with '-b'. This is especially useful for applications written in C/C++. where it is normal practice for character strings to be NULL terminated without trailing blanks padding.	no
-c compatibility mode	The Precompiler executes in extended ACE mode. ANSI mode can be enabled with '-c ANSI'. This option affects messages only. In ANSI mode, SQL syntax which does not adhere to ANSI standards will be flagged as a warning. Code generation is not affected. Values : ACE / ANSI	ace
-e error listing file	Error, Warning and Statistics messages are written to STDOUT.	<basename>.plc (Windows and Open

	These messages can be routed to an output file with '-e <error listing file>'. The <error listing file> name must consist of at least a basename and optionally an extension. If no extension is supplied, the default extension 'pcl' will be used. On z/OS platforms, a DDNAME must be used instead of a filename, using the format : -e //ddn:ddname The DDNAME assignment must be made to a physical file. It must not be assigned to // DD SYSOUT=	Systems) DD=SYSTERM (z/OS)
-f code output file	The <code output file> name must consist of at least a basename and optionally an extension. If no extension is supplied, the default extension 'cbl' will be used.	<basename>.c
-k keep the generated output file in case of compilation errors	The <code output file> is deleted after compilation errors. The <code output file> can be kept with '-k'.	no
-l generate line information	Line information in the form : /* line nn "<source file name>" */ is generated into the <code output file> with '-l'. This can be useful when you want to know the line number of the associated SQL Statement in the original C/C++ source file.	no
-m migration check of SQL statements only	The Precompiler executes in code generation mode. When migrating or checking existing C/C++ or COBOL programs, it can be useful to have a listing of the SQL statements only. These are output to STDOUT. No output file is generated. The migration check can be enabled with '-m'.	no
-n Adabas SQL Gateway server name	This option is REQUIRED Server <server_name> to which the session is to connect. <server_name> need not be enclosed in quotes.	no default
-o output code mode	Output code mode is either 32 or 64-Bit The C/C++ Precompiler generates 32-Bit code when invoked on 32-Bit	32 (when invoked on 32-Bit systems) 64 (when invoked on 64-Bit systems)

	systems (-o 32) and 64-Bit code when invoked on 64-Bit systems (-o 64). Generation of code suitable for a different environment can be enabled with '-o 32' or '-o 64'. Values : 32 / 64	
-q no SQL validation	The Precompiler performs SQL syntax validation at precompile time. SQL validation can be disabled with '-q'.	no
-r length of output line	The Precompiler generates code output files with a line length of 72. The output line length can be increased to the length defined in <output line length>. The output line length cannot be defined shorter than the default length for the appropriate Host Language. If this is attempted, the output line length will be reset to the appropriate Host Language default and a warning message issued. Values : >= 72	72
-s schema name	The SCHEMA name used by the Precompiler is derived from <user_name> as supplied with the '-u' option. The default schema name can be overridden with '-s <schema_name>'. The schema name must exist in the CDD.	user_name
-u user name [, password]	This option is REQUIRED User <user_name> and Password <password> to be connected to the Server defined with '-n'. The <password> parameter is optional, however, this must be supplied if the user being connected requires a password.	no default
-w working directory	Input and output files are searched for or stored in the current directory. This can be overridden with '-w <working directory>'. If no filename is given for a file to be generated, the basename of the precompiler source file will be taken, with the appropriate extension.	current directory
-z suppress warnings	Warning messages are generated by the Precompiler.	no

	Warning messages can be suppressed with '-z'. This option affects WARNING messages only. ERROR messages cannot be suppressed.	
-@ file containing command line options and parameters	Command line options and parameters saved in a file can be optionally input to the Precompiler with '-@ <command line options and parameters file>'. The options and parameters in the <command line options and parameters file> can be combined with other command line options when invoking the Precompiler. On z/OS platforms, a DDNAME must be used instead of a filename, using the format : -@ //ddn:ddname The DDNAME assignment must be made to a physical file. It must not be assigned to // DD *	no default

Filename Conventions

- The C precompiler source file name must consist of at least a basename and optionally an extension. If no extension is supplied the default extension 'pcc' is used. If the source file cannot be located, an error is returned.
- The C precompiler generated output file name must consist of at least a basename and optionally an extension. If no extension is supplied the default extension 'c' is used.
- The (optional) C precompiler listing file name must consist of at least a basename and optionally an extension. If no extension is supplied, the default extension 'pcl' is used.
- If no directory is given, then the file is searched or stored in the current directory. If a working directory is set (w option), then all generated files is stored there. This setting can be overridden by a directory indication within a certain filename. If no filename is given for a file to be generated, then the basename of the precompiler source file is taken, with the appropriate extension.

Examples:

```
ACEPCC -nDD=MYDSN -uMYUSER precapp.pcc
```

All generated files are stored in the current directory. The code output file (C source) is assigned the name precapp.c.

```
ACEPCC -nDD=MYDSN,GATEWAY=REMOTEHOST,PORT=7500 -uMYUSER precapp.pcc
```

All generated files are stored in the current directory. The code output file (C source) is assigned the name precapp.c.

A remote connection is made to the CONNX JDBC Server, listening on PORT 7500.

```
ACEPCC -nDD=MYDSN -uMYUSER -w/TEMP -f/CSRC/MYSRC.CPP precapp.pcc
```

This precompiler call stores all generated files in /TEMP except the generated C source. This is stored in /CSRC with the name mysrc.cpp. The name of the source file of the precompiler is precapp.pcc.

Libraries

To build an executable program (or a dynamic link library), only the shared library librciclnt_32.so must be linked to the objects which are the result of previous executions of a C compiler.

Example:

```
ld precapp.o -librciclnt_32.so
```

z/OS

Invocation and Precompiler Options (z/OS - C)

The general format of the call to the C precompiler is:

```
EXEC ACEPCC [,PARM='<precompiler options>']
```

The C source file is input from DD=PCIN. If the source file cannot be located, an error is returned.

The C precompiler generated file is output to DD=PCOUT. Additionally, 3 temporary work files should be assigned : DD=PCWRK1, DD=PCWRK2 and DD=PCWRK3.

Options

Every option begins with a minus sign (-). The names of the options are not case-sensitive.

The following C precompiler options are available:

Option and Option Name	Description	Default
-a catalog name	If a table name exists multiple times in the CDD and is not fully qualified with CATALOG.SCHEMA.TABLE, an 'ambiguous table reference' error will occur. The default catalog name can be supplied with '-a <catalog_name>'. The catalog name must exist in the CDD.	
-b suppress trailing blanks	Suppress character string trailing blanks. Character string Host-Variables used for input or output are padded with trailing blanks as required. Trailing blanks can be suppressed with '-b'. This is especially useful for applications written in C/C++. where it is normal practice for character strings to be NULL terminated without trailing blanks padding.	no
-c compatibility mode	The Precompiler executes in extended ACE mode. ANSI mode can be enabled with '-c ANSI'. This option affects messages only. In ANSI mode, SQL syntax which does not adhere to ANSI standards will be flagged as a warning. Code generation is not affected. Values : ACE / ANSI	ace

-e error listing file	Error, Warning and Statistics messages are written to STDOUT. These messages can be routed to an output file with '-e <error listing file>'. The <error listing file> name must consist of at least a basename and optionally an extension. If no extension is supplied, the default extension 'pcl' will be used. On z/OS platforms, a DDNAME must be used instead of a filename, using the format : -e //ddn:ddname The DDNAME assignment must be made to a physical file. It must not be assigned to // DD SYSOUT=	<basename>.plc (Windows and Open Systems) DD=SYSTERM (z/OS)
-f code output file	The <code output file> name must consist of at least a basename and optionally an extension. If no extension is supplied, the default extension 'cbl' will be used.	<basename>.c
-k keep the generated output file in case of compilation errors	The <code output file> is deleted after compilation errors. The <code output file> can be kept with '-k'.	no
-l generate line information	Line information in the form : /* line nn "<source file name>" */ is generated into the <code output file> with '-l'. This can be useful when you want to know the line number of the associated SQL Statement in the original C/C++ source file.	no
-m migration check of SQL statements only	The Precompiler executes in code generation mode. When migrating or checking existing C/C++ or COBOL programs, it can be useful to have a listing of the SQL statements only. These are output to STDOUT. No output file is generated. The migration check can be enabled with '-m'.	no
-n Adabas SQL Gateway server name	This option is REQUIRED Server <server_name> to which the session is to connect. <server_name> need not be enclosed in quotes.	no default
-o output code mode	Output code mode is either 32 or 64-Bit	32 (when invoked on 32-Bit systems)

	The C/C++ Precompiler generates 32-Bit code when invoked on 32-Bit systems (-o 32) and 64-Bit code when invoked on 64-Bit systems (-o 64). Generation of code suitable for a different environment can be enabled with '-o 32' or '-o 64'. Values : 32 / 64	64 (when invoked on 64-Bit systems)
-q no SQL validation	The Precompiler performs SQL syntax validation at precompile time. SQL validation can be disabled with '-q'.	no
-r length of output line	The Precompiler generates code output files with a line length of 72. The output line length can be increased to the length defined in <output line length>. The output line length cannot be defined shorter than the default length for the appropriate Host Language. If this is attempted, the output line length will be reset to the appropriate Host Language default and a warning message issued. Values : >= 72	72
-s schema name	The SCHEMA name used by the Precompiler is derived from <user_name> as supplied with the '-u' option. The default schema name can be overridden with '-s <schema_name>'. The schema name must exist in the CDD.	user_name
-u user name [, password]	This option is REQUIRED User <user_name> and Password <password> to be connected to the Server defined with '-n'. The <password> parameter is optional, however, this must be supplied if the user being connected requires a password.	no default
-w working directory	Input and output files are searched for or stored in the current directory. This can be overridden with '-w <working directory>'. If no filename is given for a file to be generated, the basename of the precompiler source file will be taken, with the appropriate extension.	current directory

-z suppress warnings	Warning messages are generated by the Precompiler. Warning messages can be suppressed with '-z'. This option affects WARNING messages only. ERROR messages cannot be suppressed.	no
-@ file containing command line options and parameters	Command line options and parameters saved in a file can be optionally input to the Precompiler with '-@ <command line options and parameters file>'. The options and parameters in the <command line options and parameters file> can be combined with other command line options when invoking the Precompiler. On z/OS platforms, a DDNAME must be used instead of a filename, using the format : -@ //ddn:ddname The DDNAME assignment must be made to a physical file. It must not be assigned to // DD *	no default

Note :
The minimum required options are server name (-n) and user name (-u). All other options are optional

JCL Examples:

```
//MY$ACE   JOB  CLASS=G,MSGCLASS=X
/*JOBPARM  LINES=9999
//*
//* Precompile a C program using ACEPCC
//*
//* Error Listing output to DD=PCERROR using the -e option
//*
//  SET ACELOD=CONNX.LOAD
//  SET USRLOD=WORK.LOAD
//  SET USROBJ=WORK.OBJ
//*
//ACEPCC   EXEC PGM=ACEPCC,
//         PARM='-e//DDN:PCERROR -nDD=MYDSN,GATEWAY=1.2.3.4 -
uMYUSER',
//         REGION=0M
//STEPLIB  DD   DISP=SHR,DSN=&ACELOD
//PCIN     DD   DISP=SHR,DSN=WORK.IN(PRECAPP)    <-- Input
//PCOUT    DD   DISP=SHR,DSN=WORK.OUT(PRECAPP)   <-- Output
//PCWRK1   DD   DSN=&&PCWRK1,
//         DCB=(RECFM=FB,LRECL=80,BLKSIZE=8000),
//          SPACE=(4096,(500,500),,,ROUND),
//          UNIT=VIO
//PCWRK2   DD   DSN=&&PCWRK2,
//          DCB=(RECFM=FB,LRECL=80,BLKSIZE=8000),
//          SPACE=(4096,(500,500),,,ROUND),
//          UNIT=VIO
//PCWRK3   DD   DSN=&&PCWRK3,
//          DCB=(RECFM=FB,LRECL=80,BLKSIZE=8000),
//          SPACE=(4096,(500,500),,,ROUND),
//          UNIT=VIO
//SYSIN    DD   DUMMY
//SYSTERM  DD   SYSOUT=*
//SYSUDUMP DD   SYSOUT=*
//PCERROR  DD   DISP=SHR,DSN=WORK.ERROR(PRECAPP) <-- Error Listing
//
```

Libraries

To build an executable program, the object file ACE3GL must be linked to the objects which are the result of preceding executions of a C compiler and prelinker.

JCL Example:

```
//MY$ACE   JOB  CLASS=G,MSGCLASS=X
/*JOBPARM  LINES=9999
 //*
 //* Linkedit a precompiled C program with ACE3GL
 //*
 //  SET USRLOD=WORK.LOAD
 //  SET USROBJ=WORK.OBJ
 //*
 //LKED     EXEC PGM=IEWL,PARM='AMODE=31,LIST,MAP'
 //SYSLIB   DD   DISP=SHR,DSN=&USRLOD
 //SYSLIN   DD   DDNAME=SYSIN
 //SYSLMOD  DD   DISP=SHR,DSN=&USRLOD(PRECAPP),
 //SYSPRINT DD   SYSOUT=*,
 //         DCB=(RECFM=FBA,LRECL=133,BLKSIZE=1330)
 //SYSTERM  DD   SYSOUT=*
 //SYSUT1   DD   DSN=&&SYSUT1,
 //         DCB=BLKSIZE=1024,
 //         SPACE=(1024,(200,50)),
 //         UNIT=VIO
 //SYSUT2   DD   DSN=&&SYSUT2,
 //         DCB=BLKSIZE=1024,
 //         SPACE=(1024,(200,50)),
 //         UNIT=VIO
 //USROBJ   DD   DISP=SHR,DSN=&USROBJ
 //SYSIN    DD   *
  INCLUDE   SYSLIB(PRECAPP)
  INCLUDE   USROBJ(ACE3GL)
  ENTRY     MAIN
  NAME      PRECAPP(R)
 //
```

SQL Statements

SQL statements are not part of the host language but are embedded in an application written in the host language. The compilation of such a program consists of two phases; the precompilation of the SQL statements contained in the application program followed by the compilation of the actual program itself.

The SQL statements must be invisible to the host language compiler during the compilation phase. In fact, the embedded SQL statements are commented out by the precompiler and are replaced by statements generated into the application program in a form that corresponds to the requirements of the host language.

The precompiler must be able to identify all embedded SQL statements. Therefore, all SQL statements are delimited by special SQL delimiters. It is not possible to have more than one SQL statement between one set of delimiters.

The SQL Starting Delimiter

The starting delimiter consists of a sequence of two words:

```
EXEC  SQL
```

These words must be separated by one or more whitespace characters. They may be separated by one or more lines or blanks, and may be in either upper case or lower case depending on what the host language permits.

In ANSI mode, the two keywords must be in upper case and must be separated by blanks (not lines).

The SQL Statement Body

Once the starting delimiter has been specified, the statement itself must be provided. It must be separated from the starting delimiter by at least one whitespace character and may be specified on the same line or on a following line to the starting delimiter. The statement may be specified in either upper case or lower case and may be split over several lines. Each keyword or token must be separated by at least one whitespace character and may not be split over two or more lines. Keywords may be written in upper case or lower case depending on the host language. In ANSI mode, keywords must be written in upper case only.

Host Variables in COBOL

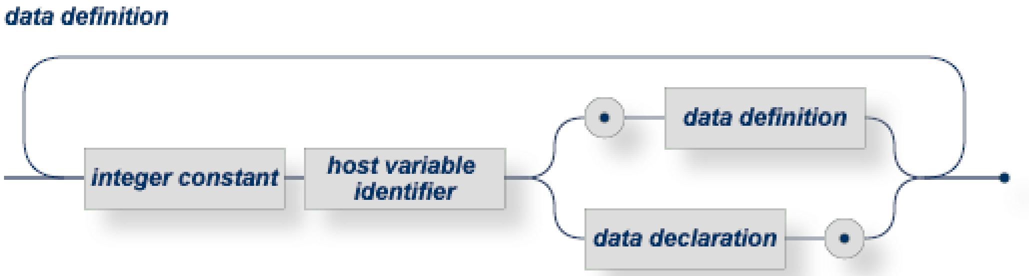
COBOL host variables used in SQL statements must be declared within the SQL BEGIN DECLARE SECTION and END DECLARE SECTION statements as well as in the COBOL DATA DIVISION. There may be any number of SQL BEGIN DECLARE SECTIONs. The host variable definition must be a valid COBOL data declaration, as described below. Adabas SQL Gateway Embedded SQL allows the use of single host variables and host variable structures.

Host Variable Declaration

COBOL host structures are a named set of COBOL single host variables and must conform to the ANSI Standard for COBOL.

Host Variable Structures

The use of COBOL host structures within SQL statements is an Adabas SQL Gateway Embedded SQL extension and not part of the SQL ANSI Standard.



integer constant	level number as described in the ANSI standard for COBOL
host variable identifier	specifies the identifier of the COBOL single variable or structure. Any valid COBOL identifier may be used.
data definition	recursive definition for nested structure level specification.
data declaration	see the syntax diagram below.

Example of a structure definition:

```
01      LEVEL1.
  02    LEVEL2.
    05  ELEMENT1 PIC 9.
    05  ELEMENT2 PIC 9.
```

Within embedded SQL statements the COBOL naming qualification rules for structure elements do not apply. Instead they must be specified top down to read "LEVEL1. LEVEL2.ELEMENT1" as shown in the example above.

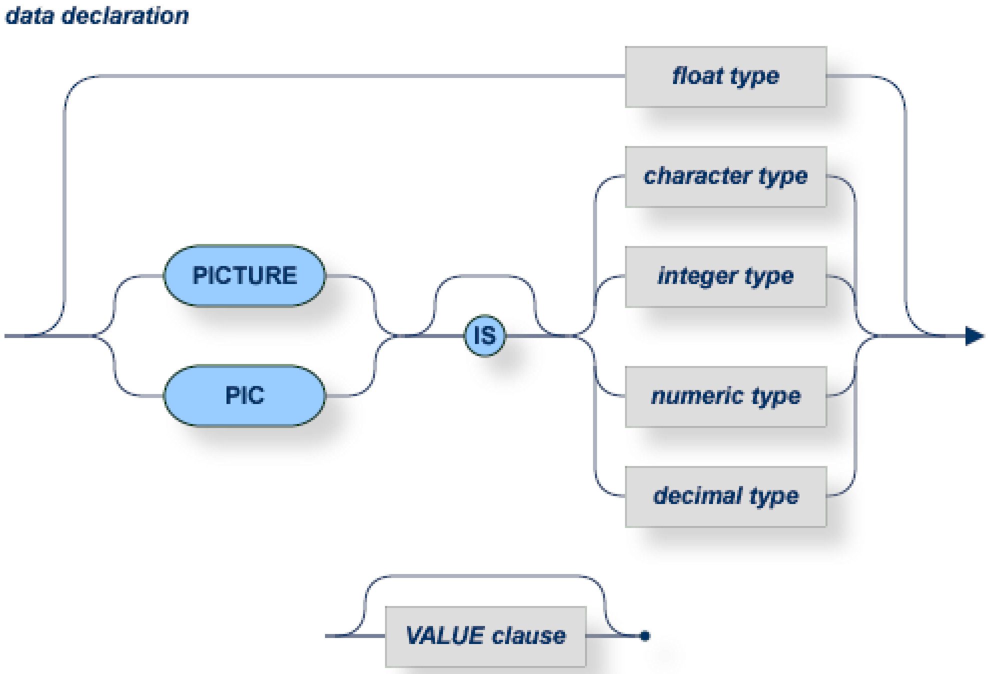
In COBOL statements, however, the structure elements must still be specified (bottom up) according to the ANSI COBOL rules: e.g. "ELEMENT1 IN LEVEL2 IN LEVEL1".

When referencing a structure element which is not uniquely identified within the compilation unit it must be sufficiently qualified with enough containing structure identifiers to unambiguously identify the variable concerned. If for example the identifier ELEMENT1 has been used in more than one structure definition then it must be qualified to give either LEVEL2.ELEMENT1 or even if necessary LEVEL1.LEVEL2.ELEMENT1.

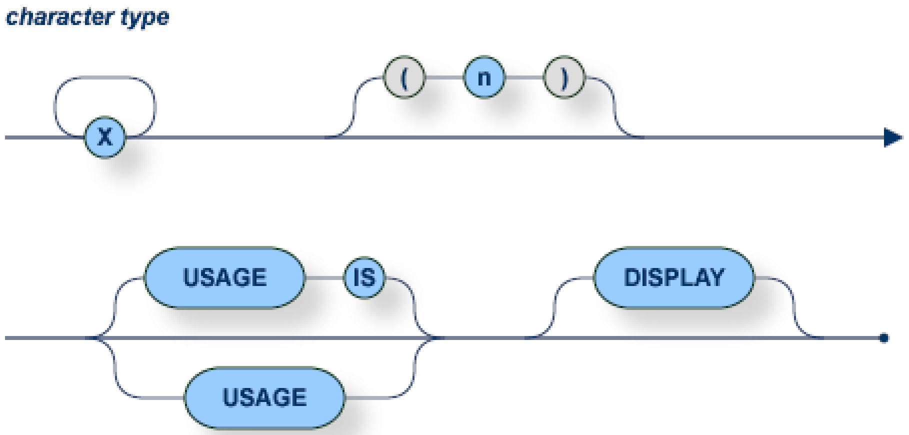
For more information about the general usage of host variables within SQL, see the topics under **Common Elements** in this help file.

Single Host Variables

The declaration must conform to the following COBOL syntax.

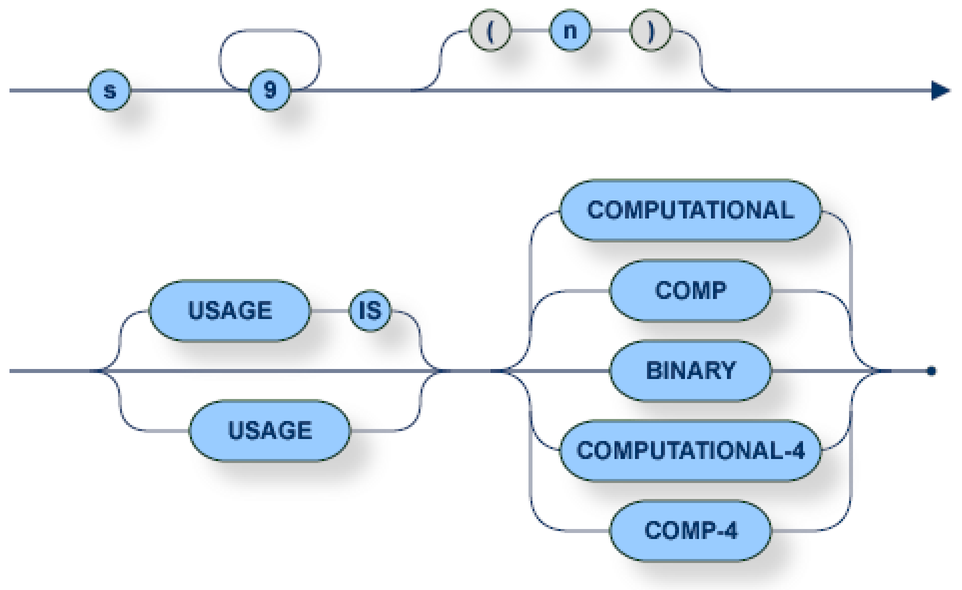


VALUE clause specifies any valid COBOL VALUE clause.



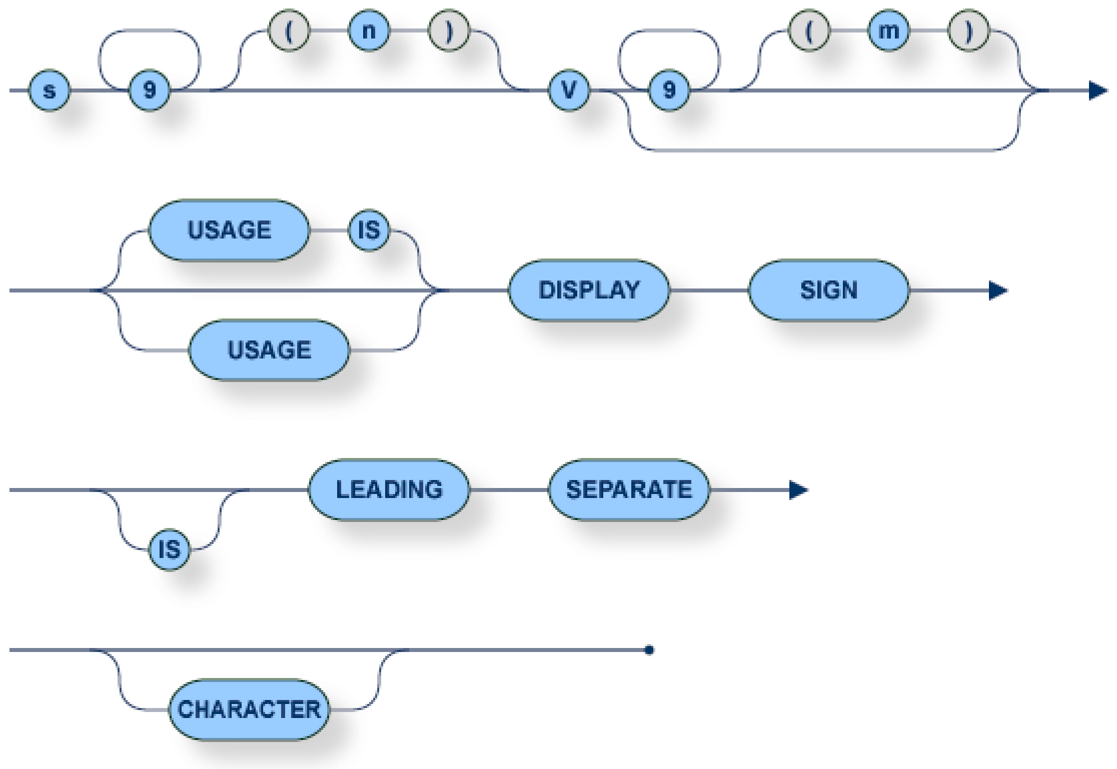
The number of significant characters must not exceed 253.

integer type



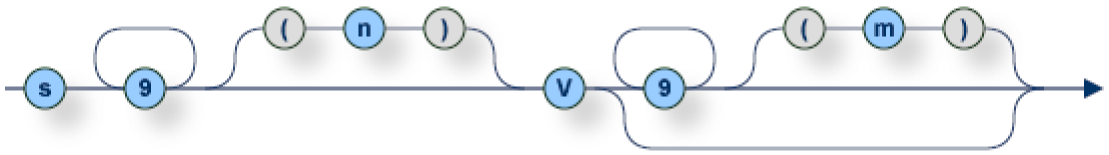
The number of digits must not exceed 9.

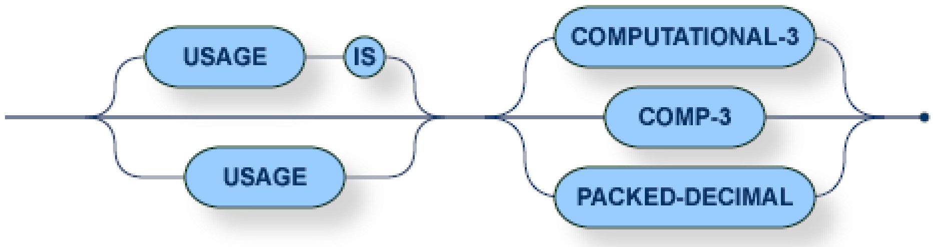
numeric type



The number of digits must not exceed 27.

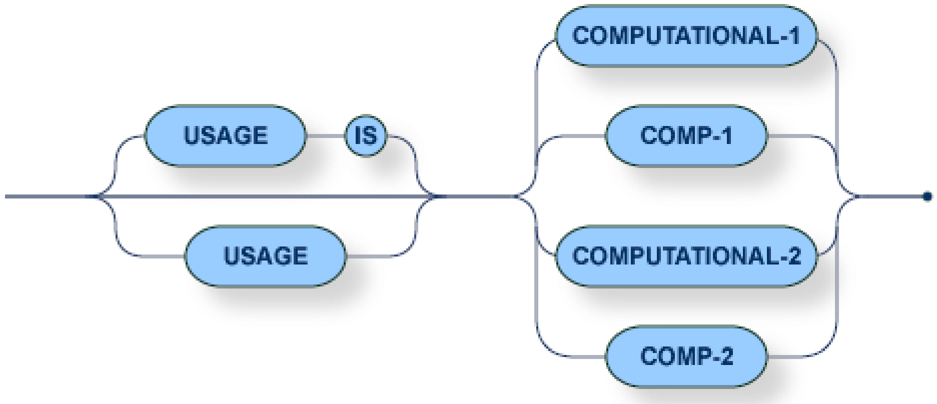
decimal type





The number of digits must not exceed 27.

float type



Data Type Conversion

The following table shows the conversion of COBOL data types to SQL data types and vice versa:

COBOL Data Types	SQL Data Types
character	CHARACTER
char (array)	BINARY
integer (5-9 digits)	INTEGER
integer (1-4 digits)	SMALLINT
numeric	NUMERIC
decimal	DECIMAL

float (comp-1)	REAL
float (comp-2)	DOUBLE-PRECISION

For more details on SQL data types and their usage in SQL statements refer to **Common Elements** in the Adabas SQL Gateway Embedded SQL Reference. The number of digits for an integer type must not exceed 9.

General Rules in COBOL

SQL Statement Delimiters

SQL statements are delimited by the prefix EXEC SQL and the terminator END-EXEC. The prefix may be written in upper or lower case letters. In Adabas SQL Gateway Embedded SQL mode, upper or lower case is permitted and the prefix may be split over numerous lines, separated by any whitespace character. In ANSI mode, upper case is required and only whitespaces may separate the prefix keywords.

The terminator END-EXEC may be followed by an optional period (.). It has to be coded on the same line to be recognized.

SQL Statement Placement

All SQL statements with the exception of the BEGIN/END DECLARE and INCLUDE statements may be specified wherever a COBOL statement may be specified within the Procedure Division of the embedded SQL COBOL program. Included COBOL source code must not contain any SQL statements or any host variable declaration to be used in SQL statements.

The SQL INCLUDE , BEGIN/END DECLARE statements must be specified in the WORKING STORAGE SECTION of the COBOL program.

Comments

SQL statements may contain COBOL comments marked with an asterisk in column 7 or SQL comments preceded by two minus characters.

COBOL Example :

```
000015    EXEC SQL WHENEVER SQLERROR
000016*                      CONTINUE
000017                       GOTO HANDLE-ERROR
000018    END-EXEC.
```

SQL Example :

```
000015    EXEC SQL WHENEVER SQLERROR
000016                    -- CONTINUE
000017                       GOTO HANDLE-ERROR
000018    END-EXEC.
```

Error Handling in COBOL

Textual error messages associated with each error number may be retrieved using the SQL statements GET DIAGNOSTICS and GET DIAGNOSTICS EXCEPTION. These can be declared in a subroutine which is called in the event of a non-zero SQLCODE being returned.

The COBOL program must declare a character variable to receive the error text, an integer variable containing the length of the error text, an integer variable containing the total number of error conditions available and an integer variable containing the current error condition. These 4 items must be declared in a DECLARE SECTION which is in scope whenever the above SQL statements are called.

A programming example using GET DIAGNOSTICS / GET DIAGNOSTICS EXCEPTION looks like this:

```
000015 WORKING-STORAGE SECTION.
000016    EXEC SQL BEGIN DECLARE SECTION END-EXEC.
000017* Error Text Buffer
000018 01  ERRBUF           PIC X(512).
000019* Length of Error Text Buffer
000020 01  ERRLEN           PIC 9(9)   COMP   SYNC VALUE 512.
000021* Count of error conditions
000022 01  CONDITIONCOUNT     PIC 9(9)   COMP   SYNC VALUE 0.
000023* Current error condition
000024 01  ERRNUMBER         PIC 9(9)   COMP   SYNC VALUE 0.
000025    EXEC SQL END   DECLARE SECTION END-EXEC.
000030    EXEC SQL INCLUDE SQLCA        END-EXEC.
000045 PROCEDURE DIVISION.
000041    IF SQLCODE NOT = 0
000042* Obtain the count of error conditions to be returned in
000043* CONDITIONCOUNT
000044      EXEC SQL
000045        GET DIAGNOSTICS :CONDITIONCOUNT = NUMBER
000046      END-EXEC
000050      IF CONDITIONCOUNT > 0
000051* Obtain each error condition text CONDITIONCOUNT times
000052* The error condition text will be returned in ERRBUF
000053        PERFORM DOERROR
000054          VARYING ERRNUMBER FROM 1 BY 1
000055            UNTIL ERRNUMBER > CONDITIONCOUNT
000056      END-IF
000057    END-IF
000058*
000070* Subroutine DOERROR
000071*
000072 DOEXCEPTION.
000073    EXEC SQL
000074      GET DIAGNOSTICS EXCEPTION :ERRNUMBER
000075                          :ERRBUF = MESSAGE_TEXT,
000076                          :ERRLEN = MESSAGE_LENGTH
000077    END-EXEC
000078    DISPLAY "ERR MSG: " G-ERRBUF
000079    .
```

where:

CONDITIONCOUNT is the count of error conditions available, returned by GET DIAGNOSTICS
ERRNUMBER is the current error condition.
ERRBUF is the target buffer, space filled on return from GET DIAGNOSTICS EXCEPTION

ERRLEN	is the length of the target buffer ERRBUF

SQL Communication Area (SQLCA)

The SQLCA provides the programmer with comprehensive information about the success or failure of each SQL command.

The following is the declaration of the SQLCA structure in COBOL:

```
01 SQLCA.
  02 SQLCAID     PIC X(8)       VALUE "sqlca   ".
  02 SQLCABC     PIC S9(9) COMP VALUE +136.
  02 SQLCODE     PIC S9(9) COMP VALUE +0.
  02 SQLERRM.
    49 SQLERRML PIC S9(4) COMP.
    49 SQLERRMC PIC X(70).
  02 SQLERRP     PIC X(8).
  02 SQLERRD     OCCURS 6 TIMES
            PIC S9(9) COMP.
  02 SQLWARN.
    03 SQLWARN0 PIC X.
    03 SQLWARN1 PIC X.
    03 SQLWARN2 PIC X.
    03 SQLWARN3 PIC X.
    03 SQLWARN4 PIC X.
    03 SQLWARN5 PIC X.
    03 SQLWARN6 PIC X.
    03 SQLWARN7 PIC X.
  02 SQLEXT      PIC X(8).
```

SQL Descriptor Area (SQLDA)

The SQLDA provides the programmer with comprehensive information about each resulting column of a dynamic SELECT statement.

The following is the declaration of the SQLDA structure in COBOL:

```
01 SQLDA.
   02 SQLDAID        PIC X(8)              VALUE "SQLDA".
   02 SQLDABC        PIC S9(9) COMP   SYNC VALUE 0.
   02 SQLN           PIC S9(4) COMP   SYNC VALUE 0.
   02 SQLD           PIC S9(4) COMP   SYNC VALUE 0.
   02 SQLVAR.
      03 SQLTYPE        PIC S9(9) COMP   SYNC VALUE 0.
      03 SQLLEN         PIC S9(9) COMP   SYNC VALUE 0.
      03 RESERVED       PIC S9(9) COMP   SYNC VALUE 0.
      03 INTERNAL       PIC S9(4) COMP   SYNC VALUE 0.
      03 SQLINDLEN      PIC S9(4) COMP   SYNC VALUE 0.
      03 SQLINDTYPE     PIC S9(9) COMP   SYNC VALUE 0.
      03 SQLIND.
         04 PTR            PIC S9(9) COMP   SYNC VALUE 0.
         04 HAD            PIC S9(9) COMP   SYNC VALUE 0.
      03 SQLDATA.
         04 PTR            PIC S9(9) COMP   SYNC VALUE 0.
         04 HAD            PIC S9(9) COMP   SYNC VALUE 0.
      03 SQLNAME.
         04 SQLNAMEL       PIC S9(9) COMP   SYNC VALUE 0.
         04 SQLNAMET       PIC S9(9) COMP   SYNC VALUE 0.
         04 SQLNAMER       PIC S9(9) COMP   SYNC VALUE 0.
```

To set the address of a host variable in SQLDATA and optionally in SQLIND, the following subroutine must be called:

```
CALL "SAGADDR" USING <host variable name>
                     sqldata of sqlvar of outsqlda(<index>)
```

Example:

```
CALL "SAGADDR" USING C2 sqldata of sqlvar of outsqlda(1)
```

Windows

Invocation and Precompiler Options - (Windows - COBOL)

The general format of the call to the COBOL precompiler is:

```
acepccob [<precompiler options>] <source file>
```

The COBOL source file can have any extension. If no extension is supplied the default extension 'pccob' is used. If the source file cannot be located, an error is returned.

Options

Every option begins with a minus sign (-). The names of the options are not case-sensitive.

The following COBOL precompiler options are available:

Option and Option Name	Description	Default
-a catalog name	If a table name exists multiple times in the CDD and is not fully qualified with CATALOG.SCHEMA.TABLE, an 'ambiguous table reference' error will occur. The default catalog name can be supplied with '-a <catalog_name>'. The catalog name must exist in the CDD.	
-b suppress trailing blanks	Suppress character string trailing blanks. Character string Host-Variables used for input or output are padded with trailing blanks as required. Trailing blanks can be suppressed with '-b'. This is especially useful for applications written in C/C++. where it is normal practice for character strings to be NULL terminated without trailing blanks padding.	no
-c compatibility mode	The Precompiler executes in extended ACE mode. ANSI mode can be enabled with '-c ANSI'. This option affects messages only. In ANSI mode, SQL syntax which does not adhere to ANSI standards will be flagged as a warning. Code generation is not affected. Values : ACE / ANSI	ace
-d string delimiter	The COBOL Precompiler by default	quote

(quotes)	uses QUOTE (") as string delimiters. Apostrophe (') string delimiters can be enabled with '-d APOST' Values :QUOTE / APOST	
-e error listing file	Error, Warning and Statistics messages are written to STDOUT. These messages can be routed to an output file with '-e <error listing file>'. The <error listing file> name must consist of at least a basename and optionally an extension. If no extension is supplied, the default extension 'pcl' will be used. On z/OS platforms, a DDNAME must be used instead of a filename, using the format : -e //ddn:ddname The DDNAME assignment must be made to a physical file. It must not be assigned to // DD SYSOUT=	<basename>.plc (Windows and Open Systems) DD=SYSTERM (z/OS)
-f code output file	The <code output file> name must consist of at least a basename and optionally an extension. If no extension is supplied, the default extension 'cbl' will be used.	<base of source file>.cbl
-h host language type	The COBOL Precompiler generates either independent (-h IND) or Microfocus (-h MF) code. When invoking the COBOL Precompiler with the '-h MF' option, an additional statement is generated for the first EXEC SQL CONNECT statement encountered in each program source and is a requirement of the Microfocus COBOL runtime on Windows and Open Systems platforms. SET SAGDLLPTR TO ENTRY "rciclnt" If you do not intend using Microfocus Cobol, the '-h IND' option should be defined (This is the default option on non Windows and Open Systems platforms). Values : IND / MF Default : MF (Windows/Open Systems) or IND (Mainframe)	Default : MF (Windows/Open Systems) IND (Mainframe)
-l tab character width	The COBOL Precompiler expands all tab characters by one or more spaces as defined in <tab character width>.	1

	Values : >= 1	
-k keep the generated output file in case of compilation errors	The <code output file> is deleted after compilation errors. The <code output file> can be kept with '-k'.	no
-m migration check of SQL statements only	The Precompiler executes in code generation mode. When migrating or checking existing C/C++ or COBOL programs, it can be useful to have a listing of the SQL statements only. These are output to STDOUT. No output file is generated. The migration check can be enabled with '-m'.	no
-n Adabas SQL Gateway server name	This option is REQUIRED Server <server_name> to which the session is to connect. <server_name> need not be enclosed in quotes.	no default
-q no SQL validation	The Precompiler performs SQL syntax validation at precompile time. SQL validation can be disabled with '-q'.	no
-r length of output line	The Precompiler generates code output files with a line length of 80. The output line length can be increased to the length defined in <output line length>. The output line length cannot be defined shorter than the default length for the appropriate Host Language. If this is attempted, the output line length will be reset to the appropriate Host Language default and a warning message issued. Values : >= 80	80
-s schema name	The SCHEMA name used by the Precompiler is derived from <user_name> as supplied with the '-u' option. The default schema name can be overridden with '-s <schema_name>'. The schema name must exist in the CDD.	user_name
-t cobol standard	The COBOL Precompiler processes all COBOL words and Host Variable definitions as case insensitive (-t 74). Case sensitivity can be enabled with '-t	74

	85'. Values : 74 / 85	
-u user name [, password]	This option is REQUIRED User <user_name> and Password <password> to be connected to the Server defined with '-n'. The <password> parameter is optional, however, this must be supplied if the user being connected requires a password.	no default
-w working directory	Input and output files are searched for or stored in the current directory. This can be overridden with '-w <working directory>'. If no filename is given for a file to be generated, the basename of the precompiler source file will be taken, with the appropriate extension.	current directory
-z suppress warnings	Warning messages are generated by the Precompiler. Warning messages can be suppressed with '-z'. This option affects WARNING messages only. ERROR messages cannot be suppressed.	no
-@ file containing command line options and parameters	Command line options and parameters saved in a file can be optionally input to the Precompiler with '-@ <command line options and parameters file>'. The options and parameters in the <command line options and parameters file> can be combined with other command line options when invoking the Precompiler. On z/OS platforms, a DDNAME must be used instead of a filename, using the format : -@ //ddn:ddname The DDNAME assignment must be made to a physical file. It must not be assigned to // DD *	no default

Note :
The minimum required options are server name (-n) and user name (-u). All other options are optional.

Filename Conventions

- The COBOL precompiler source file name must consist of at least a basename and optionally an extension. If no extension is supplied the default extension 'pccob' is used. If the source file cannot be located, an error is returned.

- The COBOL precompiler generated output file name must consist of at least a basename and optionally an extension. If no extension is supplied the default extension 'cbl' is used.
- The (optional) COBOL precompiler listing file name must consist of at least a basename and optionally an extension. If no extension is supplied the default extension 'pcl' will be used.
- If no directory is given, then the file will be searched or stored in the current directory. If a working directory is set (W option), then all generated files will be stored there. This setting can be overridden by a directory indication within a certain filename. If no filename is given for a file to be generated, then the basename of the precompiler source file will be taken, with the appropriate extension.

Examples:

```
acepccob -nDD=MYDSN -uMYUSER prebapp.pccob
```

All generated files will be stored in the current directory. The code output file (COBOL source) is assigned the name prebapp.cbl.

```
acepccob -nDD=MYDSN,GATEWAY=REMOTEHOST,PORT=7500 -uMYUSER prebapp.pccob
```

All generated files will be stored in the current directory. The code output file (COBOL source) is assigned the name prebapp.cbl.

A remote connection will be made to the CONNX JDBC Server, listening on PORT 7500.

```
acepccob -nDD=MYDSN -uMYUSER -wC:\TEMP -fD:\COBSRC\MYSRC.CBL
   prebapp.cob
```

This precompiler call stores all generated files in c:\temp except the generated COBOL source. This will be stored in d:\cobsrc with the name mysrc.cbl. The name of the source file of the precompiler is prebapp.cob.

Libraries

To build an executable program (or a dynamic link library), only the rciclnt.lib must be linked to the objects which are the result of preceding executions of a COBOL compiler.

Example:

```
cbllink -v -oprebapp.exe prebapp.cbl c:\connx32\precompiler\rciclnt.lib
```

Unix

Invocation and Precompiler Options (Unix - COBOL)

The general format of the call to the COBOL precompiler is:

```
ACEPCCOB [<precompiler options>] <source file>
```

The COBOL source file can have any extension. If no extension is supplied the default extension 'pccob' will be used. If the source file cannot be located, an error is returned.

Options

Every option begins with a minus sign (-). The names of the options are not case-sensitive.

The following COBOL precompiler options are available:

Option and Option Name	Description	Default
-a catalog name	If a table name exists multiple times in the CDD and is not fully qualified with CATALOG.SCHEMA.TABLE, an 'ambiguous table reference' error will occur. The default catalog name can be supplied with '-a <catalog_name>'. The catalog name must exist in the CDD.	
-b suppress trailing blanks	Suppress character string trailing blanks. Character string Host-Variables used for input or output are padded with trailing blanks as required. Trailing blanks can be suppressed with '-b'. This is especially useful for applications written in C/C++. where it is normal practice for character strings to be NULL terminated without trailing blanks padding.	no
-c compatibility mode	The Precompiler executes in extended ACE mode. ANSI mode can be enabled with '-c ANSI'. This option affects messages only. In ANSI mode, SQL syntax which does not adhere to ANSI standards will be flagged as a warning. Code generation is not affected. Values : ACE / ANSI	ace
-d string delimiter (quotes)	The COBOL Precompiler by default uses QUOTE (") as string delimiters. Apostrophe (') string delimiters can be enabled with '-d APOST'	quote

	Values :QUOTE / APOST	
-e error listing file	Error, Warning and Statistics messages are written to STDOUT. These messages can be routed to an output file with '-e <error listing file>'. The <error listing file> name must consist of at least a basename and optionally an extension. If no extension is supplied, the default extension 'pcl' will be used. On z/OS platforms, a DDNAME must be used instead of a filename, using the format : -e //ddn:ddname The DDNAME assignment must be made to a physical file. It must not be assigned to // DD SYSOUT=	<basename>.plc (Windows and Open Systems) DD=SYSTERM (z/OS)
-f code output file	The <code output file> name must consist of at least a basename and optionally an extension. If no extension is supplied, the default extension 'cbl' will be used.	<base of source file>.cbl
-h host language type	The COBOL Precompiler generates either independent (-h IND) or Microfocus (-h MF) code. When invoking the COBOL Precompiler with the '-h MF' option, an additional statement is generated for the first EXEC SQL CONNECT statement encountered in each program source and is a requirement of the Microfocus COBOL runtime on Windows and Open Systems platforms. SET SAGDLLPTR TO ENTRY "rciclnt" If you do not intend using Microfocus Cobol, the '-h IND' option should be defined (This is the default option on non Windows and Open Systems platforms). Values : IND / MF Default : MF (Windows/Open Systems) or IND (Mainframe)	Default : MF (Windows/Open Systems) IND (Mainframe)
-l tab character width	The COBOL Precompiler expands all tab characters by one or more spaces as defined in <tab character width>. Values : >= 1	1
-k keep the generated output file in case of compilation errors	The <code output file> is deleted after compilation errors. The <code output file> can be kept with '-k'.	no
-m migration check of SQL	The Precompiler executes in code	no

statements only	generation mode. When migrating or checking existing C/C++ or COBOL programs, it can be useful to have a listing of the SQL statements only. These are output to STDOUT. No output file is generated. The migration check can be enabled with '-m'.	
-n Adabas SQL Gateway server name	This option is REQUIRED Server <server_name> to which the session is to connect. <server_name> need not be enclosed in quotes.	no default
-q no SQL validation	The Precompiler performs SQL syntax validation at precompile time. SQL validation can be disabled with '-q'.	no
-r length of output line	The Precompiler generates code output files with a line length of 80. The output line length can be increased to the length defined in <output line length>. The output line length cannot be defined shorter than the default length for the appropriate Host Language. If this is attempted, the output line length will be reset to the appropriate Host Language default and a warning message issued. Values : >= 80	80
-s schema name	The SCHEMA name used by the Precompiler is derived from <user_name> as supplied with the '-u' option. The default schema name can be overridden with '-s <schema_name>'. The schema name must exist in the CDD.	user_name
-t cobol standard	The COBOL Precompiler processes all COBOL words and Host Variable definitions as case insensitive (-t 74). Case sensitivity can be enabled with '-t 85'. Values : 74 / 85	74
-u user name [, password]	This option is REQUIRED User <user_name> and Password <password> to be connected to the Server defined with '-n'. The <password> parameter is optional, however, this must be supplied if the	no default

	user being connected requires a password.	
-w working directory	Input and output files are searched for or stored in the current directory. This can be overridden with '-w <working directory>'. If no filename is given for a file to be generated, the basename of the precompiler source file will be taken, with the appropriate extension.	current directory
-z suppress warnings	Warning messages are generated by the Precompiler. Warning messages can be suppressed with '-z'. This option affects WARNING messages only. ERROR messages cannot be suppressed.	no
-@ file containing command line options and parameters	Command line options and parameters saved in a file can be optionally input to the Precompiler with '-@ <command line options and parameters file>'. The options and parameters in the <command line options and parameters file> can be combined with other command line options when invoking the Precompiler. On z/OS platforms, a DDNAME must be used instead of a filename, using the format : -@ //ddn:ddname The DDNAME assignment must be made to a physical file. It must not be assigned to // DD *	no default

Note :
The minimum required options are server name (-n) and user name (-u). All other options are optional.

Filename Conventions

- The COBOL precompiler source file name must consist of at least a basename and optionally an extension. If no extension is supplied the default extension 'pccob' will be used. If the source file cannot be located, an error will be returned..
- The COBOL precompiler generated output file name must consist of at least a basename and optionally an extension. If no extension is supplied the default extension 'cbl' will be used.
- The (optional) COBOL precompiler listing file name must consist of at least a basename and optionally an extension. If no extension is supplied the default extension 'pcl' will be used.
- If no directory is given, then the file will be searched or stored in the current directory. If a working directory is set (w option), then all generated files will be stored there. This setting can be overridden by a directory indication within a certain filename. If no filename is given for a file to be generated, then the basename of the precompiler source file will be taken, with the appropriate extension.

Examples:

```
ACEPCCOB -nDD=MYDSN -uMYUSER prebapp.pccob
```

All generated files will be stored in the current directory. The code output file (COBOL source) is assigned the name prebapp.cbl.

```
ACEPCCOB -nDD=MYDSN,GATEWAY=REMOTEHOST,PORT=7500 -uMYUSER prebapp.pccob
```

All generated files will be stored in the current directory. The code output file (COBOL source) is assigned the name prebapp.cbl.

A remote connection will be made to the CONNX JDBC Server, listening on PORT 7500.

```
ACEPCCOB -nDD=MYDSN -uMYUSER -w/TEMP -f/COBSRC/MYSRC.CBL prebapp.cob
```

This precompiler call stores all generated files in /TEMP except the generated COBOL source. This will be stored in /COBSRC with the name mysrc.cbl. The name of the source file of the precompiler is prebapp.cob.

Libraries

To build an executable program (or a dynamic link library), only the shared library rciclnt.so must be linked to the objects which are the result of preceding executions of a COBOL compiler.

Example:

```
ld prebapp.o -lrciclnt
```

z/OS

Invocation and Precompiler Options (z/OS- COBOL)

The general format of the call to the COBOL precompiler is:

```
EXEC ACEPCCOB [,PARM='<precompiler options>']
```

The COBOL source file is DD=PCIN. If the source file cannot be located, an error is returned.

The COBOL precompiler generated file is output to DD=PCOUT. Additionally, 1 temporary work file should be assigned : DD=PCWRK1.

Options

Every option begins with a minus sign (-). The names of the options are not case-sensitive.

The following COBOL precompiler options are available:

Option and Option Name	Description	Default
-a catalog name	If a table name exists multiple times in the CDD and is not fully qualified with CATALOG.SCHEMA.TABLE, an 'ambiguous table reference' error will occur. The default catalog name can be supplied with '-a <catalog_name>'. The catalog name must exist in the CDD.	
-b suppress trailing blanks	Suppress character string trailing blanks. Character string Host-Variables used for input or output are padded with trailing blanks as required. Trailing blanks can be suppressed with '-b'. This is especially useful for applications written in C/C++. where it is normal practice for character strings to be NULL terminated without trailing blanks padding.	no
-c compatibility mode	The Precompiler executes in extended ACE mode. ANSI mode can be enabled with '-c ANSI'. This option affects messages only. In ANSI mode, SQL syntax which does not adhere to ANSI standards will be flagged as a warning. Code generation is not affected. Values : ACE / ANSI	ace

-d string delimiter (quotes)	The COBOL Precompiler by default uses QUOTE (") as string delimiters. Apostrophe (') string delimiters can be enabled with '-d APOST' Values :QUOTE / APOST	quote
-e error listing file	Error, Warning and Statistics messages are written to STDOUT. These messages can be routed to an output file with '-e <error listing file>'. The <error listing file> name must consist of at least a basename and optionally an extension. If no extension is supplied, the default extension 'pcl' will be used. On z/OS platforms, a DDNAME must be used instead of a filename, using the format : -e //ddn:ddname The DDNAME assignment must be made to a physical file. It must not be assigned to // DD SYSOUT=	<basename>.plc (Windows and Open Systems) DD=SYSTERM (z/OS)
-f code output file	The <code output file> name must consist of at least a basename and optionally an extension. If no extension is supplied, the default extension 'cbl' will be used.	<base of source file>.cbl
-h host language type	The COBOL Precompiler generates either independent (-h IND) or Microfocus (-h MF) code. When invoking the COBOL Precompiler with the '-h MF' option, an additional statement is generated for the first EXEC SQL CONNECT statement encountered in each program source and is a requirement of the Microfocus COBOL runtime on Windows and Open Systems platforms. SET SAGDLLPTR TO ENTRY "rciclnt" If you do not intend using Microfocus Cobol, the '-h IND' option should be defined (This is the default option on non Windows and Open Systems platforms). Values : IND / MF Default : MF (Windows/Open Systems) or IND (Mainframe)	Default : MF (Windows/Open Systems) IND (Mainframe)
-l tab character width	The COBOL Precompiler expands all tab characters by one or more spaces	1

	as defined in <tab character width>. Values : >= 1	
-k keep the generated output file in case of compilation errors	The <code output file> is deleted after compilation errors. The <code output file> can be kept with '-k'.	no
-m migration check of SQL statements only	The Precompiler executes in code generation mode. When migrating or checking existing C/C++ or COBOL programs, it can be useful to have a listing of the SQL statements only. These are output to STDOUT. No output file is generated. The migration check can be enabled with '-m'.	no
-n Adabas SQL Gateway server name	This option is REQUIRED Server <server_name> to which the session is to connect. <server_name> need not be enclosed in quotes.	no default
-q no SQL validation	The Precompiler performs SQL syntax validation at precompile time. SQL validation can be disabled with '-q'.	no
-r length of output line	The Precompiler generates code output files with a line length of 80. The output line length can be increased to the length defined in <output line length>. The output line length cannot be defined shorter than the default length for the appropriate Host Language. If this is attempted, the output line length will be reset to the appropriate Host Language default and a warning message issued. Values : >= 80	80
-s schema name	The SCHEMA name used by the Precompiler is derived from <user_name> as supplied with the '-u' option. The default schema name can be overridden with '-s <schema_name>'. The schema name must exist in the CDD.	user_name
-t cobol standard	The COBOL Precompiler processes all COBOL words and Host Variable definitions as case insensitive (-t 74).	74

	Case sensitivity can be enabled with '-t 85'. Values : 74 / 85	
-u user name [, password]	This option is REQUIRED User <user_name> and Password <password> to be connected to the Server defined with '-n'. The <password> parameter is optional, however, this must be supplied if the user being connected requires a password.	no default
-w working directory	Input and output files are searched for or stored in the current directory. This can be overridden with '-w <working directory>'. If no filename is given for a file to be generated, the basename of the precompiler source file will be taken, with the appropriate extension.	current directory
-z suppress warnings	Warning messages are generated by the Precompiler. Warning messages can be suppressed with '-z'. This option affects WARNING messages only. ERROR messages cannot be suppressed.	no
-@ file containing command line options and parameters	Command line options and parameters saved in a file can be optionally input to the Precompiler with '-@ <command line options and parameters file>'. The options and parameters in the <command line options and parameters file> can be combined with other command line options when invoking the Precompiler. On z/OS platforms, a DDNAME must be used instead of a filename, using the format : -@ //ddn:ddname The DDNAME assignment must be made to a physical file. It must not be assigned to // DD *	no default

Note :
The minimum required options are server name (-n) and user name (-u). All other options are optional.

JCL Example:

```
//MY$ACE JOB CLASS=G,MSGCLASS=X
```

```
/*JOBPARM LINES=9999
//*
//* Precompile a COBOL program using ACEPCCOB
//*
//* Error Listing output to DD=PCERROR using the -e option
//*
// SET ACELOD=CONNX.LOAD
// SET USRLOD=WORK.LOAD
// SET USROBJ=WORK.OBJ
//*
//ACEPCCOB EXEC PGM=ACEPCCOB
// PARM='-e//DDN:PCERROR -nDD=MYDSN,GATEWAY=1.2.3.4 -uMYUSER',
// REGION=0M
//STEPLIB DD DISP=SHR,DSN=&ACELOD
//PCINCL DD DISP=SHR,DSN=WORK.INCL <-- Include Directory
//PCIN DD DISP=SHR,DSN=WORK.IN(PREBAPP) <-- Input
//PCOUT DD DISP=SHR,DSN=WORK.OUT(PREBAPP) <-- Output
//PCWRK1 DD DSN=&&PCWRK1,
// DCB=(RECFM=FB,LRECL=80,BLKSIZE=8000),
// SPACE=(4096,(500,500),,,ROUND),
// UNIT=VIO
//PCWRK2 DD DSN=&&PCWRK2,
// DCB=(RECFM=FB,LRECL=80,BLKSIZE=8000),
// SPACE=(4096,(500,500),,,ROUND),
// UNIT=VIO
//PCWRK3 DD DSN=&&PCWRK3,
// DCB=(RECFM=FB,LRECL=80,BLKSIZE=8000),
// SPACE=(4096,(500,500),,,ROUND),
// UNIT=VIO
//SYSIN DD DUMMY
//SYSTERM DD SYSOUT=*
//SYSUDUMP DD SYSOUT=*
//PCERROR DD DISP=SHR,DSN=WORK.ERROR(PREBAPP) <-- Error Listing
//
```

Libraries

To build an executable program, the object file ACE3GL must be linked to the objects which are the result of preceding executions of a COBOL compiler.

JCL Example:

```
//MY$ACE   JOB  CLASS=G,MSGCLASS=X
/*JOBPARM  LINES=9999
//*
//* Linkedit a precompiled COBOL program with ACE3GL
//*
//  SET USRLOD=WORK.LOAD
//  SET USROBJ=WORK.OBJ
//*
```

```
//LKED     EXEC PGM=IEWL,PARM='AMODE=31,LIST,MAP'
//SYSLIB   DD   DISP=SHR,DSN=&USRLOD
//SYSLIN   DD   DDNAME=SYSIN
//SYSLMOD  DD   DISP=SHR,DSN=&USRLOD(PREBAPP),
//SYSPRINT DD   SYSOUT=*,
//         DCB=(RECFM=FBA,LRECL=133,BLKSIZE=1330)
//SYSTERM  DD   SYSOUT=*
//SYSUT1   DD   DSN=&&SYSUT1,
//         DCB=BLKSIZE=1024,
//         SPACE=(1024,(200,50)),
//         UNIT=VIO
//SYSUT2   DD   DSN=&&SYSUT2,
//         DCB=BLKSIZE=1024,
//         SPACE=(1024,(200,50)),
//         UNIT=VIO
//USROBJ   DD   DISP=SHR,DSN=&USROBJ
//SYSIN    DD   *
 INCLUDE   SYSLIB(PREBAPP)
 INCLUDE   USROBJ(ACE3GL)
 ENTRY     PREBAPP
 NAME      PREBAPP(R)
//
```

Chapter 7 - SQL Statements

Standard SQL Statements

The SQL Statements supported by the Adabas SQL Gateway Embedded SQL is a superset of statements supported

by the Adabas SQL Gateway. For documentation on standard SQL Statements, please refer to the Adabas SQL Gateway User Guide.

The SQL Statements listed in this section only apply to Embedded SQL.

ALLOCATE SQLCONTEXT

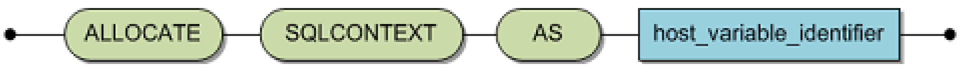
Function:

The ALLOCATE SQLCONTEXT statement identifies to the SQL precompiler which SQL Context is in scope for subsequent embedded SQL statements.

Invocation:

Embedded Mode ✓	Dynamic Mode	Interactive Mode

Syntax:



host_variable_identifier	A valid single host variable identifier. It must resolve to the address of an SQL Context (a SAGContext structure).

Description

The ALLOCATE SQLCONTEXT statement is provided to facilitate the writing of multi-threaded SQL client applications. This is a programmatic mechanism to allow the SQL application to maintain multiple SQL contexts, each corresponding to a logical SQL user, in a multi-threaded application. The ALLOCATE SQLCONTEXT statement identifies to the SQL precompiler which SQL Context is active for subsequent embedded SQL statements. At runtime, the host variable specified in the statement must resolve to the address of a valid SQL Context.

The address in the host variable can be changed, with the effect that the new address identifies that another SQL Context is now active. No actual memory is allocated by this statement. Instead, the address in the host variable must point to an already allocated SAGContext structure. This statement does not result in a call to the Adabas SQL Gateway Embedded SQL; it is a programmatic directive to the SQL precompiler only.

Limitations:

An instance of an ALLOCATE SQLCONTEXT statement is in scope from the point in the embedded-SQL source module at which the statement appears until one of the following:

- another ALLOCATE SQLCONTEXT statement is encountered
- any DEALLOCATE SQLCONTEXT statement is encountered
- the end of the source module.

ANSI Specifics:

The ALLOCATE SQLCONTEXT statement is not part of the Standard.

Adabas SQL Gateway Embedded SQL Specifics:

See also the related Adabas SQL Gateway Embedded SQL statement: DEALLOCATE SQLCONTEXT.

Example

An SQL context can be identified as follows:

```
  exec sql begin declare section;
        SAGContext  sqlCtx;
     exec sql end declare section;
     exec sql allocate sqlcontext as :&sqlCtx
```

BEGIN DECLARE SECTION

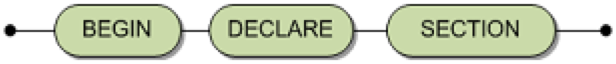
Function

This statement is the starting delimiter for a host variable declaration block.

Invocation

Embedded Mode ✓	Dynamic Mode	Interactive Mode

Syntax



Description

SQL application programs need to retrieve and provide values to and from Adabas SQL Gateway Embedded SQL during runtime. This is achieved by using host variables which are specified in embedded SQL statements. During compilation, the nature of the host variables has to be known. To identify the relevant host variables they must be declared in a special section. This section is delimited by the BEGIN DECLARE SECTION and END DECLARE SECTION statements. These statements are always paired and can not be nested. The host variable declarations must be specified between the two statements and more than one of these sections are permitted. The statement does not result in an update of the SQLCA.

Limitations

The positioning of the statement must conform to the rules governing the positioning of host variable declarations with the host applications. At least one host variable should be declared in such a block.

ANSI Specifics

Any host variable referenced within an embedded SQL statement must have been declared with a host variable declaration section. Structures are not permitted in this context.

Example

The following example shows the start of the host variable declaration section.

```
BEGIN DECLARE SECTION
   char a
END DECLARE SECTION;
```

CLOSE

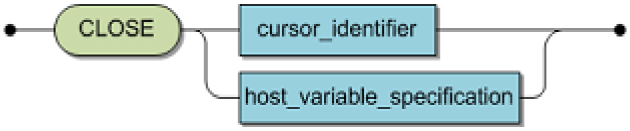
Function

The CLOSE statement closes a cursor.

Invocation

Embedded Mode ✓	Dynamic Mode	Interactive Mode

Syntax



cursor_identifier	A valid cursor identifier which identifies the cursor to be closed.
host_variable_specification	A valid single host variable specification. It must have been defined in the application program according to the host language rules.

Description

The CLOSE statement closes a cursor. It releases resources allocated by an OPEN cursor statement. The value of the host variable must be a valid cursor identifier. A host variable can be used as cursor identifier only if the cursor is a dynamically declared cursor.

Limitations

The cursor to be closed must have been opened.

ANSI Specifics

All cursors opened within a transaction are automatically closed by a COMMIT or ROLLBACK statement. The associated DECLARE CURSOR statement must precede the CLOSE statement in the host program.

Adabas SQL Gateway Embedded SQL Specifics

The CLOSE statement does not have to be preceded by the associated DECLARE CURSOR statement. It may appear anywhere in the host program, even in another compilation unit.

Example

The following example closes cursor1.

```
CLOSE cursor1;
```

COMMIT

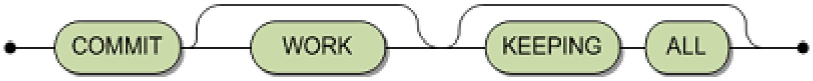
Function

The COMMIT statement terminates a transaction and makes permanent all changes that were made to the database during the terminated transaction.

Invocation

Embedded Mode ✓	Dynamic Mode ✓	Interactive Mode ✓

Syntax



Description

The COMMIT statement terminates the current transaction and starts a new transaction. All changes to the database that have been made during the terminated transaction are made permanent. All cursors that have been opened during the current transaction are closed.

The KEEPING ALL is currently ignored by the Adabas SQL Gateway.

ANSI Specifics

The keyword WORK is mandatory. The keywords KEEPING ALL are not part of the Standard.

Adabas SQL Gateway Embedded SQL Specifics

The keyword KEEPING ALL is an Adabas SQL Gateway Embedded SQL extension.

Example

The following example commits all changes made to the database in the current transaction.

```
COMMIT WORK;
```

CONNECT

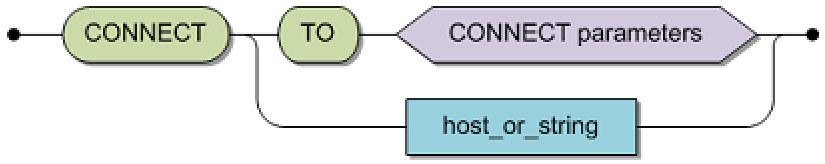
Function

The CONNECT statement explicitly establishes an SQL session between a user application and Adabas SQL Gateway Embedded SQL.

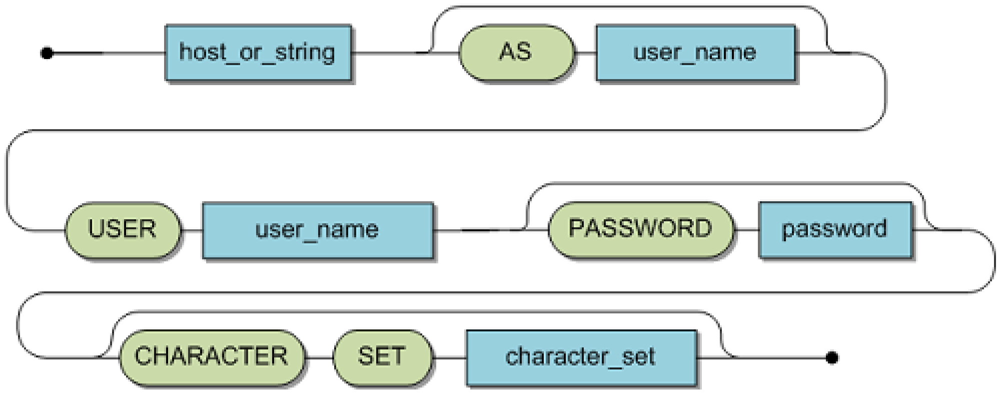
Invocation

Embedded Mode ✓	Dynamic Mode	Interactive Mode ✓

Syntax



CONNECT parameters :=



All specifiers can either be character string constants or single host variable identifiers. The host variables must have been defined in the application program according to the host language rules and their values must be character strings.

The maximum lengths of these specifiers are as follows:

- Server (255 characters)
- Connection (32 characters)
- User (32 characters)
- Password (32 characters)

Description

The CONNECT statement explicitly establishes an SQL session between the user application and Adabas SQL Gateway Embedded SQL.

The user specified must exist (see CREATE USER statement). If user is not specified, the connection will be refused. If a password is not specified, blanks will be generated as default password.

If a server is not specified, the default server is used.

Limitations

None.

ANSI Specifics

None

Adabas SQL Gateway Embedded SQL Specifics

none

Examples:

```
CONNECT TO MYASG user XXX password YYY
```

DEALLOCATE PREPARE

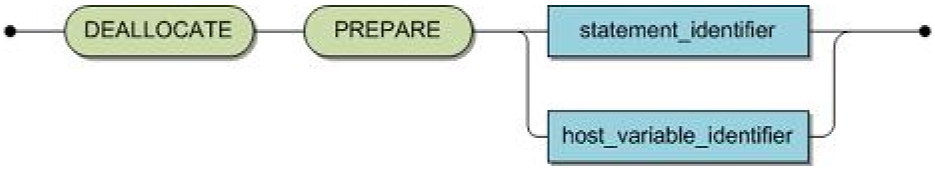
Function

The DEALLOCATE PREPARE statement deallocates a prepared statement by releasing all associated resources. After the successful execution of a DEALLOCATE PREPARE statement the relevant prepared statement no longer exists and can, therefore, not be addressed anymore.

Invocation

Embedded Mode ✓	Dynamic Mode	Interactive Mode

Syntax



statement_identifier	A valid identifier used to identify the statement to be deallocated.
host_variable_identifier	A valid single host variable identifier. It must contain the statement identifier.

Description

The effect of a DEALLOCATE PREPARE statement is that the identified statement will be deleted.

The effect of a DEALLOCATE PREPARE statement is also achieved implicitly when an already existing prepared statement is specified in a PREPARE statement. An implicit DEALLOCATE PREPARE statement also occurs in either of the following situations:

- a COMMIT or ROLLBACK is executed
- a DISCONNECT is issued to end a session.

Limitations

All cursors must be closed before executing a DEALLOCATE PREPARE statement.

ANSI Specifics

None.

Adabas SQL Gateway Embedded SQL Specifics

This statement may be mixed with any other DML, DDL and/or DCL statements in the same transaction.

Example

The following example deletes the prepared statement identified by statement_identifier.

```
DEALLOCATE PREPARE statement_identifier;
```

DEALLOCATE SQLCONTEXT

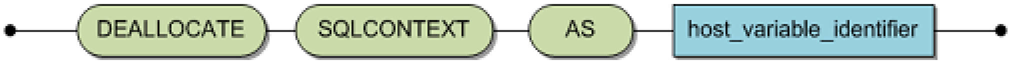
Function

The DEALLOCATE SQLCONTEXT statement instructs the run-time client support libraries to delete all memory resources relating to an SQL Context.

Invocation

Embedded Mode ✓	Dynamic Mode	Interactive Mode

Syntax



host_variable_identifier	A valid single host variable identifier must resolve to the address of an SQL Context (an SAGContext structure).

Description

The effect of a DEALLOCATE PREPARE statement is that the all resources associated with the specified SQL Context will be deleted.

This statement results in a call, at runtime, to the client support libraries provided by the Adabas SQL Gateway Embedded SQL. No call is made to the Adabas SQL Gateway Embedded SQL itself. The SAGContext structure itself is not deallocated - only the associated resources that have been created by the client support libraries in the course of activities related to that SQL Context.

An error will be returned if:

- the SQL Context contains open connections
- the host variable does not resolve to a valid address of a SQL Context
- the SQL Context is currently already in use.

Limitations

All SQL connections must be closed before executing a DEALLOCATE SQLCONTEXT statement.

ANSI Specifics

The DEALLOCATE SQLCONTEXT statement is not part of the Standard.

Adabas SQL Gateway Embedded SQL Specifics

See also the related Adabas SQL Gateway Embedded SQL statement: ALLOCATE SQLCONTEXT.

Example

A SQL context can be deallocated as follows:

```
exec sql begin declare section;
      SAGContext  sqlCtx;
   exec sql end declare section;
   exec sql allocate sqlcontext as :&sqlCtx
```

```
          :
    < other statements >
          :
exec sql deallocate sqlcontext as :&sqlCtx
```

DECLARE CURSOR

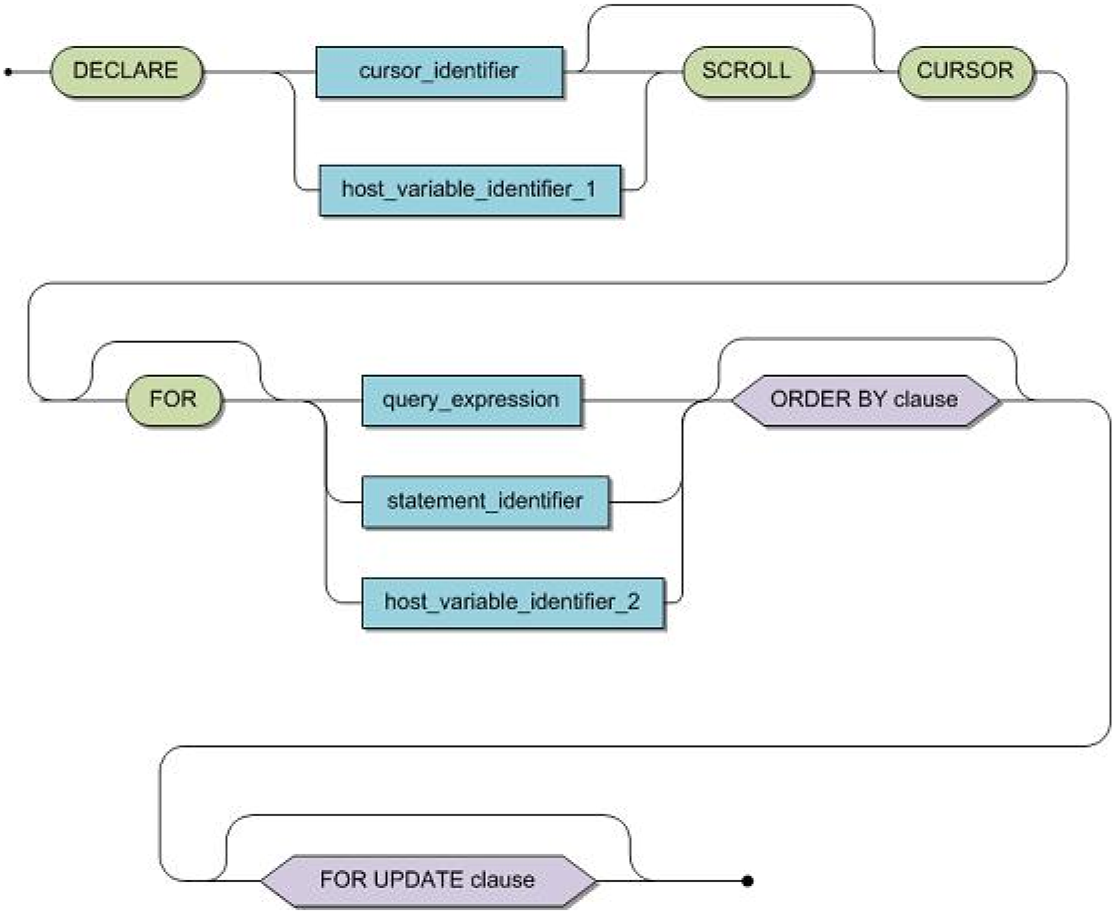
Function

The DECLARE CURSOR statement associates a query expression and hence a resultant table with a cursor identifier. The statement only defines the contents of the resultant table; it does not establish it.

Invocation

Embedded Mode ✓	Dynamic Mode	Interactive Mode

Syntax



cursor_identifier	A valid identifier of no more than 32 characters, and which has not previously been used as a cursor identifier within the same compilation unit.
host_variable_identifier_1	A valid single host variable which is used to contain a unique dynamic cursor identifier.
query_expression	The specification of the resultant table associated with this cursor.
statement_identifier	A valid SQL identifier identifying a SELECT statement which has

	previously been prepared.
host_variable_identifier_2	A valid single host variable. The value of the host variable must be the value returned by the PREPARE statement and thus identify the prepared statement.
ORDER BY clause	The specification of a user-defined ordering of the resultant table. Otherwise the resultant table is not ordered.
FOR UPDATE clause	The explicit indication that this cursor is to be used in conjunction with either an UPDATE and/or DELETE WHERE CURRENT OF cursor-id statement.

Description

A cursor can be declared either as static using a static DECLARE CURSOR statement or as dynamic using a dynamic DECLARE CURSOR statement.

The static DECLARE CURSOR statement

A static DECLARE CURSOR statement associates a query expression and the definition of a resultant table with an SQL identifier, namely the cursor identifier . The DECLARE CURSOR statement is only a definition. The OPEN statement associated with this cursor establishes the resultant table at execution time.

Although the characteristics of the derived column list are completely defined, the actual number of rows returned is unknown until execution time. In other words the format of each row associated with the cursor is known but the number of rows established upon opening the cursor is not. This is in direct contrast to the SINGLE ROW SELECT where by definition only one row may be returned. The host program is, therefore, not in a position to receive all the data established upon opening and must sequentially execute associated FETCH statements in order to retrieve one row at a time. This is the classic DECLARE-OPEN-FETCH cycle. The cursor identifier can be thought of as a pointer into the resultant table identifying the row currently under consideration. In general, executing an associated FETCH statement advances the pointer by one row.

In addition to the OPEN and FETCH statements, other associated statements are positioned UPDATE, positioned DELETE, and CLOSE.

The query expression defines the resultant table associated with the cursor. In theory, the expression can be unlimited in complexity. Certain query expressions are considered to be "updatable," i.e., the positioned DELETE or UPDATE statements are valid for this cursor.

Updatable Cursors

For a cursor to be updatable the following rules must be observed:

- The specification of a UNION operator in a query expression is not allowed. Therefore, the expression must consist of only one query specification.
- Derived columns in the derived column list must be based on base tables not views. No operators, functions or literals are allowed in the derived column list.
- No column may be specified more than once in the derived column list of the query specification.
- The specification of DISTINCT in the derived column list is not allowed.
- A grouped or joined query specification is not allowed.
- If a subquery is specified, it may not reference the same table as that one referenced in the outer query, i.e the table which would be the subject of any amendment statement.
- If the query specification is derived from a view that view must be updatable.
- An ORDER BY clause is not specified.
- A FOR FETCH ONLY clause is not specified.

If the above conditions for a read-only cursor have been met, positioned UPDATE or DELETE statements will result in compilation errors.

Non-Updatable Cursors

A static cursor can be explicitly declared as being non-updatable by use of the FOR FETCH ONLY clause. In such a case, the use of positioned UPDATE or DELETE statements associated with the cursor is not allowed. Furthermore rows will not be locked once they are established regardless of the default locking specification.

Alternatively, a static cursor can be declared as FOR UPDATE, as long as it is updatable of course. In such a case, rows will be locked regardless of the default locking specification. In general, this clause need not be specified. However, if the associated UPDATE or DELETE statement is actually in a separate compilation unit, as is possible with Adabas SQL Gateway Embedded SQL, then this clause is required in order to avoid a runtime error.

If neither a FOR FETCH ONLY clause nor a FOR UPDATE clause is specified and there are no associated UPDATE or DELETE statements within the same compilation unit, then the resulting rows will or will not be locked according to the system default locking specification.

Similar behavior can be ensured for a dynamic cursor by appending the clause to the dynamic SELECT statement. A column specification list is optional and indeed has no effect.

The Dynamic DECLARE CURSOR Statement

A dynamic DECLARE CURSOR statement associates a dynamically created and prepared SELECT statement with a cursor identifier. The prepared SELECT statement can be identified either by a hard-coded SQL identifier or by a host variable containing the unique statement identification provided by the relevant PREPARE statement.

The dynamic DECLARE CURSOR statement thus associates this previously prepared SELECT statement with a cursor identifier . The cursor can be identified in the normal way or by a host variable, which Adabas SQL Gateway Embedded SQL fills with a unique cursor identifier.

Limitations

The syntax elements host variable identifier 1, host variable identifier 2 and statement identifier are not valid within a static DECLARE CURSOR statement.

Within a dynamic DECLARE CURSOR statement such host variables must be of data type character-string.

Any ORDER BY clause, FOR UPDATE clause is part of the prepared SELECT and the use of these clauses is not valid within a dynamic cursor statement, but only in a static DECLARE CURSOR statement.

ANSI Specifics

The use of the FOR UPDATE and FOR FETCH ONLY clauses.

The DECLARE CURSOR statement must precede any other associated statement in the source. All associated statements must be contained within one compilation unit.

Adabas SQL Gateway Embedded SQL Specifics

The physical order of the associated statements within a compilation unit is irrelevant. The OPEN statement must be present in the same compilation unit as the DECLARE statement although its relative position is irrelevant. Associated UPDATE, DELETE, FETCH and CLOSE statements need not be in the same compilation unit. However, such a program design is more error prone as full compilation checks cannot be performed.

The physical position of any associated PREPARE statement relative to the dynamic DECLARE CURSOR statement is irrelevant.

The function of a dynamic DECLARE CURSOR statement can also be accomplished by an extended OPEN statement. This saves one request to Adabas SQL Gateway Embedded SQL, since a dynamic DECLARE CURSOR statement is an executable statement.

Example

The following example declares a cursor to select all the cruise and start dates for every cruise that leaves BARBADOS.

```
DECLARE cursor1 CURSOR FOR
   SELECT cruise_identifier,start_date FROM cruise
      WHERE start_harbor = 'BARBADOS';
```

To declare a cursor to list all the start harbor's in ASCENDING alphabetical order and each related cruise id, for each cruise that costs less than 1000 the following syntax applies:

```
DECLARE cursor1 CURSOR FOR
   SELECT cruise_identifier,start_harbor FROM cruise
      WHERE cruise_price < 1000
      ORDER BY 2 ASC;
```

To ensure that the cursor as declared in the first example can only be used for retrieval the following syntax applies:

```
DECLARE cursor1 CURSOR FOR
   SELECT cruise_identifier,start_date FROM cruise
    WHERE start_harbor = 'BARBADOS'
    FOR FETCH ONLY;
```

DESCRIBE

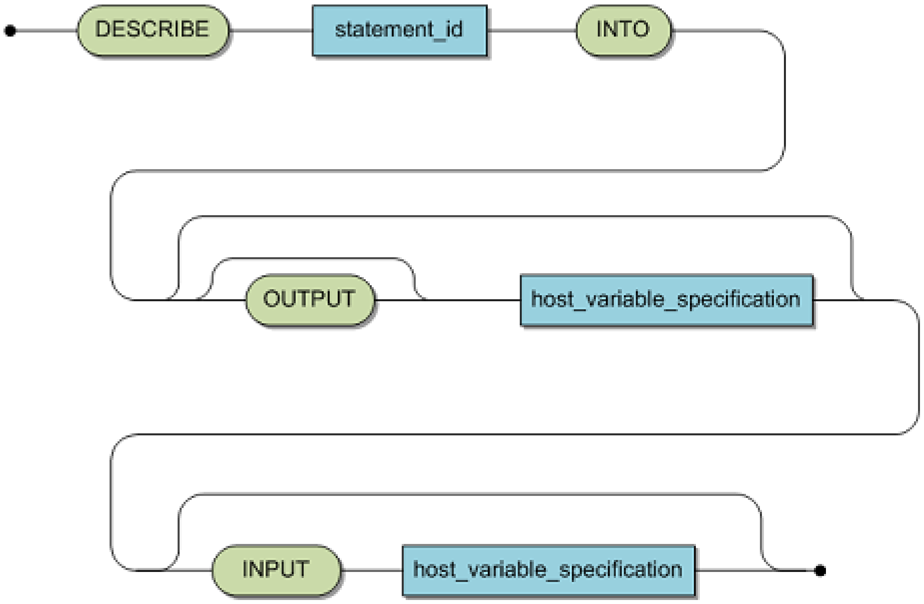
Function

The DESCRIBE statement makes information about a prepared statement available to the application program.

Invocation

Embedded Mode ✓	Dynamic Mode	Interactive Mode

Syntax



statement_identifier	A valid identifier denoting the prepared statement of which the information is to be retrieved.
host_variable_identifier	A valid single host variable identifier. It must have been defined in the application program according to the host language rules. The value of the host variable must be the value returned by the PREPARE statement and thus identifying the prepared statement.
OUTPUT hvu	The definition of the SQL descriptor area used to describe the expected output of the identified statement.
INPUT hvu	The definition of the SQL descriptor area used to describe the expected input of the identified statement.
host_variable_specification	A valid single host variable identifier and must have been defined in the application program according to the host-language-dependent rules. The value of the host variable must be the address of an SQL descriptor area (SQLDA).

Description

The DESCRIBE statement places information about the prepared statement identified by statement identifier or host variable identifier in one or two SQL descriptor areas.

The keyword OUTPUT is relevant only if the prepared statement is a SELECT statement. In this case, the SQL descriptor area indicated by host variable identifier 2 is filled with information concerning the elements in the derived column list of the SELECT statement. For each element in the derived column list, an element in the SQL descriptor area is filled. The elements in the derived column list are processed from left to right and the descriptive elements in the SQL descriptor area are filled in the that order.

The keyword INPUT is relevant only if the prepared statement contains host variable markers, i.e., "?". In this case, the SQL descriptor area indicated by host variable identifier 2 is filled with information concerning the host variable markers used in the prepared statement. For each host variable marker, an element in the SQL descriptor area is filled. The host variable markers are processed in the order that they appear in the prepared statements. The descriptive elements in the SQL descriptor area are filled in that order.

If the prepared statement is a SELECT statement where host variable markers have been used, the usage of not only the OUTPUT clause but also the INPUT clause is recommended.

Limitations

The statement indicated by statement identifier or host variable identifier 1 must be a successfully prepared statement.

If not enough elements have been provided in the SQL descriptor area to cater for the total number of elements that need to be described, all the elements that can be catered for are described, the rest of the information is ignored. The actual number of elements required is returned in field SQLN in the SQLDA.

ANSI Specifics

None.

Adabas SQL Gateway Embedded SQL Specifics

This statement may be mixed with any other DML, DDL and/or DCL statements in the same transaction.

Example

```
DESCRIBE statement_identifier INTO
   OUTPUT :sqlda_address ;
```

DISCONNECT

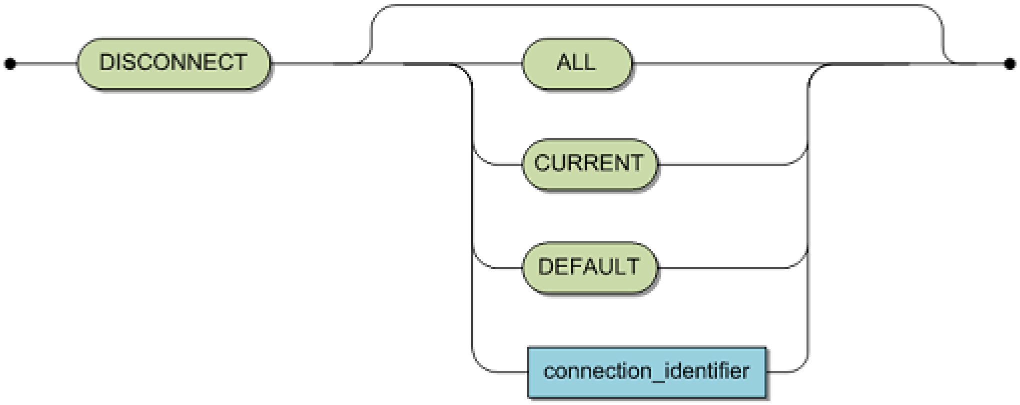
Function

The DISCONNECT statement explicitly terminates an SQL session between a user and the Adabas SQL Gateway Embedded SQL environment.

Invocation

Embedded Mode ✓	Dynamic Mode	Interactive Mode ✓

Syntax



connection_identifier	Can either be a character-string constant or single host variable identifier. The host variable must have been defined in the application program according to the host language rules and its value must be a character string. The maximum length is 32 characters.

Description

The DISCONNECT statement terminates an SQL session between an application program and Adabas SQL Gateway Embedded SQL. The DISCONNECT statement performs an implicit ROLLBACK.

DISCONNECT/DISCONNECT CURRENT	Terminate current SQL session. The previous syntax of the DISCONNECT statement is still supported and is represented as the DISCONNECT CURRENT statement.
DISCONNECT ALL	Terminates all SQL sessions. A DISCONNECT ALL statement is performed automatically by the exit handler of Adabas SQL Gateway Embedded SQL when terminating an application.
DISCONNECT DEFAULT	Terminates the SQL session with the default Adabas SQL Gateway Embedded SQL.
DISCONNECT connection specifier	Terminates the SQL session with the server specified by the connection identifier.

Limitations

None.

ANSI Specifics

None.

Adabas SQL Gateway Embedded SQL Specifics

The DISCONNECT statement is an Adabas SQL Gateway Embedded SQL extension.

The previous syntax of the DISCONNECT statement is still supported and is equivalent to Version's 1.3 (or higher)DISCONNECT CURRENT.

Example

The following example disconnects from the session identified by the connection specifier MYSESSION:

```
DISCONNECT :MYSESSION;
```

END DECLARE SECTION

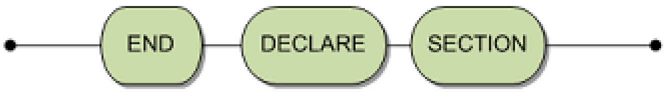
Function

This statement is the end delimiter for a host variable declaration section.

Invocation

Embedded Mode ✓	Dynamic Mode	Interactive Mode

Syntax



Description

This statement ends a host variable declaration section. Please refer to the BEGIN DECLARE SECTION statement earlier in this section for more details.

Limitations

Please refer to the BEGIN DECLARE SECTION statement earlier in this section for more details.

ANSI Specifics

Please refer to the BEGIN DECLARE SECTION statement earlier in this section for more details.

Adabas SQL Gateway Embedded SQL Specifics

Please refer to the BEGIN DECLARE SECTION statement earlier in this section for more details.

Example

```
BEGIN DECLARE SECTION
   char a
END DECLARE SECTION;
```

EXECUTE

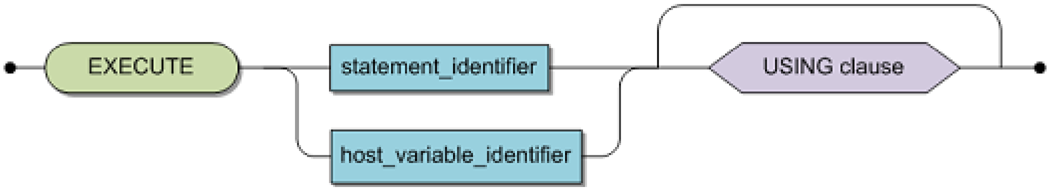
Function

The EXECUTE statement executes a prepared statement.

Invocation

Embedded Mode ✓	Dynamic Mode	Interactive Mode

Syntax



statement_identifier	A valid identifier denoting the name of the prepared statement which is to be executed.
host_variable_identifier	A valid single host variable identifier. It must have been defined in the application program according to the host language rules. The value of the host variable must be the value returned by the PREPARE statement and thus identifying the prepared statement.
USING clause	Defines an SQL descriptor area used to provide dynamic input variables if required by the statement to be executed.

Description

The EXECUTE statement executes the prepared statement identified by a statement identifier or host variable identifier. If the prepared statement contains host variable markers, then values must be provided to satisfy these. In this case, a USING clause is required.

Limitations

The statement indicated by statement identifier, host variable identifier must be a successfully prepared statement.

A previously prepared SELECT statement cannot be submitted to the EXECUTE statement.

ANSI Specifics

None.

Adabas SQL Gateway Embedded SQL Specifics

A host variable identifier can be used to identify the prepared statement.

This statement may be mixed with any other DML, DDL and/or DCL statements in the same transaction.

Example

The following example executes a prepared statement.

```
EXECUTE statement_identifier;
```

The following example executes a prepared statement that requires 3 values.

```
EXECUTE statement_identifier USING :hv1, :hv2, :hv3 ;
```

The following example executes a prepared statement where the input information is stored in the SQLDA.

```
EXECUTE statement_identifier USING DESCRIPTOR
:input_sqlda;
```

EXECUTE IMMEDIATE

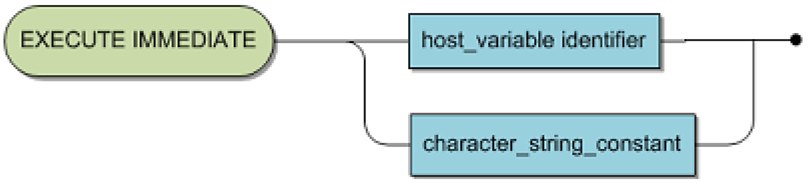
Function

The EXECUTE IMMEDIATE statement prepares and executes an SQL statement for immediate execution. After execution, the prepared statement is deleted.

Invocation

Embedded Mode ✓	Dynamic Mode	Interactive Mode

Syntax



host_variable_identifier	A valid single host variable identifier. It must have been defined in the application program according to the host language rules. The data type of the host variable must be a character string.
character_string_constant	A valid character-string constant.
EXECUTE IMMEDIATE	See Description below.

Description

The EXECUTE IMMEDIATE statement performs the following actions:

- COMPILATION: The SQL statement in a character-string representation is compiled into a prepared SQL statement. If an error is encountered by the SQL compiler which prevents the SQL statement being compiled successfully, an error is passed back to the application program in the SQLCODE field of the SQLCA. In this case, no prepared statement is created and the execution phase is not entered.
- EXECUTION: The prepared SQL statement is executed. If an error is encountered during the execution of the prepared statement, the error is passed back to the application program in the SQLCODE field of the SQLCA.
- DELETION: The prepared SQL statement is deleted. The prepared statement is deleted after execution. This means, that even if the exact same statement will have to be executed twice with two separate EXECUTE IMMEDIATE statements within the same transaction, it will have to be compiled twice.

Limitations

The character-string must contain one of the following statements:

COMMIT, CREATE, DELETE, DROP, INSERT, ROLLBACK, or UPDATE.

Host variable markers or references are not permitted in the statement.

ANSI Specifics

None

Adabas SQL Gateway Embedded SQL Specifics

This statement may be mixed with any other DML, DDL and/or DCL statements in the same transaction.

Example

```
EXECUTE IMMEDIATE 'DELETE FROM cruise' ;
```

GET DIAGNOSTICS

Function

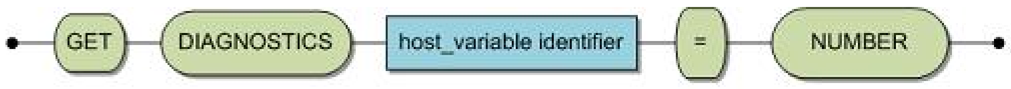
The GET DIAGNOSTICS statements are used to get and handle errors and warnings at SQL runtime. There are three GET DIAGNOSTICS statements:

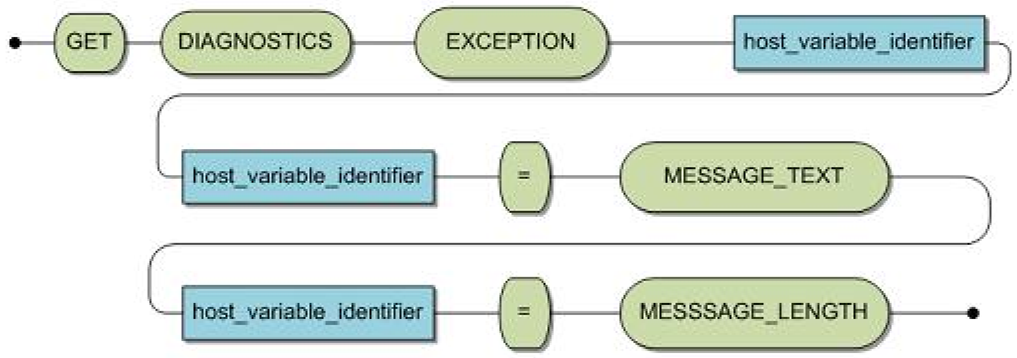
GET DIAGNOSTICS NUMBER
GET DIAGNOSTICS EXCEPTION MESSAGE_TEXT
GET DIAGNOSTICS ROW_COUNT

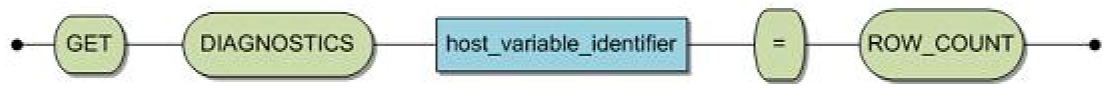
Invocation

Embedded Mode ✓	Dynamic Mode	Interactive Mode

Syntax







Description

The GET DIAGNOSTICS statements are used in combination with the SQLCODE variable to handle runtime SQL errors or warnings. The host variable in the GET DIAGNOSTICS NUMBER statement is of type integer and indicates how many errors or warnings are present for the executed SQL statement. The GET DIAGNOSTICS EXCEPTION MESSAGE TEXT statement has 3 host variables, a character string which will contain the error or warning text message, an integer containing the length of the host variable receiving the error or warning text message and an integer counter containing the message number to be retrieved.

GET DIAGNOSTICS statements should be used if the SQLCODE is not equal 0 and has one of the following values:

+100 indicating that no data were found.

< 0 indicating that a run-time SQL error occurred, or

> 0 indicating that SQL warnings have occurred.

The GET DIAGNOSTICS NUMBER statement should be used first to determine how many errors or warnings were received. This number can then be used as the step number in an application loop in which the GET DIAGNOSTICS MESSAGE TEXT statement is used to get the error or warning message text.

The GET DIAGNOSTICS ROW_COUNT statement may be used immediately after an INSERT, UPDATE, DELETE, SELECT INTO or FETCH statement to determine the number of rows affected by the preceding statement. The host variable to receive the row count should be of type integer.

ANSI Specifics

None.

Adabas SQL Gateway Embedded SQL Specifics

None.

Example

```
EXEC SQL BEGIN DECLARE SECTION;
int  conditionCount;
int  errNumber = 0;
int  errLen    = 512;
char errBuf [512];
EXEC SQL END DECLARE SECTION;
EXEC SQL GET DIAGNOSTICS :conditionCount = NUMBER;
for (errNumber = 1; errNumber <= conditionCount; errNumber++)
{
memset (errBuf, '\0', sizeof(errBuf));
EXEC SQL GET DIAGNOSTICS EXCEPTION
   :errNumber
          :errBuf = MESSAGE_TEXT,
          :errLen = MESSAGE_LENGTH;
printf ("%s\n", errBuf);

EXEC SQL BEGIN DECLARE SECTION;
int  rowCount;
EXEC SQL END DECLARE SECTION;
EXEC SQL GET DIAGNOSTICS :rowCount = ROW_COUNT;
```

FETCH

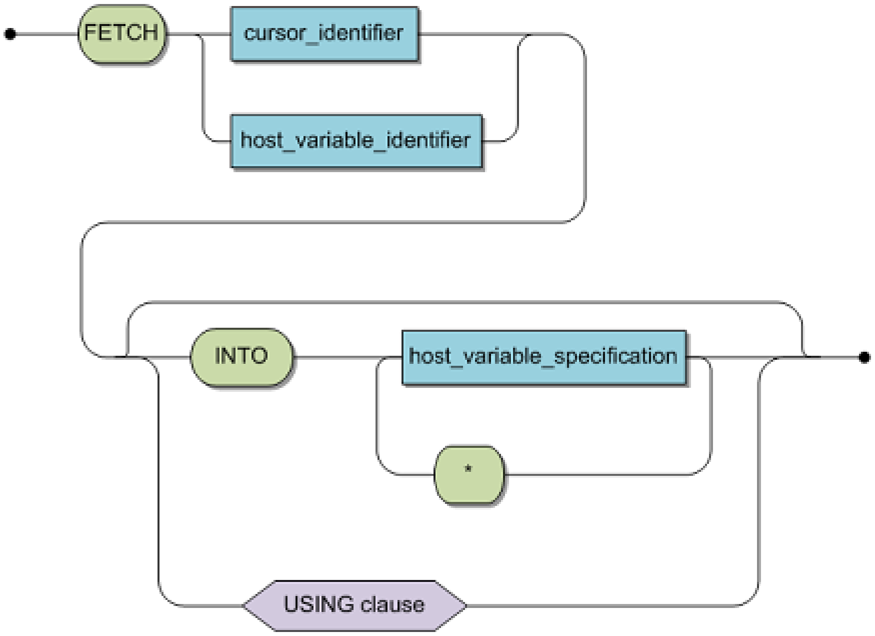
Function

The FETCH statement positions the cursor on a row within the resultant table and makes the values of that row available to the application program.

Invocation

Embedded Mode ✓	Dynamic Mode	Interactive Mode

Syntax



cursor_identifier	Identifies the cursor to be used in the FETCH operation.
host_variable_identifier	A valid single host variable identifier. It must have been defined in the application program according to the host language rules. The value of the host variable must be a valid cursor identifier. A host variable can be used as cursor identifier only if the cursor is a dynamically declared cursor.
host_variable_specification	A valid host variable specification. It must reference a structure and must have been defined in the application program according to the host language rules.
USING clause	Defines an SQL descriptor area used to receive data from the associated dynamic cursor.

Description

The FETCH statement performs two functions: it moves the cursor in the resultant table from top to bottom, one row at a time, and makes the relevant values of a row available to the application program according to the specification of the INTO clause or the USING clause. The mechanism used when the USING clause has been specified in USING Clause .

The FETCH statement changes the position of the cursor as follows:

- If the cursor is positioned before the first row of the resultant table (as would be the case if the cursor had just been opened), it is moved to the first row.
- If the cursor is positioned on a row of the resultant table, it is moved to the next one.
- If the cursor is positioned on the last row of an resultant table, it is moved past the last row and the SQLCODE field in the SQLCA is set to +100.
- If the row on which the cursor is positioned is deleted, the cursor is, then positioned in front of the next row in the table.
- A host variable specification which references a host variable structure is equivalent to individual host variable specifications which reference all the elements of a structure singularly.
- Each host variable corresponds to a resultant column of the resultant table of the cursor in question, where the first host variable is passed with the first column and so on.
- Each value of a resultant column is assigned to the corresponding host variable . The assignment operation follows the normal conversion rules as described in Expressions in the Adabas SQL Gateway User Guide.

Limitations

- The cursor must have been prepared and opened prior to the execution of the FETCH statement.
- The data type of a host variable must be compatible with the data type of its corresponding resultant column. If the data type is not compatible, an error occurs. The value of the unassignedhost variable is unpredictable.
- If the number of resultant columns is smaller than the number of host variables, as many host variables as possible are assigned the values of their corresponding resultant columns. The remaining host variables are left untouched.
- If the number of resultant columns is greater than the number of host variables , an error message (warning) is generated.
- A USING clause may only be used in association with a dynamic cursor.

ANSI Specifics

An INTO clause is mandatory, the USING clause must not be used. Only single host variable specifications are permitted.

Adabas SQL Gateway Embedded SQL Specifics

The OPEN statement and the FETCH statement can be in different compilation units (see also DECLARE CURSOR).

Example

The following example fetches data from a cursor and places the data into three host variables.

```
FETCH cursor_identifier
   INTO    :host_var1,
      :host_var2,
:host_var3 ;
```

Host Variable Specification

Host variables serve as a data exchange medium between Adabas SQL Gateway Embedded SQL and the application program written in a host language. When used in an SQL statement, a host variable specification has one of the following purposes:

- to identify a variable in the host language program which is to receive a value(s) from Adabas SQL Gateway Embedded SQL.
- to identify a variable in the host language program which is to pass a value(s) to Adabas SQL Gateway Embedded SQL.

A host variable is a single variable or structure declared in the host program.

A host variable identifier is used to identify a single host variable or structure from within an SQL statement.

A host variable specification consists of a host variable identifier and an associated optional INDICATOR variable and defines either a single variable, a structure, or an element in a structure.

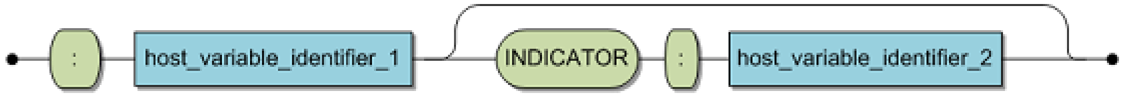
This section contains the following topics:

- Single Variables
- INDICATOR Variables
- Host Variable Markers
- Host Structures

Single Variables

The identified single host variable may actually be a single element within a host variable structure. Such a reference is not permitted in ANSI compatibility mode.

A single host variable is identified by a host variable identifier which has the following syntax:



host variable identifier 1	Identifies a single variable which is assigned any value but the NULL value.
host variable identifier 2	Identifies an INDICATOR variable, see INDICATOR Variables below.

Example:

Select the price of the cruise with a cruise ID of 5064 into a host variable.

```
SELECT cruise_price
   INTO :host_variable1
      FROM cruise
      WHERE cruise_id=5064;
```

INDICATOR Variables

An INDICATOR variable can serve as one of two purposes:

- Signifies the presence of a NULL value in a host variable assignment. If the NULL value is to be assigned to a target host variable specification then an accompanying INDICATOR variable must be present and is assigned a negative value to signify the NULL value. If the NULL value is to be assigned and the INDICATOR variable is missing, then a runtime error will occur.

The INDICATOR variable must be of a numeric data type with the exception of double precision, real and floating point data types. It must be of the appropriate data type for the host language.

Example:

Select the cancellation date of Contract 2025 into a host variable. (The column 'date_cancellation' could contain NULL values)

```
SELECT date_cancellation
   INTO :host_variable1 INDICATOR :host_variable2
      FROM contract
         WHERE contract_id=2025 ;
```

- Signifies that truncation has occurred in a host variable assignment. If truncation occurred during the assignment of a character string to a host variable, then the INDICATOR variable will show the total number of characters in the originating source prior to truncation.

SUMMARY:

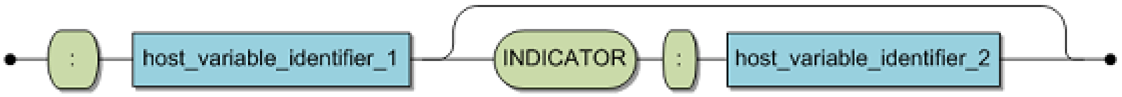
Indicator Value	Meaning	Host Variable Name
<0	indicates NULL value	undefined
=0	indicates non-NULL value	actual value
>0	number of characters	actual value in originating source

Host Variable Markers

A dynamic SQL statement can not contain host variables directly. It is, however, possible to provide a dynamic SQL statement after it has been prepared with value parameters at execution time. The dynamic statement must then contain a host variable marker for every host variable specification. A host variable marker is represented by a question mark (?) in the statement's source text. For details, see the section on Dynamic SQL.

Host Structures

A host structure is a C structure or a COBOL group that is referenced in an SQL statement. The exact rules to which a host structure must conform are described in the host language sections of the Programming Guide.



host variable identifier 1	Identifies a host structure. It can only be specified in the INTO clause of a single row SELECT or FETCH statement. A reference to a host structure is equivalent to a reference to each element in that structure.
	Each element of the host structure identified by host variable identifier 1 is a host variable which is assigned a value, if that value is not the NULL value.
host variable	An INDICATOR structure. An INDICATOR structure is a host

identifier 2	structure consisting of elements each identifying an INDICATOR variable.
	Each element of the INDICATOR structure identified by host variable identifier 2 identifies an INDICATOR variable, see also INDICATOR Variables in this section.

The i th element in the host structure indicated by host variable identifier 2 is the INDICATOR variable for the i th element in the host structure indicated by host variable identifier 1.

Note: Pointer expressions will be supported in the next release version.

Assume the number of elements in the host structure identified by host variable identifier 1 is m and the number of elements in the host structure identified by host variable identifier 2 is n:

- If m > n, then the last m-n elements in the host structure identified by host variable identifier 1 do not have an INDICATOR variable.
- If m < n, then the last n-m elements in the host structure identified by host variable identifier 2 are ignored.

Examples:

If two host structures have been declared, one for actual returned values and one for indicator values, and the variables 'struct1' and 'indicator1' identify these structures respectively, then the following syntax shows how values from a derived column list are entered into host variables (assuming that the host structures match the derived columns).

```
SELECT cruise_identifier,start_date,cruise_price
   INTO :struct1 INDICATOR :indicator1
      FROM cruise;
```

The following example inserts a resulting value from a query into a particular 'Element' of a defined structure. 'struct1' is a structure identifier that contains an element identified by 'price_element' and 'indicator1' is a structure identifier that contains the element identified by 'price_ind'.

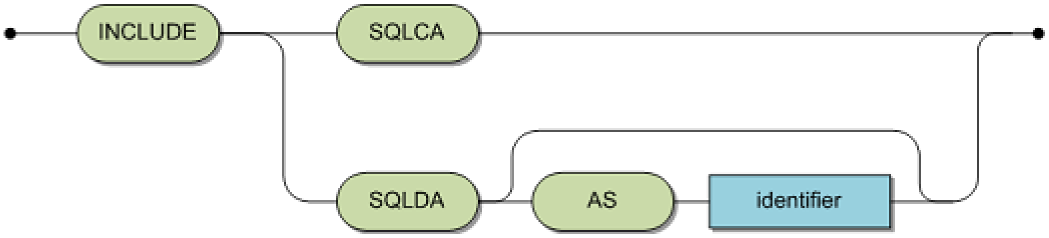
```
SELECT cruise_price
   INTO :struct1.price_element INDICATOR :indicator1.price_ind
      FROM cruise;
```

INCLUDE

Function

This statement includes the data description for the SQLCA or SQLDA.

Syntax



AS identifier	A host language specific identifier used to explicitly name the pointer variable to the SQLDA structure.

Description

The application programs must be able to determine if an SQL statement has been successfully completed or if it failed. The respective control values are available in the host variable structure called SQLCA. Although, such a structure may be defined and declared explicitly, it is much easier to let Adabas SQL Gateway Embedded SQL generate a definition and a declaration into the host program's source code. Such a generation will occur whenever the SQL statement INCLUDE SQLCA is specified. The position of this statement must conform to the rules of the declaration of host variables and the resulting structure will be represented by the identifier SQLCA. The SQLCA is not updated as a result of an INCLUDE statement.

Note: *An explicit identifier can not be specified for an SQLCA structure.*

Certain embedded dynamic SQL statements require the use of an SQLDA. Again, such a structure could be defined explicitly, but it is much easier to let Adabas SQL Gateway Embedded SQL generate a definition into the host program's source code. Only an SQLDA structure definition is generated along with a declaration of a pointer to such a structure. The user must actually provide an appropriate structure. The generated pointer will, by default, be identified by SQLDA unless the AS clause is supplied in which case the given identifier is used. The use of such an identifier within an appropriate SQL statement identifies this instance of the SQLDA pointer variable.

Limitations

This statement must be placed outside of a BEGIN DECLARE SECTION. It must also be positioned so that it obeys the rules regarding the declaration of host variables. In accordance with the host language rules governing the declaration of variables and their scope, any number of INCLUDE statements may be specified.

ANSI Specifics

The INCLUDE statement is not part of the Standard.

Adabas SQL Gateway Embedded SQL Specifics

The AS clause is an Adabas SQL Gateway Embedded SQL extension.

Examples

```
INCLUDE SQLCA;
INCLUDE SQLDA AS sql_pointer;
```

OPEN

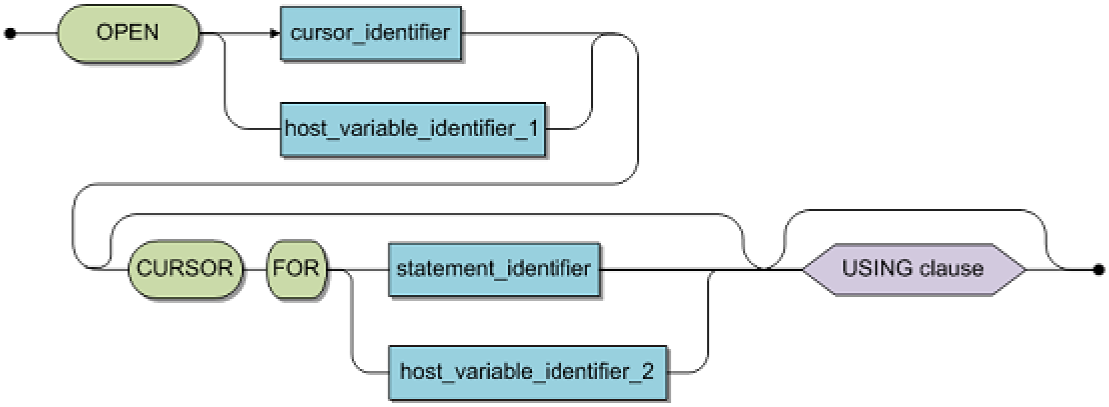
Function

An OPEN statement establishes the contents of a cursor.

Invocation

Embedded Mode ✓	Dynamic Mode	Interactive Mode

Syntax



cursor_identifier	Identifies the cursor to be used.
host_variable_identifier_1	A valid single host variable identifier. It must have been defined in the application program according to the host language rules. The value of the host variable must be a valid cursor identifier. A host variable can be used as cursor identifier only if the cursor is a dynamically declared cursor.
statement_identifier	A valid identifier denoting the name of the prepared statement.
host_variable_identifier_2	A valid single host variable identifier. It must have been defined in the application program according to the host language rules. The value of the host variable must be the value returned by the PREPARE statement and thus identifies the prepared statement.
USING clause	Defines an SQL descriptor area used to supply data to the associated dynamic cursor.

Description

The OPEN statement causes the contents of the associated resultant table to be established. The cursor is initially positioned before to the first row. The cursor can be identified by use of a host variable only if the cursor is declared dynamically. Likewise, the USING clause can be used to provide input values only if the cursor is a dynamically declared cursor. Alternatively, values can be provided by the direct use of host variables.

Limitations

The cursor to be opened must have been declared and must not be open. If the statement does not contain a CURSOR FOR clause the cursor must have been declared before.

The statement must be in the same compilation unit as the associated DECLARE CURSOR statement.

ANSI Specifics

- All cursors opened within a transaction are automatically closed by a COMMIT or ROLLBACK statement.
- The USING clause must not be used.
- The associated DECLARE CURSOR statement must physically precede the OPEN statement in the host program.

Adabas SQL Gateway Embedded SQL Specifics

- The CLOSE statement is the only statement apart from the DISCONNECT statement that closes a cursor.
- The cursor identifier can be given as a host variable if the cursor has been dynamically prepared.
- The OPEN statement may appear anywhere in relation to the associated cursor statement in the host language.
- DML statements must not be mixed with DDL/DCL statements in the same transaction.

Example

The following example opens a cursor.

```
OPEN cursor1 ;
```

The following example opens a dynamic cursor and provides values within host variables.

```
OPEN cursor1 USING :hv1, :hv2, :hv3 ;
```

PREPARE

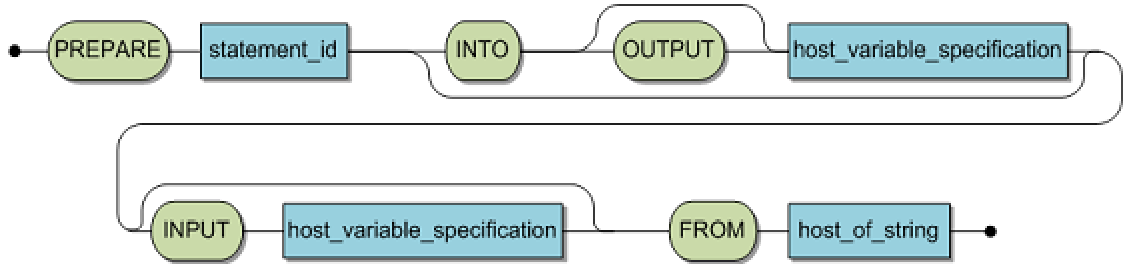
Function

The PREPARE statement prepares an SQL statement for later execution.

Invocation

Embedded Mode ✓	Dynamic Mode	Interactive Mode

Syntax



statement_identifier	A single identifier used to identify the statement to be prepared.
host_variable_identifier_1	A single host variable identifier of type character string. It receives the unique value which is either generated by Adabas SQL Gateway Embedded SQL or is defined in the application program. This value identifies the statement to be prepared.
OUTPUT hvu	The definition of the SQL descriptor area used to describe the expected output of the identified statement.
INPUT hvu	The definition of the SQL descriptor area used to describe the expected input of the identified statement.
character_string_constant	Explicitly contains the source statement to be prepared.
host_variable_identifier_2	A single host variable identifier which contains the character-string representation of the statement to be prepared.
host_variable_specification	A single host variable identifier. It must have been defined in the application program according to the host language rules. The value of the host variable must be the address of an SQL descriptor area (SQLDA).

Description

The PREPARE statement performs the following actions:

- COMPILATION: An SQL statement in a character-string representation is compiled into an executable SQL statement which is called the prepared statement. If an error is encountered by Adabas SQL Gateway Embedded SQL which prevents the SQL statement to be compiled successfully, an error is passed back to the application program in the SQLCODE field of the SQLCA. In this case no prepared statement is created.

- IDENTIFICATION: The prepared statement is kept for later execution. It is identified by the statement identifier provided by the application program or is generated by Adabas SQL Gateway Embedded SQL and passed back into host variable 1. If it is intended that the statement identifier is to be generated by Adabas SQL Gateway Embedded SQL it is necessary to initialize the variable with blanks or an empty string prior to execution. Otherwise, Adabas SQL Gateway Embedded SQL will use the actual (non-blank) value of the variable. This identification will be used to refer to the prepared statement in a DESCRIBE, DECLARE CURSOR or EXECUTE statement.
- DEALLOCATE PREPARE may be used to explicitly delete a prepared statement.
- DESCRIPTION: The nature of the prepared statement can be determined and conveyed to the user by supplying appropriate SQL descriptor area variables. The functionality of a DESCRIBE statement can be incorporated into the PREPARE statement. For a full description of this functionality refer to the relevant passages of the section DESCRIBE Statement in this section.

Limitations

The character-string must contain one of the following statements:

COMMIT, CREATE, DELETE, DROP, INSERT, ROLLBACK, SELECT, or UPDATE.

The statement string cannot contain host variables, instead it may contain host variable markers . A host variable marker is represented by a question mark (?). Host variable markers mark those places where values are to be inserted at the time the prepared statement is executed. For a description of how host variable markers are replaced by real values, see EXECUTE. In general, a host variable marker can be used in an SQL statement wherever a host variable can normally appear with the following restriction:

Note: *At compilation time, it must be possible to determine the data type resulting from the expression(s) contained in this statement.*

ANSI Specifics

None.

Adabas SQL Gateway Embedded SQL Specifics

This statement may be mixed with any other DML, DDL and/or DCL statements in the same transaction.

Example

The following example prepares the SQL statement with Id 'identifier1' to remove all rows from the table 'cruise'.

```
PREPARE identifier1 FROM
   'delete from cruise';
```

The following example prepares an SQL statement to delete a single row from the table cruise, where the row to be deleted is identified by it's cruise identifier given in a host variable. Note the use of the host variable marker `?'.

```
PREPARE statement_identifier FROM
   'delete from cruise where cruise_identifier = ?';
```

The following example prepares a dynamic SELECT statement where the format of the derived columns is not known until runtime, and hence, the SQLDA needs to be used.

```
PREPARE statement_identifier INTO OUTPUT :sqlda
   FROM :dyn_select_identifier ;
```

ROLLBACK

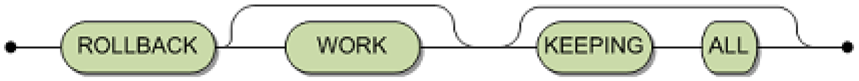
Function

The ROLLBACK statement terminates a transaction and removes all changes to the database that were made during the current transaction.

Invocation

Embedded Mode ✓	Dynamic Mode ✓	Interactive Mode ✓

Syntax



Description

The ROLLBACK statement terminates the current transaction and starts a new transaction. All changes to the database that have been made during the transaction are not applied and the database is as it existed at the time the transaction was started. All cursors that have been opened during the current transaction are closed.

The KEEPING ALL is currently ignored by the Adabas SQL Gateway.

Limitations

None.

ANSI Specifics

The keyword WORK is mandatory. The keywords KEEPING ALL are not supported.

Adabas SQL Gateway Embedded SQL Specifics

The keyword WORK is optional.

Example

```
ROLLBACK WORK ;
```

SELECT (SINGLE ROW)

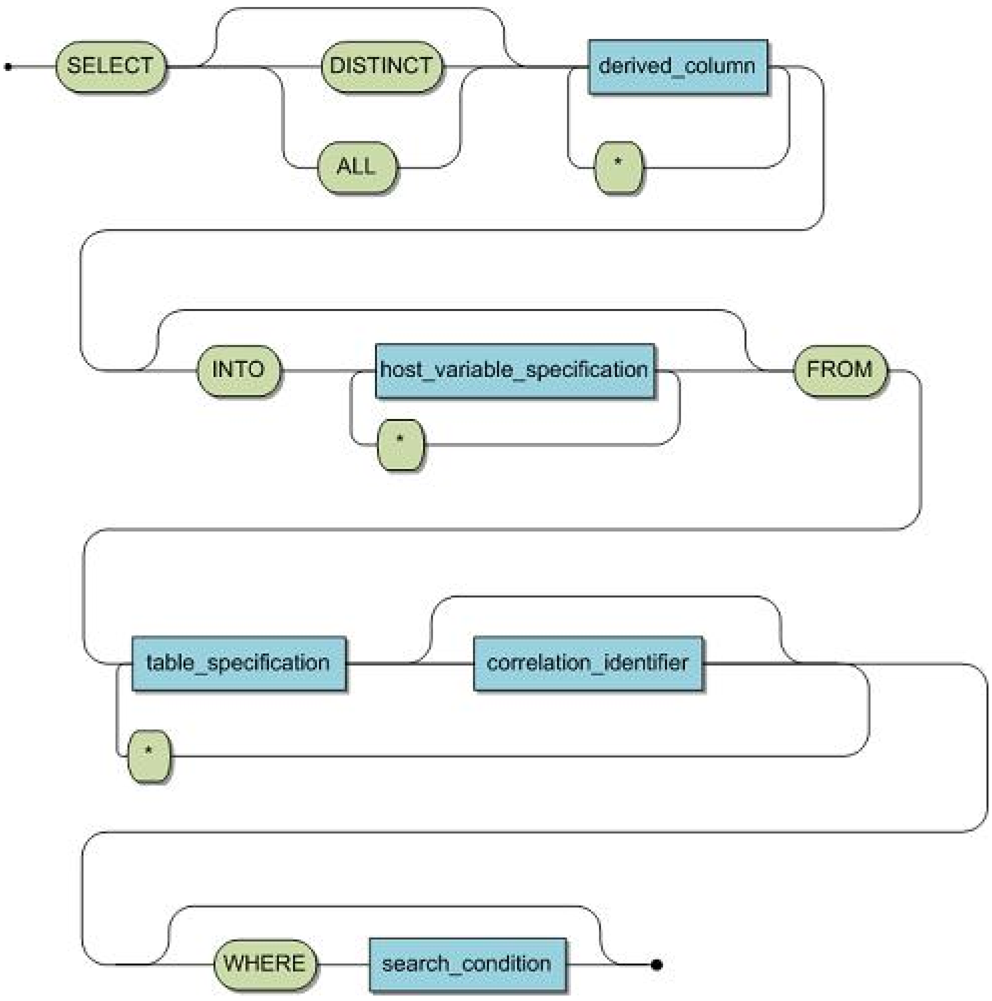
Function

The single row SELECT statement obtains a single row of data from the database according to the specified conditions.

Invocation

Embedded Mode ✓	Dynamic Mode	Interactive Mode

Syntax



Elements which are also part of the query specification are described in detail in Query Specification in the Adabas SQL Gateway User Guide.

derived_column	The corresponding columns in the final resultant table derived by the query. Derived columns are separated by commas and all of

	them together are referred to as the derived column list.
*	An abbreviated form of listing all derived columns of all tables in the table name list. In ANSI compatibility mode, it is not permitted to qualify the asterisk by using the correlation identifier or the table specification.
host_variable_specification	A single host variable. Only relevant for (INTO clause) single row SELECT. The host variables are intended to receive the returned data as specified by the SELECT statement's derived column list.
table_specification	A table specification as described in Table Specification in the Adabas SQL Gateway User Guide.
correlation_identifier	Alternative name for a particular table for use within the query and subqueries which are in scope.
WHERE clause	Search condition which candidate rows must fulfill in order to become part of the resultant table.

Description

The single row SELECT statement is used to obtain a single row of data from the database.

The single row SELECT statement can only be embedded and can only return one or no rows. A negative error code is returned in the sqlcode field of the SQLCA, if the resultant table actually contains more than one row. This is because the specified host variables in the INTO clause can only receive one row of data. A host variable specification which references a host variable structure is equivalent to individual host variable specifications which reference all the elements of a structure singularly.

The single row SELECT statement is the only invocation of a SELECT statement where an INTO clause is allowed and required. The only other way to specify an INTO clause is as a part of the FETCH statement. For details, refer to the FETCH Statement.

Limitations

A maximum of one row may be returned. The use of a valid INTO clause is required.

ANSI Specifics

None.

Adabas SQL Gateway Embedded SQL Specifics

DML statements must not be mixed with DDL/DCL statements in the same transaction.

Example

The following example determines how many persons the yacht 6230 can accommodate.

```
SELECT bunks
   INTO :bunks
      FROM yacht
      WHERE yacht_identifier = 6230;
```

SET

Function

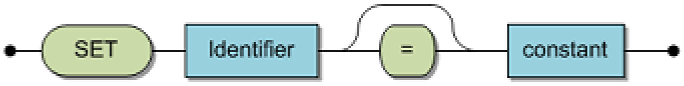
The SET statement is used to set the following client-specific parameters.

- Switch AUTOCOMMIT on/off
- Set a default catalog for an Adabas SQL Gateway Embedded SQL session
- Set a default schema for an Adabas SQL Gateway Embedded SQL session

Invocation

Embedded Mode ✓	Dynamic Mode ✓	Interactive Mode ✓

Syntax



Description

The SET statement can be used to set a client-specific parameter. The parameter to be set is provided as identifer and the value as constant.

The parameters which may be set are described in the table below.

Parameter	Usage	Further Information
Autocommit	Set AUTOCOMMIT on/off.	See SET AUTOCOMMIT.
Catalog	Set a default catalog.	See SET CATALOG in the SQL Gateway User Guide.
Schema	Set a default schema.	See SET SCHEMA in the SQL Gateway User Guide.

SET AUTOCOMMIT

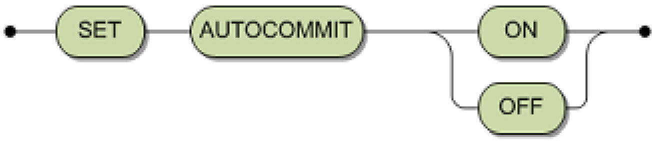
Function

The SET COMMIT statement is used to switch on/off AUTOCOMMIT.

Invocation

Embedded Mode ✓	Dynamic Mode	Interactive Mode ✓

Syntax



Description

With Adabas SQL Gateway, a database transaction, which can consist of one more SQL statements for data manipulation, remains open until an SQL statement for schema definition and manipulation or a COMMIT statement is issued. If a ROLLBACK statement is executed or if the SQL connection is terminated and the transaction is not explicitly terminated, which implies an implicit ROLLBACK statement, all changes to the database which were performed during the transaction will not be applied to the database.

- The SET AUTOCOMMIT ON statement can be used to cause Adabas SQL Gateway to issue an internal COMMIT after every SQL statement, thereby effectively closing the transaction after each SQL statement.
- The statement SET AUTOCOMMIT OFF can be used to switch off AUTOCOMMIT. The default behavior is as if an implicit SET AUTOCOMMIT OFF has been issued.
- The default setting is off.

ANSI Specifics

The SET AUTOCOMMIT statement is not part of the ANSI standard.

Adabas SQL Gateway Embedded SQL Specifics

None.

Example

```
set autocommit on ;
```

SET CONNECTION

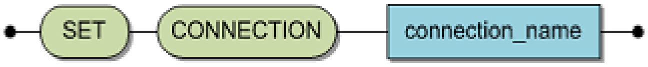
Function

The SET CONNECTION statement is used to specify which database connection to use for subsequent SQL statements.

Invocation

Embedded Mode ✓	Dynamic Mode	Interactive Mode ✓

Syntax



Description

SET CONNECTION connection_name

connection_name

The value for connection_name must match the connection name of an existing database connection specified in a previous CONNECT TO statement.

The connection_name can be either a literal or a host variable.

ANSI Specifics

The SET CONNECTION statement is not part of the ANSI standard.

Adabas SQL Gateway Embedded SQL Specifics

None.

Example

```
EXEC SQL CONNECT TO EMPLOYEES AS S1 USER U1;
EXEC SQL CONNECT TO VEHICLES  AS S2 USER U1;
EXEC SQL SET CONNECTION 'S1';
EXEC SQL SELECT PERSONNEL_ID FROM EMPLOYEES INTO :PERSONNEL_ID;
EXEC SQL SET CONNECTION 'S2';
EXEC SQL SELECT MAKE FROM VEHICLES INTO :MAKE;
EXEC SQL DISCONNECT 'S1';
EXEC SQL DISCONNECT 'S2';
```

USING Clause

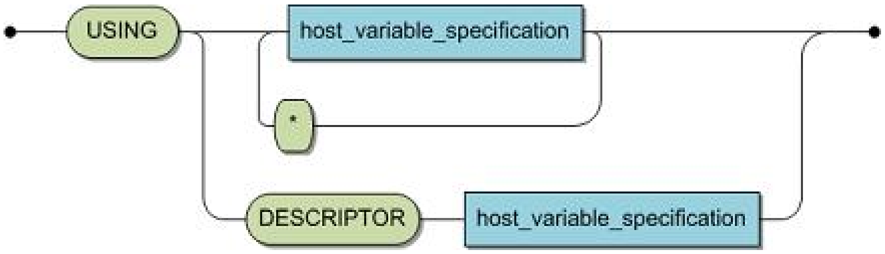
Function:

The USING clause is used to provide references to host variables for use in either a dynamic FETCH, OPEN or EXECUTE statement.

Invocation:

The USING clause is part of dynamic EXECUTE, FETCH or OPEN statements.

Syntax:



host_variable_specification	A valid host variable specification and must have been defined in an application program. The host variable specification may reference a structure.
host_variable_identifier	A valid single host variable identifier and which must be the address of an SQL descriptor area (SQLDA).

Description:

The USING clause defines a set of host variables for use either as value sources in a dynamic OPEN or EXECUTE statement or as target receptors in a dynamic FETCH statement.

A host variable specification, which references a host variable structure is equivalent to individual host variable specifications which reference all the elements of the structure singularly.

For a dynamic OPEN or EXECUTE statement, if the associated prepared statement contained host variable markers, i.e., `?' then these markers must be satisfied by use of a USING clause. Prior to use the referenced host variables must have been assigned appropriate values. Each referenced host variable provides a value for its corresponding host variable marker. The user must make sure that the host variables are supplied with the correct values and formats in the correct order.

For a dynamic FETCH statement, the host variables provided are intended to receive the results of the statement.

Host variables within the USING clause can be provided in two ways:

- by explicitly specifying a list of host variables. The number, type and order of the required host variables must be known at compilation time of the host program.
- by providing an SQL descriptor area. This facility enables a more dynamic approach to be adopted. The DESCRIBE statement provides the necessary information in the SQLDA for each host variable marker or derived column. The user must then provide a pointer in each field description which references an appropriate host variable. The number, type and order of the host variables can be completely unknown at compilation time of the host program. An SQL

descriptor area is identified by means of a host variable which contains the address of the SQLDA.

A host variable specification which references a host variable structure is equivalent to individual host variable specifications which reference all the elements of a structure singularly.

Limitations:

None.

ANSI Specifics:

The USING clause is not part of the Standard.

Adabas SQL Gateway Embedded SQL Specifics:

This is an Adabas SQL Gateway Embedded SQL extension.

Example:

The following example provides values for an an EXECUTE statement which requires the input of three values, for example, 'SELECT * FROM contract WHERE price IN (?, ?, ?). The USING clause provides these values.

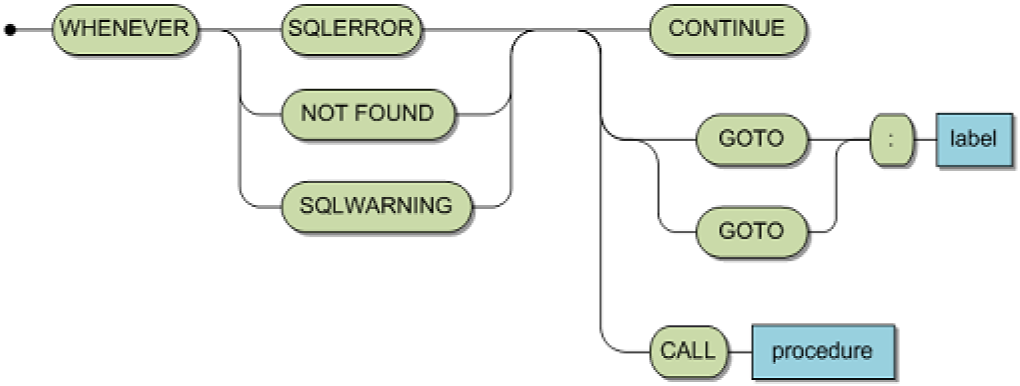
```
USING :hv1, :hv2, :hv3;
```

WHENEVER

Function

This statement specifies the action to be performed when an SQL statement results in an exception condition.

Syntax



label	Host language label.
procedure	Host language procedure, routine or function identifier.

Description

The variables in the SQLCA are updated during program execution and should be verified by the application program. This may be done in two different ways:

- by explicitly testing the contents of the appropriate variable in the SQLCA, usually the SQLCODE field.
- by specifying the SQL statement WHENEVER.

Note: *If no testing takes place, the default action for errors is to continue with the application program.*

An application program may contain any number of WHENEVER statements. The WHENEVER statement may appear anywhere after the declaration of an SQLCA. WHENEVER statements are pre-processed strictly in the order of their physical appearance in the source code, regardless of the execution order or conditional execution that the application program might imply. They will also only refer to the SQLCA which is currently in scope.

- Should two or more WHENEVER statements contradict each other, then the statement which was physically specified last is relevant for a particular SQL statement.
- The SQLCA is not updated as a result of a WHENEVER statement.

The condition is determined to be true according to the value of the variable SQLCODE and may be one of the following:

- NOT FOUND if the value is +100, indicating that no rows were found.
- SQLERROR if the value is negative, indicating an error.
- SQLWARNING if the value is positive other than +100, indicating a warning.

The action taken, should the condition be true, may be one of the following:

- CONTINUE ignores the exception condition and continues with the next executable statement.
- GOTO label continues the application program's logic with the statement identified by the label. The label must conform to the rules of the host language. The label may be prefixed with or without a `:' . GOTO may also be specified as GO TO.
- CALL procedure continues the application program's logic with the procedure identified by procedure. The procedure name must conform to the rules of host language. The procedure may not specify any host language parameters.
- Generally, a single WHENEVER statement will be valid for all SQL statements in the program. If an error occurs, control can be passed to an error handling routine. If SQL statements are to be executed from within this error handling routine, they too are subject to the conditions of the relevant WHENEVER statement. This means, if an error occurs during execution of the called error handling routine, an attempt will be made to call this same routine again, because the initial WHENEVER statement is still valid. This situation can be avoided by having a second WHENEVER statement in the error handling routine which specifies the option CONTINUE. It is recommended to test the SQLCA explicitly within the error routine.

Limitations

For the ANSI 74 standard (Embedded SQL setting COBOL II = off) every SQL statement is treated as if the optional period was coded. That means the generated code will always be terminated with a period. It is not possible to code more than one SQL statement in an IF statement.

ANSI Specifics

The SQLWARNING condition and the CALL option are not part of the Standard.

Adabas SQL Gateway Embedded SQL Specifics

This is an Adabas SQL Gateway Embedded SQL extension.

Example

The following example continues normal execution of a program if, an SQL query returns no rows.

```
WHENEVER NOT FOUND CONTINUE;
```

The following example continues program execution at another point (where that point is specified by a label) whenever an SQL statement produces a warning.

```
WHENEVER SQLWARNING GOTO label_name;
```

The following example diverts a program flow to a procedure whenever an SQL statement produces an error.

```
WHENEVER SQLERROR CALL procedure_name;
```

Error Messages

CODE	TEXT
0	Success.
1-99	Adabas nucleus response code. Please consult the Adabas documentation for more information.
100	End of recordset
101-255	Adabas nucleus response code. Please consult the Adabas documentation for more information.
554	Additional error information, such as "additions 2" or subcode.
1001	Value of option %s must be an integer greater than zero.
1002	Unknown option: %s
1003	Error in command line syntax: %s
1004	Password missing.
1005	Unknown value for COBOL standard: %s
1006	Unknown value for compatibility. %s
1007	Not a suitable precompiler source.
1008	Missing source file.
1009	Missing value for %s
1010	Output line length corrected to minimum supported %s bytes\n.
1011	Unknown value for host language: %s
1051	Cannot open include file: %s
1052	Cannot open input file: %s
1053	Cannot open output file:%s
1054	Internal function type error: %s
1055	The line is truncated to %s characters.
1101	Comment not closed.
1102	Macro name not defined: %s
1103	Illegal use of array declarator.
1104	Redeclaration of a name with different type: %s
1105	Redeclaration of a name with smaller size: %s
1106	Invalid declaration.
1107	Variable or typedef name expected.
1108	Illegal use of parentheses in declarator.
1109	No type definition for typedef name: %s

1110	Initializer syntax error.
1111	SQLCODE and SQLCA both declared in the same program unit.
1112	Parameter not declared: %s
1113	Parameters %s and indicators %s are of different struct type.
1114	Invalid preprocessor command.
1115	Invalid combination of specifiers.
1116	Statement not terminated.
1117	Statement syntax error.
1118	Quoted string not closed.
1119	Newline in string constant.
1151	Token longer than output line length.
1201	Invalid cursor name.
1251	Invalid level number: %s
1252	Invalid picture (Size of precision/scale = 0)
1253	Invalid picture (Digits count > 18)
1254	Invalid picture (Dimension)
1255	Invalid picture (Blank without when/zero)
1256	Invalid picture (Binary > 18 digits or Scale > 0)
1257	Invalid picture (Occurs)
1258	Invalid picture (Packed Decimal)
1259	Invalid picture (Display)
1260	Invalid picture (Comp Type)
1261	Invalid picture (Comp-1 Type)
1262	Invalid picture (Comp-2 Type)
1263	Invalid picture (Comp-5 Type)
1264	Invalid picture (Sign without Leading/Trailing)
1265	Invalid picture (Sign Type)
1266	Invalid picture (Usage)
1267	Invalid picture (Picture symbol)
1268	Invalid picture (Db Character symbol)
1269	Invalid picture (Character definition)
1270	Invalid picture (Number definition)
1271	Invalid picture (Clause keyword)
1301	Cannot get dynamic memory.

1302	Internal string error.
3001	Invalid input: %s
3002	Unsupported command.
3003	Internal application error: %s
3004	Not yet implemented: %s
3005	Unable to allocate dynamic memory.
3006	Incorrect number of arguments: %s
3007	Null argument passed: %s
3008	Index out of bounds: (%d)
3009	An OS specific function call failed. Please contact technical support.
3010	An internal buffer size has been exceeded.
3011	No element exists for the given key: %s
3012	Path specification exceeds system limit.
3013	Path not found: %s
3497	No exception exists for the given condition: %d
3498	Info: %s
3499	LOGIC ERROR: %s
3501	Unable to initialize TCP/IP.
3502	Unable to create a socket: (%d)
3503	Unable to find host: %s
3504	Connection is already active.
3505	Unable to connect: (%d)
3506	No active connection.
3507	Unable to send to socket: (%d)
3508	Send failure, not all bytes written.
3509	Unable to read from socket: (%d)
3510	Read failure, incorrect packet size.
3511	Packet processing error: pack.
3512	Packet processing error: unpack.
3513	Invalid port specification: (%d)
3514	IO Buffer lacks sufficient capacity to accept data.
4001	Session already defined: (%s)
4002	Not connected. Session not found: (%s)
4003	Unable to connect to SQL driver: %s

4004	Unsupported SQL statement type: (%d)
4005	Invalid SQL content argument.
4006	Invalid recordset descriptor.
4007	Unsupported SQL data type: (%d)
4008	SQL Driver already connected.
4009	Invalid attempt to deallocate a context before the connection has been closed.
4010	SQL cursor already defined: (%s)
4011	SQL cursor not defined: (%s)
4012	No DSN specified in CONNECT statement.
4013	Invalid connect parameters. DSN probably not registered.
4014	SQL statement not defined: (%s)
4015	Unsupported connection command id: (%d)
4016	Invalid statement handle: (%d)
4017	SERVER ERROR: ODBC: (%s) NATIVE: (%d): %s
4018	Null value returned from server and no indicator present.
4019	Value returned from server truncated and no indicator present.
4020	Invalid type conversion truncated integral type.
4021	Unsupported type conversion.
4022	No HOST ADDRESS specified in CONNECT statement.
4023	Unsupported SQL command.
4024	Invalid handle length (minimum is 32): (%s)
4025	Invalid static cursor name. Value must be embedded in the statement.
4026	Session (%s) already owned: (%s)
4027	Invalid session id. (%s)
4028	Context (%s) is not associated with a session.
4029	Invalid handle, is NULL: (%s)
4030	%s identifier contains illegal characters: (%s)
4031	Null host var passed for fetch column: (%d)
4032	A CONNECT statement must be successfully executed first. SQL execution not possible yet.
4033	Statement handle already associated: (%s)
4034	Incorrect number of parameter arguments: Received: (%d), Expected: (%d)
4035	Data supplied as parameter would have been truncated.

4036	Indicator variable type must be a non-nullable type of exact numeric or integer.
4037	Invalid Numeric/Decimal specification.
4038	Single select returned more than one row.
4039	NULL value assigned to a non-nullable type.
4040	Catalog identifier length exceeds maximum.
4041	Schema identifier length exceeds maximum.
4042	Context is already active: (%s)
4043	Unsupported connect command: (%d)
4044	Indicator value exceeds maximum/minimum.
4045	Invalid SQL command for Execute Immediate.
4501	Parser error: Unrecognized pattern: (%d) for command: (%d).
4502	Parser error: NULL token passed.
4503	Invalid operation. Parse in progress.
4504	SQL statement length exceeds maximum: (%d)
4505	SQL statement is empty.
4506	Invalid command statement.
4507	Illegal to pass the %s as a host variable in this context.
4508	Illegal token, host variable expected: %s
4509	Open called for an undeclared cursor: %s
4510	Invalid token: %s
4511	Invalid token (%s) expected: %s
4512	Unbalanced quotes in statement string.
4513	Unexpected end of buffer.
4514	Unbalanced parenthesis in statement string.
4515	Invalid SQL statement.
4516	Invalid SQL statement for %s command.
4517	Expected token not found: %s
4518	Indicator variable is of an invalid type.
4519	SQL statement string not allowed here.
4520	Identifier contains illegal characters: %s
4521	Identifier (%s) has an invalid length. Maximum: %d
4522	List in statement missing separator.
4523	Identifier contains illegal characters or is not properly quoted: %s

4524	Unable to set catalog (%s) during parse: [%s]
4525	Unable to set schema (%s) during parse: [%s]

ACEAPI

ACAPI General Information

This section describes ACEAPI, the Adabas SQL Gateway Application Programming Interface.

This interface provides Dynamic SQL access to supported Database Systems from any High Level programming language.

The installation directory contains sample programs, header files, copybooks, scripts and JCL for C, COBOL and PL/I which demonstrate the usage of each ACEAPI function against a standard Software AG 'employees' table. The program logic flow in each program is identical.

ACEAPI Argument Functions

SQL_CTR_BLOCK

```
typedef  struct
{
        int sqlcode;
        struct SQLDA * input_sqlda_ptr;
        struct SQLDA * output_sqlda_ptr;
        int nr_tuples_modified;
        void * context_ptr;
        int echo_warning;
        char error_text [512 + 1];
}
SQL_CTR_BLOCK;
```

The SQL_CTR_BLOCK is used to pass a common set of parameters to the API functions.

SQL_CTR_BLOCK Variable	Description
sqlcode	4 byte integer field containing the sqlcode returned by the function call.
input_sqlda_ptr	4 byte pointer to a structure of type SQLDA. This should be declared as NULL if an input sqlda is not required by the function being called.
output_sqlda_ptr	4 byte pointer to a structure of type SQLDA. This should be declared as NULL if an output sqlda is not required by the function being called.
nr_tuples_modified	4 byte integer field containing the count of rows affected by an insert, update or delete operation.
context_ptr	4 byte pointer to a structure of type SAGContext. Used by a multi-threaded application to specify the context for the function call. The structure must be zero filled prior to the first function call in a specific context. In a normal single threaded environment, context_ptr may be declared as NULL.
echo_warning	4 byte integer field containing 0 or 1. If NO (0) all warnings (sqlcode > 0 except for 100) will be changed to sqlcode = 0. If YES (1) all warnings will be returned with their original sqlcode.
error_text	512 byte character field containing the error text returned by the requested function for errors (sqlcode < 0) and warnings (sqlcode > 0) including sqlcode = 100. If echo_warning is set to NO, error_text will not be returned.

SAGColumn

```
typedef struct
{
        int sqlnamel;
        int sqlnamet;
        int sqlnamer;
        unsigned char sqlnamed [128];
}
SAGColumn;
```

SAGColumn is a structure used to describe a column declared in an SQLVAR.

SAGColumn Variable	Description
sqlnamel	4 byte integer field containing the length of the column name declared in structure member sqlnamed.
sqlnamet	4 byte integer field containing the type of the column name declared in structure member sqlnamed.
sqlnamer	4 byte integer field. Reserved for internal use.
sqlnamed	128 byte character field containing the data name of the column within the Catalog.

SAGContext

```
typedef  struct
{
        SAGPointer multithreaded_context_pointer;
        int async_id [2];
        SAGPointer itc_ptr;
        int reserved [10];
}
SAGContext;
```

SAGContext is used to support multithreaded applications. It is the application's responsibility to hold the context in a manner which allows the API to be called from different threads during a session. The structure must be initialized to NULLS prior to being used for the first time. Subsequent modifications of SAGContext by the application may have undesirable side-effects and should be avoided.

SAGPointer

```
typedef union
{
        void * ptr;
        void * host_address;
        int address_space [2];
}
SAGPointer;
```

SAGPointer is used as a pointer to column host variables and indicators declared in an application program.

SAGPointer Variable	Description
ptr (redefined as host_address)	8 byte pointer to a host variable or indicator declared in an application program. Example usage : **C/C++ :** char PERSONNEL_ID [8 + 1]; API_OUTPUT_SQLDA->sqlvar [0].sqldata.ptr = (void *)PERSONNEL_ID; **COBOL :** 01 PERSONNEL-ID PIC X(8) VALUE SPACES. SET PTR OF SQLDATA OF SQLVAR OF API-OUTPUT-SQLDA(1) TO ADDRESS OF PERSONNEL-ID **PL/I :** DCL 1 PERSONNEL_ID CHAR (8) INIT (' ') VARYINGZ; API_OUTPUT_SQLDA_PTR -> API_OUTPUT_SQLDA.SQLVAR(1).SQLDATA.PTR = ADDR(PERSONNEL_ID);

SQLDA

```
typedef struct sqlda;
{
        unsigned char sqldaid [8];
        int sqldabc;
        short sqln;
        short sqld;
        struct sqlvar sqlvar [1];
}
SQLDA;
```

The SQLDA is a structure that defines a multi-column file.

SQLDA variables and fields

An SQLDA consists of four variables (sqldaid, sqldabc, sqln and sqld), followed by an arbitrary number of SQLVARs.

SQLDA Variable	Description
sqldaid	8 byte character field containing an eye catcher for use in storage dumps. An SQLDA dynamically allocated by a call to SAGQPREP will contain the 7 byte reserved value 'SQLDADA' which must NOT be modified by the application. The 8th byte of the field may be modified by the application if required. A static SQLDA previously allocated by the application can contain any value except for 'SQLDADA*'.
sqldabc	4 byte integer field containing the length of the SQLDA. Not required.
sqln	2 byte integer field containing the total number of occurrences of SQLVAR.
sqld	2 byte integer field containing the number of columns described by occurrences of SQLVAR.

SQLVAR

```
typedef struct sqlvar
{
        int sqltype;
        int sqllen;
        int reserved;
        short internal;
        short sqlindlen;
        int sqlindtype;
        SAGPointer sqlind;
        SAGPointer sqldata;
        SAGColumn sqlname;
}
SQLVAR;
```

SQLVAR is a structure declared within an SQLDA (input or output) describing each column of a result row.

Each occurrence of SQLVAR describes one column of the result row being transferred to the client application.

An SQLVAR occurrence consists of nine fields:

SQLVAR Variable	Description
sqltype	4 byte integer field containing the data-type of the variable being used for data transfer for the column, and whether NULL values are supported.
sqllen	4 byte integer field containing the length (or precision and scale for DECIMAL and NUMERIC types) of the variable being used for data transfer for the column. Please see following section 'Decoding and Encoding SQLLEN' for more information regarding the usage of this field.
reserved	4 byte integer field. Reserved for internal use.
internal	2 byte integer field. Reserved for internal use.
sqlindlen	2 byte integer field containing the length of the variable being used as the column indicator.
sqlindtype	4 byte integer field containing the data-type of the variable being used as the column indicator.
sqlind	8 byte pointer structure of type SAGPointer containing the address of an indicator.
sqldata	8 byte pointer structure of type SAGPointer containing the address of the of the variable being used for data transfer.
sqlname	140 byte aggregate structure of type SAGColumn containing the length, type and name of the column within the Catalog.

Decoding and Encoding SQLLEN

The sqllen field of the sqlvar contains different values based on the sqltype of the column. For all data types except DECIMAL and NUMERIC the value contained in sqllen is the actual length of the column.

The sqllen for DECIMAL and NUMERIC types contains the precision, scale and length (identical to precision) in the following format :

Byte 0 - Precision / Length
Byte 1 - Scale
Byte 2 - Scale
Byte 3 - Precision / Length

This format ensures that an application program can be moved between big-endian and little-endian platforms without change and ensures that the appropriate byte order is maintained.

There are two methods available to decode and encode SQLLEN.

1. (Preferred method) Use the ACEAPI Utility Functions 'SAGQDECO' and 'SAGQENCO'. These functions can be called from applications written in any supported programming language. Please see the section 'ACEAPI Utility Functions' for more information regarding the usage of these two functions.
2. C and C++ users can use macros which are defined in the ACEAPI header file (aceapic.h).

   The sqllen can be decoded into precision, scale and length variables by using the macro DECODE_PRECISION_SCALE_LENGTH.

   The sqllen can be encoded with precision, scale and length by using the macro ENCODE_PRECISION_SCALE_LENGTH.

When encoding sqllen, it is essential that ALL four bytes are correctly filled if the above macros are not being used, for example when the application is written in a language other than "C".

The following SQLTYPES require decoding or encoding of the sqllen :

SQL_TYP_DECIMAL
SQL_TYP_NUMERIC
SQL_TYP_NUMERIC_LD
SQL_TYP_NUMERIC_TR
SQL_TYP_NUMERIC_SLD
SQL_TYP_NUMERIC_STR
SQL_TYP_NUMERIC_BINARY
SQL_TYP_NUMERIC_BIN_BE
SQL_TYP_NDECIMAL
SQL_TYP_NNUMERIC
SQL_TYP_NNUMERIC_LD
SQL_TYP_NNUMERIC_TR
SQL_TYP_NNUMERIC_SLD
SQL_TYP_NNUMERIC_STR
SQL_TYP_NNUMERIC_BINARY

SQL_TYP_NNUMERIC_BIN_BE

The sqllen field in the sqlvar should remain encoded and not be modified. If the length of the column is required, when allocating storage for example, the decoded values sqlprec or sqllength should be used and not the sqllen itself.

```
/*
Example to decode and encode 'sqllen' for DECIMAL and NUMERIC type columns. The DECODE_PRECISION_SCALE_LENGTH and ENCODE_PRECISION_SCALE_LENGTH macros are defined in the ACEAPI header file (aceapic.h).
*/
int sqlprec; /* Precision */
int sqlscale; /* Scale */
int sqllength; /* Length */

if (sqlda->sqlvar[column].sqltype == SQL_TYP_DECIMAL
                                                        || SQL_TYP_NUMERIC
                                                        || SQL_TYP_NUMERIC_LD
                                                        || SQL_TYP_NUMERIC_TR
                                                        || SQL_TYP_NUMERIC_SLD
                                                        || SQL_TYP_NUMERIC_STR
                                                        || SQL_TYP_NUMERIC_BINARY
                                                        || SQL_TYP_NUMERIC_BIN_BE
                                                        || SQL_TYP_NDECIMAL
                                                        || SQL_TYP_NNUMERIC
                                                        || SQL_TYP_NNUMERIC_LD
                                                        || SQL_TYP_NNUMERIC_TR
                                                        || SQL_TYP_NNUMERIC_SLD
                                                        || SQL_TYP_NNUMERIC_STR
                                                        || SQL_TYP_NNUMERIC_BINARY
                                                        || SQL_TYP_NNUMERIC_BIN_BE)
{
DECODE_PRECISION_SCALE_LENGTH(sqlda->sqlvar[column].sqllen,
                                                sqlprec,
                                                sqlscale,
                                                sqllength);
/*
Allocate storage using the sqllength returned by the macro DECODE_PRECISION_SCALE_LENGTH.
*/
sqlda->sqlvar[column].sqldata.host_address = calloc(1, sqllength);
..
ENCODE_PRECISION_SCALE_LENGTH(sqlda->sqlvar[column].sqllen,
                                                sqlprec,
                                                sqlscale,
                                                sqllength);
}
```

SQLDA Data Types

The following table contains the SQLDA data types. The data types are returned by the Adabas SQL Gateway Server for each column described by function 'SAGQDESC'. The relevant data type can be used to declare the type of host variable and indicator referenced by each column in the input or output SQLDA->SQLVAR.

SQLDA Data Type	Value	Description
SQL_TYP_CHAR	1	Character string
SQL_TYP_NUMERIC	2	Numeric
SQL_TYP_DECIMAL	3	Packed-Decimal
SQL_TYP_INTEGER	4	Integer (4 byte)
SQL_TYP_SMALLINT	5	Small Integer (2 byte)
SQL_TYP_FLOAT	6	Floating Point
SQL_TYP_LARGEINT	9	Large Integer (8 byte)
SQL_TYP_VARCHAR	12	Variable Length Character
SQL_TYP_NUMERIC_LD	20	Numeric Signed Leading
SQL_TYP_NUMERIC_TR	21	Numeric Signed Trailing
SQL_TYP_NUMERIC_SLD	22	Numeric Signed Leading Separate
SQL_TYP_NUMERIC_STR	23	Numeric Signed Trailing Separate
SQL_TYP_NUMERIC_BINARY	24	Numeric Binary (Little Endian)
SQL_TYP_NUMERIC_BIN_BE	25	Numeric Binary (Big Endian)
SQL_TYP_NCHAR	-1	Nullable Character string
SQL_TYP_NNUMERIC	-2	Nullable Numeric
SQL_TYP_NDECIMAL	-3	Nullable Packed-Decimal
SQL_TYP_NINTEGER	-4	Nullable Integer (4 byte)
SQL_TYP_NSMALLINT	-5	Nullable Small Integer (2 byte)
SQL_TYP_NFLOAT	-6	Nullable Floating-Point
SQL_TYP_NLARGEINT	-9	Nullable Large Integer (8 byte)
SQL_TYP_NVARCHAR	-12	Nullable Variable Length Character
SQL_TYP_NNUMERIC_LD	-20	Nullable Numeric Signed Leading
SQL_TYP_NNUMERIC_TR	-21	Nullable Numeric Signed Trailing
SQL_TYP_NNUMERIC_SLD	-22	Nullable Numeric Signed Leading Separate
SQL_TYP_NNUMERIC_STR	-23	Nullable Numeric Signed Trailing
SQL_TYP_NNUMERIC_BINARY	-24	Nullable Numeric Signed Leading Separate
SQL_TYP_NNUMERIC_BIN_BE	-25	Nullable Numeric Signed Trailing Separate
SQL_TYP_BINARY	-51	Nullable Numeric Binary (Little Endian)
SQL_TYP_NBINARY	-52	Nullable Numeric Binary (Big Endian)
SQL_TYP_NATDATE	-53	Binary
SQL_TYP_NNATDATE	-54	Nullable BINARY

SQL_TYP_NATTIME	-55	NATURAL DATE
SQL_TYP_NNATTIME	-56	Nullable NATURAL TIME
SQL_TYP_SQLDA	-57	SQLDA
SQL_TYP_NATTIMESTAMP	-58	NATURAL TIMESTAMP
SQL_TYP_NNATTIMESTAMP	-59	Nullable NATURAL TIMESTAMP
SQL_TYP_SQLBIT	-60	BIT
SQL_TYP_NSQLBIT	-61	Nullable BIT

ACEAPI SQL Functions

About ACEAPI SQL Functions

Each SAGQ* SQL function requires an **anchor**. This must be declared as a 4 byte integer global variable. Before calling the first function, **anchor** must be initialized to zero, i.e. int anchor = 0;.

SAGQACOM

```
sagqacom (
        int anchor,
        SQL_CTR_BLOCK *  sql_cb_ptr,
        int autocommit
        );
```

The SAGQACOM function enables or disables autocommit.

autocommit	AUTOCOMMIT_ON (9)	Autocommit is enabled.
	AUTOCOMMIT_OFF (10)	Autocommit is disabled (default).

Autocommit is disabled (default).

For information about anchor (required by each SAGQ* SQL function), see About ACEAPI SQL Functions.

SAGQCLOS

```
sagqclos (
            int anchor,
            SQL_CTR_BLOCK * sql_cb_ptr,
            char * cursor_ptr
            );
```

The SAGQCLOS function closes a cursor identified by *cursor_ptr*.

cursor_ptr	The cursor opened in SAGQOPEN to be closed.

For information about anchor (required by each SAGQ* SQL function), see About ACEAPI SQL Functions.

SAGQCONN

```
sagqconn (
            int anchor,
            SQL_CTR_BLOCK * sql_cb_ptr,
            int statement_type,
            char * server_ptr,
            char * session_ptr,
            char * user_ptr,
            char * password_ptr,
            char * charset_ptr
            );
```

The SAGQCONN function establishes a connection with an SQL Server.

statement_type	CONNECT_USER_STMT (1)	The user-defined connection will be identified by the name given in session_ptr.
	OTHER_STMT (99) or NULL	The connection will be made using the server name supplied in server_ptr (default).
server_ptr	Server name	Default server if NULL.
session_ptr	Session name	User-defined connection specifier used to set different connections. If not specified, the server name will be used as the connection name.
user_ptr	User name	Default user if NULL.
password_ptr	Password	No password if NULL.
charset_ptr	Character set	Default character set if NULL.
	NONULLTERMINATE	All input and output strings will be processed without null termination considerations. Recommended for all non C/C++ applications.

For information about anchor (required by each SAGQ* SQL function), see About ACEAPI SQL Functions.

SAGQDEAL

```
sagqdeal (
        int anchor,
        SQL_CTR_BLOCK * sql_cb_ptr,
        char * stmt_id_ptr
        );
```

The SAGQDEAL function deallocates a previously prepared statement identified by *stmt_id_ptr*.

Dynamic SQLDA structures allocated by SAGQPREP will also be freed if *sql_cb_ptr->input_sqlda_ptr* and/or *sql_cb_ptr->output_sqlda_ptr* -> NULL.

stmt_id_ptr	The statement id returned by SAGQPREP to be deallocated.
sql_cb_ptr->input_sqlda_ptr	**Static SQLDA** NULL **Dynamic SQLDA** SQLDA structure, the result of the SAGQPREP function.
sql_cb_ptr->output_sqlda_ptr	**Static SQLDA** NULL **Dynamic SQLDA only** SQLDA structure, the result of the SAGQPREP function.

For information about anchor (required by each SAGQ* SQL function), see About ACEAPI SQL Functions.

SAGQDECH

sagqdech (
int anchor,
SQL_CTR_BLOCK * sql_cb_ptr,
char * cursor_ptr,
char * stmt_id_ptr
);

The SAGQDECH function declares a cursor **with hold option** for a previously prepared statement identified by *stmt_id_ptr*.

Declaring a cursor is optional (but recommended) and is required only if specific processing on a cursor is performed.

If used, *stmt_id_ptr* in the subsequent open call (SAGQOPEN) must be set to NULL (null pointer). Otherwise, a cursor **without hold option** will be implicitly declared within the open function.

cursor_ptr	The cursor to be declared.
stmt_id_ptr	The statement id returned by SAGQPREP to be declared.

Important: There is no current support for **Declare Cursor With Hold**.

For information about anchor (required by each SAGQ* SQL function), see About ACEAPI SQL Functions.

SAGQDECL

```
sagqdecl (
                int anchor,
                SQL_CTR_BLOCK * sql_cb_ptr,
                char * cursor_ptr,
                char * stmt_id_ptr
                );
```

The SAGQDECL function declares a cursor (**without hold option**) for a previously prepared statement identified by *stmt_id_ptr*.

Declaring a cursor is optional (but recommended) and is required only if specific processing on a cursor is performed.

If used, *stmt_id_ptr* in the subsequent open call (SAGQOPEN) must be set to NULL (null pointer). Otherwise, a cursor (**without hold option)** will be implicitly declared within the open function.

cursor_ptr	The cursor to be declared.
stmt_id_ptr	The statement id returned by SAGQPREP to be declared.

For information about anchor (required by each SAGQ* SQL function), see About ACEAPI SQL Functions.

SAGQDESC

```
sagqdesc (
                int anchor,
                SQL_CTR_BLOCK * sql_cb_ptr,
                char * stmt_id_ptr
                );
```

The SAGQDESC function describes a previously prepared statement identified by *stmt_id_ptr*. The *input_sqlda_ptr* and/or *output_sqlda_ptr* in *sql_cb_ptr* must have been previously allocated.

stmt_id_ptr	The statement id returned by SAGQPREP to be executed.
sql_cb_ptr->input_sqlda_ptr	**Static SQLDA** SQLDA structure allocated by the application. **Dynamic SQLDA** SQLDA structure, the result of the SAGQPREP function.
sql_cb_ptr->output_sqlda_ptr	**Static SQLDA** SQLDA structure allocated by the application. **Dynamic SQLDA** SQLDA structure, the result of the SAGQPREP function.

For information about anchor (required by each SAGQ* SQL function), see About ACEAPI SQL Functions.

SAGQDISC

```
sagqdisc (
          int anchor,
          SQL_CTR_BLOCK * sql_cb_ptr,
          int disconnect_type,
          char * session_ptr
          );
```

The SAGQDISC function terminates one or more connected SQL Server sessions.

session_ptr		If non NULL, the identified session is terminated.
disconnect_type	DISC_ALL_STMT (2)	All connections are terminated.
	DISC_DEFAULT_STMT (3)	The default server connection is terminated.
	DISC_CURRENT_STMT (4)	The current connection is terminated.

For information about anchor (required by each SAGQ* SQL function), see About ACEAPI SQL Functions.

SAGQEXEC

sagqexec (
int anchor,
SQL_CTR_BLOCK * sql_cb_ptr,
char * stmt_id_ptr
);

The SAGQEXEC function executes the previously prepared statement identified by *stmt_id_ptr*.

stmt_id_ptr	The statement id returned by SAGQPREP to be executed.

For information about anchor (required by each SAGQ* SQL function), see About ACEAPI SQL Functions.

SAGQEXIM

```
sagqexim (
                int anchor,
                SQL_CTR_BLOCK * sql_cb_ptr,
                char * stmt_ptr
                );
```

The SAGQEXIM function immediately executes the statement identified by *stmt_ptr*.

stmt_ptr	The statement to be executed immediately.

For information about anchor (required by each SAGQ* SQL function), see About ACEAPI SQL Functions.

SAGQFETC

sagqfetc (
    int anchor,
    SQL_CTR_BLOCK * sql_cb_ptr,
    char * cursor_ptr);

The SAGQFETC function fetches a row from the cursor identified by *cursor_ptr*. The results are stored in the memory indicated by the pointers *output_sqlda_ptr->sql_var[x]-> sqldata*.

cursor_ptr	The cursor opened in SAGQOPEN.
sql_cb_ptr->output_sqlda_ptr->sql_var[x]-> sqldata	The result of the SAGQFETC function.

For information about anchor (required by each SAGQ* SQL function), see About ACEAPI SQL Functions.

SAGQGERR

```
sagqgerr (
        int anchor,
        SQL_CTR_BLOCK * sql_cb_ptr
        );
```

The SAGQGERR function returns an error text for the last sqlcode returned.

sql_cb_ptr->error_text	The error text associated with the last sqlcode returned.

For information about anchor (required by each SAGQ* SQL function), see About ACEAPI SQL Functions.

SAGQGRCI

```
sagqgrci (
                int anchor,
                SQL_CTR_BLOCK * sql_cb_ptr,
                char * rci_version,
                int rci_version_nonullterm);
```

The SAGQGRCI function returns the RCI version string. The sqlcode returned by this function is always 0.

rci_version		The RCI version string, declared as a 256 byte character field : char rci_version [256 + 1];
rci_version_nonullterm	0	RCI version string will be NULL terminated. (Recommended for C/C++ applications).
	1	RCI version string will not be NULL terminated. (Recommended for non C/C++ applications).

For information about anchor (required by each SAGQ* SQL function), see About ACEAPI SQL Functions.

SAGQOPEN

```
sagqopen (
        int anchor,
        SQL_CTR_BLOCK * sql_cb_ptr ,
        char * cursor_ptr,
        char * stmt_id_ptr
        );
```

The SAGQOPEN function opens the cursor identified by *cursor_ptr*.

cursor_ptr	The previously declared cursor to be opened or the implicitly declared cursor name returned by SAQGOPEN itself.
stmt_id_ptr	If supplied, the cursor is implicitly declared and opened for the previously prepared statement. The cursor name will be returned in *cursor_ptr*. If a cursor was previously declared by calling SAGQDECL, *stmt_id_ptr* must be set to NULL (null pointer).

For information about anchor (required by each SAGQ* SQL function), see About ACEAPI SQL Functions.

SAGQPREP

```
sagqprep (
                int anchor,
                SQL_CTR_BLOCK * sql_cb_ptr,
                int statement_type,
                char * stmt_id_ptr,
                char * stmt_ptr
                );
```

The SAGQPREP function prepares and describes the statement given in *stmt_ptr*. The result can be accessed with the statement id *stmt_id_ptr.*

Both static and dynamic SQLDA structures are supported.

Static SQLDA

The input and/or output SQLDA structures of the correct size must be allocated by the application, and be appropriately initialized. The *sqldaid* field in the SQLDA structure may contain any value except the seven byte reserved value **SQLDADA**, which identifies a dynamically allocated SQLDA.

A call to SAGQPREP requires:

sql_cb_ptr->input_sqlda_ptr = API_INPUT_SQLDA ;

sql_cb_ptr->output_sqlda_ptr = API_OUTPUT_SQLDA ;

A static SQLDA cannot be freed or reallocated by ACEAPI; the application must ensure that sufficient storage has been allocated to contain the results for all input/output values returned by the call to SAGQPREP. The *sqln* value must not exceed the number of SQLVAR occurrences provided in the SQLDA, otherwise unpredictable results may occur.

The *sqld* value contains the number of columns in the referenced table or view.

If the *sqln* value is smaller than the number of columns required for the referenced table or view, SAGQPREP will return with SQLCODE = -9985 (Unable to allocate/free memory).

Dynamic SQLDA

Dynamic SQLDA structures are allocated by SAGQPREP and not by the application.

The *sqldaid* field in the SQLDA structure contains the seven byte reserved value **SQLDADA.** Do not change this field; it identifies a dynamically allocated SQLDA. The eighth byte of the field may be modified by the application.

The initial call to SAGQPREP requires :

sql_cb_ptr->input_sqlda_ptr = NULL

sql_cb_ptr->output_sqlda_ptr = NULL

If any of the following pointers are NULL, they will be allocated:

sql_cb_ptr->input_sqlda_ptr

sql_cb_ptr->output_sqlda_ptr

The following pointers are allocated when statement_type = SELECT_STMT:

sql_cb_ptr->output_sqlda_ptr

Once the SQLDA's have been allocated by SAGQPREP, the pointers returned may be used for subsequent calls to SAGQPREP until permanently deallocated by a call to SAGQDEAL.

Dynamic SQLDA structures are always allocated with the correct number of input/output values required by the statement. The *sqln* value indicates the number of SQLVAR occurrences provided in the SQLDA. The *sqld* value will contain the number of columns in the referenced table or view.

Use the latest input/output pointers returned by SAGQPREP for the statement because the previous pointers may have been reallocated and no longer valid.

statement_type	SELECT_STMT (5)	The statement being prepared is 'SELECT'.
	OTHER_STMT (99) or NULL	The statement being prepared is not 'SELECT'.
sql_cb_ptr->input_sqlda_ptr		**Static SQLDA** SQLDA structure allocated by the application. **Dynamic SQLDA** SQLDA structure, the result of the SAGQPREP function.
sql_cb_ptr->output_sqlda_ptr		**Static SQLDA** SQLDA structure allocated by the application. **Dynamic SQLDA** SQLDA structure, the result of the SAGQPREP function.
stmt_id_ptr		The statement id for the statement to be prepared.
stmt_ptr		The statement to be prepared and described.

For information about anchor (required by each SAGQ* SQL function), see About ACEAPI SQL Functions.

SAGQSETA

```
sagqseta (
        int anchor,
        SQL_CTR_BLOCK * sql_cb_ptr,
        char * catalog_ptr
        );
```

The SAGQSETA function sets the default catalog to the catalog identified by *catalog_ptr*.

catalog_ptr	The default catalog name.

For information about anchor (required by each SAGQ* SQL function), see About ACEAPI SQL Functions.

SAGQSETC

```
sagqsetc (
            int anchor,
            SQL_CTR_BLOCK *sql_cb_ptr,
            char * session_ptr
            );
```

The SAGQSETC function sets the active connection to the connection identified by *session_ptr*.

session_ptr	Set or establish the connection to this session. If NULL the current connection is set to the default server.

For information about anchor (required by each SAGQ* SQL function), see About ACEAPI SQL Functions.

SAGQSETS

```
sagqsets (
        int anchor,
        SQL_CTR_BLOCK * sql_cb_ptr,
        char * schema_ptr
        );
```

The SAGQSETS function sets the default schema to the schema identified by *schema_ptr*.

schema_ptr	The default schema name.

For information about anchor (required by each SAGQ* SQL function), see About ACEAPI SQL Functions.

SAGQSETT

```
sagqsett (
        int anchor,
        SQL_CTR_BLOCK * sql_cb_ptr,
        int timeout_value
        );
```

The SAGQSETT function sets the inactivity timeout value.

timeout_value	The inactivity timeout value in minutes.

For information about anchor (required by each SAGQ* SQL function), see About ACEAPI SQL Functions.

Set Timeout is currently unsupported.

ACEAPI Utility Functions

SAGQDECO

```
sagqdeco (
            int sqllen,
            int sqlprecision,
            int sqlscale,
            int sqllength
            );
```

The SAGQDECO function decodes the sqllen of decimal and numeric SQL data type columns into three components: sqlprecision, sqlscale and sqllength. The values for sqlprecision and sqllength are interchangeable.

sqllen	Input	SQLVAR->SQLLEN to be decoded.
sqlprecision	Output	SQLPRECISION
sqlscale	Output	SQLSCALE
sqllength	Output	SQLLENGTH

Examples

C/C++

```
#include <aceapic.h>
sagqdeco (&API_OUTPUT_SQLDA->sqlvar[column].sqllen,
          &API_SQLPRECISION,
          &API_SQLSCALE,
          &API_SQLLENGTH);
```

COBOL

All parameters must be declared as COMP-5.

```
COPY ACEAPIB.
CALL "SAGQDECO" USING
      BY REFERENCE SQLLEN OF SQLVAR OF API-OUTPUT-SQL(column)
      BY REFERENCE API_SQLPRECISION
      BY REFERENCE API_SQLSCALE
      BY REFERENCE API_SQLLENGTH
```

PL/I

```
%INCLUDE aceapip ;
CALL sagqdeco(
     API_OUTPUT_SQLDA_PTR->API_OUTPUT_SQLDA.SQLVAR(column).SQLLEN,
     API_SQLPRECISION,
     API_SQLSCALE,
     API_SQLLENGTH) ;
```

SAGQENCO

```
sagqenco (
            int sqllen,
            int sqlprecision,
            int sqlscale,
            int sqllength
            );
```

The SAGQENCO function encodes the sqlprecision, sqlscale and sqllength into sqllen of decimal and numeric SQL data type columns.

Any value may be supplied for sqllength, only sqlprecision and sqlscale will be used for the encoding.

sqllen	Output	SQLVAR->SQLLEN
sqlprecision	Input	SQLPRECISION to be encoded
sqlscale	Input	SQLSCALE to be encoded
sqllength	Input	SQLLENGTH to be encoded (not used)

Examples

C/C++

```
#include <aceapic.h>
sagqenco (&API_OUTPUT_SQLDA->sqlvar[column].sqllen,
          &API_SQLPRECISION,
          &API_SQLSCALE,
          &API_SQLLENGTH);
```

COBOL

All parameters must be declared as COMP-5.

```
COPY ACEAPIB.
CALL "SAGQENCO" USING
      BY REFERENCE SQLLEN OF SQLVAR OF API-OUTPUT-SQL(column)
      BY REFERENCE API_SQLPRECISION
      BY REFERENCE API_SQLSCALE
      BY REFERENCE API_SQLLENGTH
```

PL/I

```
%INCLUDE aceapip ;
CALL sagqenco(
     API_OUTPUT_SQLDA_PTR->API_OUTPUT_SQLDA.SQLVAR(column).SQLLEN,
     API_SQLPRECISION,
     API_SQLSCALE,
     API_SQLLENGTH) ;
```

ACEAPI Compilation and Linkage examples

The application has to be compiled and linked to the appropriate stub modules for the environment.

Windows

The required static libraries **aceapi.lib** and **rciclnt.lib** are located in the CONNX -> PRECOMPILER -> WINDOWS directory.

C/C++

```
cl example.c /link
C:\CONNX32\PRECOMPILER\WINDOWS\aceapi.lib
C:\CONNX32\PRECOMPILER\WINDOWS\rciclnt.lib
```

Microfocus COBOL / NET Express

```
cbllink -v -o.example.exe example.cbl
```

Microfocus COBOL / NET Express does not require the static libraries to be linked, but the DLL's themselves must be dynamically loaded by the application at execution time, as follows :

```
*   ACEAPI DLL POINTERS (REQUIRED BY MICROFOCUS COBOL)
 01 API-DLL-POINTER.
    02 DLLPTR1                  PROCEDURE-POINTER.
    02 DLLPTR2                  PROCEDURE-POINTER.
    SET DLLPTR1
     TO ENTRY "aceapi"
    SET DLLPTR2
     TO ENTRY "rciclnt"
```

The above code is already declared in the supplied COBOL copybooks, **aceapib.cbl** and **apidll.cbl**.

Open COBOL

```
cobc -fstatic-call -fixed -v -Wall -Wtruncate -x example.cbl
C:\CONNX32\PRECOMPILER\WINDOWS\aceapi.lib
C:\CONNX32\PRECOMPILER\WINDOWS\rciclnt.lib
```

VisualAge PL/I

```
pli   example.pli
(LANGLVL(SPROG) LIMITS(EXTNAME(8)) MACRO NOINSOURCE NOT("\") OR("!")
   SOURCE SNAP)
ilink example.obj
C:\CONNX32\PRECOMPILER\WINDOWS\aceapi.lib
C:\CONNX32\PRECOMPILER\WINDOWS\rciclnt.lib
/OUT:example.exe
```

Open Systems

At execution time, include the CONNX 'embedded' directory in the LD_LIBRARY_PATH.

C/C++

```
gcc -I../DC_Work -Wall -c -o example.o example.c

gcc -o example example.o
-L/home/connx/embedded/LibRCI_32 -laceapi_32 -lrciclnt_32
```

Open COBOL

```
cobc -fstatic-call -fixed -v -Wall -Wtruncate -x example.cbl
-L/home/connx/embedded/LibRCI_32 -laceapi_32 -lrciclnt_32
```

z/OS

On z/OS platforms, link the application to the ACEAPI stub **API3GL** after successful compilation. At execution time, include the CONNX RCI load library in the STEPLIB.

IBM C

```
//  SET USRLOD=CONNX.LOAD
//  SET USROBJ=CONNX.OBJ
//  SET USRSRC=CONNX.SRCE
//*
//* -------------------------
//*        Compile / Link
//* -------------------------
//*
//C        EXEC PROC=EDCCL,
//         CPARM='DEF(BIGFUNC),longname,margins(1,80)',
//         CPARM2='noseq,noshowinc',
//         CPARM3='',
//         LPARM='NOMAP,ALIASES=NO,UPCASE=NO',
//         INFILE=EXAMPLE
//COMPILE.SYSIN  DD DISP=SHR,DSN=&USRSRC(EXAMPLE)
//COMPILE.SYSLIB DD
//             DD
//             DD DISP=SHR,DSN=&USRSRC
//LKED.SYSLMOD   DD DISP=SHR,DSN=&USRLOD
```

```
//LKED.SYSIN     DD *
   INCLUDE USROBJ(EXAMPLE)
   INCLUDE USROBJ(API3GL)
   ENTRY   MAIN
   NAME    EXAMPLE(R)
//LKED.USROBJ    DD DISP=SHR,DSN=&USROBJ
```

IBM COBOL

```
//  SET USRLOD=CONNX.LOAD
//  SET USROBJ=CONNX.OBJ
//  SET USRSRC=CONNX.SRCE
//*
//* --------------------------
//*        Compile / Link
//* --------------------------
//*
//C        EXEC PROC=IGYWCL,
// PARM.COBOL='NOLIST,MAP,XREF(SHORT),RENT,RMODE(ANY),LIB,TEST(SYM)',
// PARM.LKED='LIST,MAP,XREF,ALIASES=NO,UPCASE=NO,MSGLEVEL=4,EDIT=YES'
//*
//COBOL.SYSIN    DD DISP=SHR,DSN=&USRSRC(EXAMPLE)
//COBOL.SYSLIB   DD DISP=SHR,DSN=&USRSRC
//COBOL.SYSLIN   DD DISP=SHR,DSN=&USROBJ(EXAMPLE)
//COBOL.SYSPRINT DD SYSOUT=*
//COBOL.SYSTERM  DD SYSOUT=*
//LKED.SYSLIB    DD
//               DD DISP=SHR,DSN=&USRLOD
//LKED.SYSLIN    DD DISP=SHR,DSN=&USROBJ(EXAMPLE)
//               DD DDNAME=SYSIN
//LKED.SYSLMOD   DD DISP=SHR,DSN=&USRLOD
//LKED.SYSIN     DD *
   MODE    AMODE(31),RMODE(ANY)
   INCLUDE USROBJ(API3GL)
   ENTRY   EMPLOYEE
   NAME    EXAMPLE(R)
//LKED.USROBJ    DD DISP=SHR,DSN=&USROBJ
//LKED.USRLOD    DD DISP=SHR,DSN=&USRLOD
```

IBM PL/I

```
//  SET USRLOD=CONNX.LOAD
//  SET USROBJ=CONNX.OBJ
//  SET USRSRC=CONNX.SRCE
//*
//* -------------------------
//*        Compile / Link
//* -------------------------
//*
//C        EXEC PROC=IBMZCB,
//         LNGPRFX='IBMZ',
//         PARM.PLI=('LANGLVL(SPROG),MACRO,NOINSOURCE,SOURCE,',
//         'LIMITS(EXTNAME(8)),NOT("\"),OR("!")'),
//         PARM.BIND=('LIST,MAP,XREF,ALIASES=NO,UPCASE=NO,',
//         'MSGLEVEL=4,EDIT=YES')
//*
//PLI.SYSIN      DD DISP=SHR,DSN=&USRSRC(EXAMPLE)
//PLI.SYSLIB     DD DISP=SHR,DSN=&USRSRC
//PLI.SYSLIN     DD DISP=SHR,DSN=&USROBJ(EXAMPLE)
//PLI.SYSPRINT   DD SYSOUT=*
//PLI.SYSTERM    DD SYSOUT=*
//BIND.SYSLIB    DD
//               DD DISP=SHR,DSN=&USRLOD
//BIND.SYSLIN    DD DISP=SHR,DSN=&USROBJ(EXAMPLE)
//               DD DDNAME=SYSIN
//BIND.SYSLMOD   DD DISP=SHR,DSN=&USRLOD
//BIND.SYSIN     DD *
  MODE    AMODE(31),RMODE(ANY)
  INCLUDE USROBJ(API3GL)
  ENTRY   CEESTART
  NAME    EXAMPLE(R)
//BIND.USROBJ    DD DISP=SHR,DSN=&USROBJ
//BIND.USRLOD    DD DISP=SHR,DSN=&USRLOD
```

Index