AA-ND79A-TE

VAX Rdb/VMS

Guide to Using SQL/Services

December 1989

This manual describes how to develop application programs using the SQL/Services component of Rdb/VMS Version 3.1. It is intended for programmers who are familiar with the dynamic SQL interface to the VAX Rdb/VMS relational database management system.

Revision/Update Information:	This is a new manual.
Operating System:	VMS
Software Version:	VAX Rdb/VMS Version 3.1

digital equipment corporation
maynard, massachusetts

The information in this document is subject to change without notice and should not be construed as a commitment by Digital Equipment Corporation. Digital Equipment Corporation assumes no responsibility for any errors that may appear in this document.

The software described in this document is furnished under a license and may be used or copied only in accordance with the terms of such license.

No responsibility is assumed for the use or reliability of software on equipment that is not supplied by Digital Equipment Corporation or its affiliated companies.

Restricted Rights: Use, duplication, or disclosure by the U.S. Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.227-7013.

The Reader's Comments forms at the end of this document request your critical evaluation to assist in preparing future documentation.

The following are trademarks of Digital Equipment Corporation:

ACMS
ALL–IN–1
DATATRIEVE
DEC
DEC/CMS
DEC/MMS
DECforms
DECintact
DECmate
DECnet
DECUS
DECwindows
DECwriter
DIBOL

MASSBUS
MicroVAX
PDP
P/OS
Professional
Rainbow
RALLY
Rdb/ELN
Rdb/VMS
ReGIS
RSTS
RSX
RT
TDMS

ULTRIX
UNIBUS
VAX
VAX CDD
VAX FMS
VAXcluster
VAXELN
VAXstation
VIDA
VMS
VT
Work Processor
digital™

MS-DOS is a trademark of Microsoft Corporation.
dBASE is a registered trademark of Ashton-Tate Corporation.
dBASE IV is a trademark of Ashton-Tate Corporation.

This document was prepared using VAX DOCUMENT, Version 1.2

Contents

4 Programming Guidelines

5 Data Types and Environment Variables

6 API Routines

7 Data Structures

A Filter Expression Functions

B SQL/Services Sample Application

C Sample Log Files

Index

Examples

Figures

Tables

Preface

VAX Rdb/VMS, often referred to as Rdb/VMS in this manual, is a general purpose database management system based on the relational data model.

SQL/Services is a client/server component of Rdb/VMS. It allows application programs running on various types of computers to access DIGITAL Standard Relational Interface (DSRI) compliant databases on other computers. For example, an application program running on an MS-DOS personal computer (a client) can access an Rdb/VMS database on a VAX computer (a server). This manual describes how to develop SQL/Services application programs.

Intended Audience

This manual is intended for experienced applications programmers. To use SQL/Services, you should be familiar with:

- The Rdb/VMS SQL interface (an implementation of the industry-standard structured query language)
- A high-level programming language (preferably C) that supports pointer variables

If you are unfamiliar with SQL, it is recommended that you read the VAX Rdb/VMS Guide to Using SQL and the *VAX Rdb/VMS SQL Reference Manual* before attempting to write SQL/Services application programs.

Operating System Information

Information about the operating systems and related software that are compatible with this version of Rdb/VMS is included in the Rdb/VMS media kit.

For information on the compatibility of other software products with this version of Rdb/VMS, refer to the System Support Addendum (SSA) that comes with the Software Product Description (SPD). You can use the SPD/SSA to verify which versions of your operating system are compatible with this version of Rdb/VMS.

Contact your Digital representative if you have questions about the compatibility of other software products with this version of Rdb/VMS.

Structure

This manual has seven chapters and three appendixes.

Chapter 1 Introduces SQL/Services

Chapter 2 Is a condensed discussion of dynamic SQL for those unfamiliar with it

Chapter 3 Is an overview of the routines and data structures that make up SQL/Services

Chapter 4 Provides guidelines for application development, including a detailed description of the sample application

Chapter 5 Is a detailed reference description of the SQL/Services data types and environment variables

Chapter 6 Is a detailed reference description of the SQL/Services API routines

Chapter 7 Is a detailed reference description of the SQL/Services data structures

Appendix A Describes the functions that can be used in filter expressions

Appendix B Contains listings of the sample application

Appendix C Contains listings of the log files produced by the Installation Verification Procedure

SQL/Services error message descriptions and user actions are provided in the file SYS$HELP:SQLSRV$MSG.DOC.

Related Manuals

The following manuals contain information related to SQL/Services.

- *VAX Rdb / VMS Guide to Using SQL*

  Introduces the Rdb/VMS SQL (structured query language) interface, and shows how to retrieve, store, and update data interactively and through application programs.

- *VAX Rdb / VMS SQL Reference Manual*

  Provides reference material and a complete description of the statements, the interactive, dynamic, and module language interfaces, and the syntax for SQL, the structured query language interface for Rdb/VMS.

- *VAX Rdb / VMS Release Notes*

  Describes new features, problems and problems fixed, restrictions, and other information related to the current release of Rdb/VMS. Contains information about SQL and other Rdb/VMS interfaces and utilities.

- *VAX Rdb / VMS Installation Guide*

  Describes how to install Rdb/VMS.

- *VAX Rdb / VMS Introduction and Master Index*

  Introduces Rdb/VMS and explains major terms and concepts. Includes a glossary, a directory of Rdb/VMS documentation, and a master index that combines entries from all the Rdb/VMS manuals.

Conventions

This section explains the conventions used in this manual:

.
.
. — A vertical ellipsis in an example means that information not directly related to the example has been omitted.

Color — In printed manuals, color in examples shows user input.

[ ] — Brackets enclose optional clauses from which you can choose one or none.

$ — The dollar sign represents the DIGITAL Command Language prompt. This symbol indicates that the DCL interpreter is ready for input.

> The right angle bracket represents the MS-DOS command prompt. This symbol indicates that the MS-DOS command language interpreter is ready for input.

% The percent sign represents the ULTRIX shell prompt. This symbol indicates that the ULTRIX shell is ready for input.

e, f, t Index entries in the printed manual may have a lowercase e, f, or t following the page number; the e, f, or t is a reference to the example, figure, or table, respectively, on that page.

References to Products

The SQL/Services documentation to which this document belongs often refers to VAX Rdb/VMS software as Rdb/VMS.

1 Introduction

SQL/Services is a client/server component of Rdb/VMS. It allows application programs running on various types of computers to access DIGITAL Standard Relational Interface (DSRI) compliant databases on other computers, as shown in Figure 1–1. For example, an application program running on an MS-DOS personal computer (a client) can access an Rdb/VMS database on a VAX computer (a server).

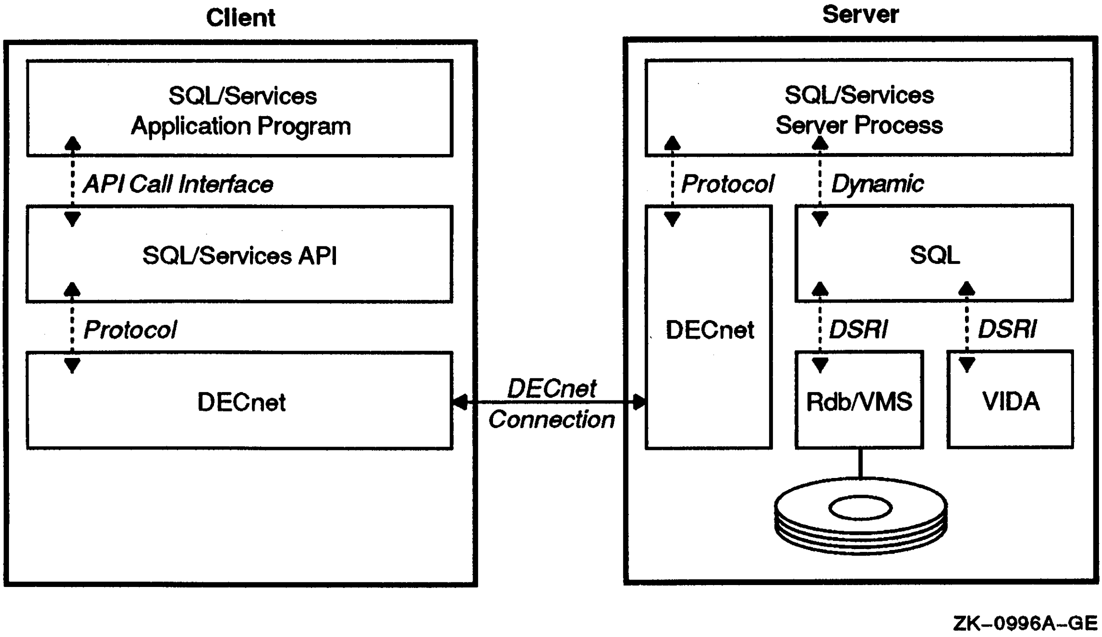
Application programs access SQL/Services through an **Application Programming Interface** (API), which is a set of callable routines that perform functions similar to dynamic SQL. In other words, an SQL/Services application program executes SQL statements at run time. The SQL statements can be embedded in the source code or can be formulated at run time. The SQL statement syntax accepted by SQL/Services is identical to that accepted by dynamic SQL.

The SQL/Services API communicates by means of DECnet with a server process on the VAX system on which the target database resides. The server software is present on all VAX systems running Rdb/VMS Version 3.1 or higher.

The client/server association runs in the context of a user account. Thus, the application program must provide a valid account name and password on the server system.

The client/server association uses a message-based protocol that is virtually transparent to the application program. Other than ensuring that DECnet is installed on both the client and server system and allocating message buffers, you need no knowledge of networking to develop SQL/Services applications.

Figure 1–1 SQL/Services Architecture



2 Dynamic SQL

This chapter provides a condensed discussion of dynamic SQL and discusses the factors to consider when using it. If you are already familiar with dynamic SQL, you may want to skip to Chapter 3, which provides an overview of SQL/Services and how it differs from dynamic SQL.

Dynamic SQL allows application programs to formulate and execute SQL statements at run time. It consists of:

- Statements

  A set of SQL statements with which you can write applications using either the SQL precompiler or the module language processor

- Data Structures

  A set of data structures that provides a way for dynamic SQL and application programs to exchange data and metadata (data about data)

Applications that use dynamic SQL might, for example, translate interactive user input into SQL statements, or open, read, and execute files containing SQL statements. The SQL/Services server is itself a dynamic SQL application.

2.1 Overview of Dynamic SQL Statements

The dynamic SQL statements are summarized in Section 2.1.1 and Section 2.1.2, which group the statements according to function. For each dynamic SQL statement, there is an SQL/Services API routine that performs the same function. (Some API routines combine the functions of two dynamic SQL statements.)

2.1.1 Execution Statements

Execution statements prepare and execute SQL statements and release prepared SQL statement resources.

- PREPARE

  Checks the SQL statement to be dynamically executed for errors and assigns a user-defined name to it. That name is referred to in DESCRIBE, EXECUTE, and DECLARE CURSOR statements.

- DESCRIBE

  Checks a prepared SQL statement for the existence of select list items or parameter markers (as explained in Section 2.2). If either is present, DESCRIBE stores information about it in the SQL Descriptor Area (SQLDA). (Using the SELECT LIST clause of the PREPARE statement is equivalent to using the DESCRIBE statement with the SELECT LIST argument.)

- EXECUTE

  Executes a previously prepared SQL statement other than SELECT.

- EXECUTE IMMEDIATE

  Prepares and executes in one step any SQL statement (other than SELECT) that does not contain parameter markers.

- RELEASE

  Releases all resources used by a prepared SQL statement and prevents the prepared statement from executing again.

Except for the DESCRIBE statement, each of these dynamic SQL statements has an equivalent SQL/Services routine. In SQL/Services, the DESCRIBE and PREPARE statements are combined in a single routine, as shown in Table 2–2.

2.1.2 Result Table Statements

Result table statements allow your program to declare a cursor, open a cursor, fetch data from an open cursor, and close an open cursor.

- DECLARE CURSOR

  Declares a cursor for a prepared SELECT statement.

- OPEN

  Opens a cursor declared for a prepared SELECT statement.

- FETCH

  Retrieves values from a cursor declared for a prepared SELECT statement.

- CLOSE

  Closes a cursor.

Except for the DECLARE CURSOR statement, each of these dynamic SQL statements has an equivalent SQL/Services routine. In SQL/Services, the DECLARE CURSOR and OPEN CURSOR statements are combined in a single routine, as shown in Table 2–2.

2.2 Using Dynamic SQL

In its simplest form, dynamic SQL consists of passing complete SQL statements as string constants or variables to the EXECUTE IMMEDIATE statement. This simple approach may be sufficient for some applications.

However, when you want to dynamically execute the same SQL statement more than once, the EXECUTE IMMEDIATE approach is inefficient because it does not save any context. A more efficient approach is to call the PREPARE statement once, then call the EXECUTE statement as many times as needed. As before, this approach may be sufficient for some applications.

However, to write applications that deal with the entire spectrum of SQL statements, you must also consider the following restrictions:

- *Not all SQL statements can be dynamically executed.* The statements that can be dynamically executed are listed in Table 2–1. Statements that are valid only in interactive SQL cannot be dynamically executed. The statements that are valid in precompiled and module language SQL but cannot be dynamically executed are listed in Table 2–2. Most of the statements in Table 2–2 are statements that make up dynamic SQL itself.
- Dynamically executed SELECT, INSERT, UPDATE, and DELETE statements can contain parameters. The parameters can be constants but they cannot be host variables. *To pass the value of a variable, it must be represented by a parameter marker.*
- *You cannot use parameter markers when using the EXECUTE IMMEDIATE statement;* they are valid only when you are using the PREPARE and EXECUTE statements.
- Because it generates output, *you cannot pass a SELECT statement to the EXECUTE or EXECUTE IMMEDIATE statement.* Instead, you call the PREPARE statement followed by DECLARE CURSOR, OPEN, FETCH, and so forth.

Section 2.2.1 describes how to dynamically execute statements that contain parameter markers. Section 2.2.2 describes how to access the data returned by SELECT statements. Section 2.2.3 describes how to handle statements about which the program has no information.

Table 2–1 SQL Statements That Can Be Dynamically Executed

Statement	Parameter Markers Allowed?	Select List Items?	Associated Dynamic SQL Statements
SELECT	Yes	Yes	PREPARE DESCRIBE (optional) DECLARE CURSOR OPEN FETCH CLOSE RELEASE (optional)

INSERT
UPDATE
DELETE
	Yes	No	PREPARE
			DESCRIBE (optional)
			EXECUTE
			RELEASE (optional)
			EXECUTE IMMEDIATE (if no parameter markers)

CREATE ALTER DROP DECLARE SCHEMA DECLARE TRANSACTION SET TRANSACTION COMMIT ROLLBACK GRANT REVOKE COMMENT ON	No	No	PREPARE EXECUTE RELEASE (optional) EXECUTE IMMEDIATE

Table 2–2 SQL Statements That Cannot Be Dynamically Executed

SQL Statement	Related SQL/Services Routine
BEGIN DECLARE	none
CLOSE	sqlsrv_close_cursor
DECLARE CURSOR	sqlsrv_open_cursor (implicit in)
DECLARE STATEMENT	none
DECLARE TABLE	none
DESCRIBE	sqlsrv_prepare (implicit in)
END DECLARE	none
EXECUTE	sqlsrv_execute

(continued on next page)

Table 2–2 (Cont.) SQL Statements That Cannot Be Dynamically Executed

SQL Statement	Related SQL/Services Routine
EXECUTE IMMEDIATE	sqlsrv_execute_immediate
FETCH	sqlsrv_fetch, sqlsrv_fetch_many
INCLUDE	none
OPEN	sqlsrv_open_cursor
PREPARE	sqlsrv_prepare
RELEASE	sqlsrv_release_statement
SELECT . . . INTO (singleton select)	none
WHENEVER	none

2.2.1 Parameter Markers

Parameter markers represent variables in dynamically executed SQL SELECT, INSERT, UPDATE, and DELETE statements. Question marks (?) embedded in the statement string denote parameters that are to be replaced when the statement is dynamically executed. An example of an SQL statement with parameter markers is:

```
INSERT INTO EMPLOYEES
       (EMPLOYEE_ID, FIRST_NAME, LAST_NAME, CITY)
       VALUES ( ?, ?, ?, ? );
```

The mechanism for mapping parameter markers to variables in application programs is a data structure called the SQLDA (see Section 2.2.4 and Section 7.5). The DESCRIBE statement writes information about parameter markers into an SQLDA structure. Your program examines the SQLDA structure, allocates a data variable for each parameter marker, obtains values for the data variables, and writes the addresses of those variables into the SQLDA, before dynamically executing the SQL statement. Alternatively, your program can initialize the SQLDA itself, instead of calling the DESCRIBE statement.

2.2.2 SELECT Statements

Programs that dynamically execute SELECT statements must declare a cursor to receive the result table and must allocate memory for each select list item in the SELECT statement. After the cursor is opened, FETCH statements return values for rows of the result table.

As with parameter markers, the mechanism for mapping select list items to host variables is a data structure called the SQLDA (see Section 2.2.4 and Section 7.5). The DESCRIBE and PREPARE statements both write select list information into the SQLDA.

If the SELECT statement contains parameter markers, the program must also set up host variables for the parameter markers and assign values to them.

2.2.3 Unknown Statements

It is possible to dynamically execute SQL statements about which the program has no prior information. Such unknown statements may contain parameter markers or select list items (or both). The program can use the PREPARE and DESCRIBE statements to obtain two separate SQLDA structures containing information about the numbers and data types of select list items and parameter markers. Then the program allocates data variables as appropriate and writes the addresses of those variables into the SQLDA structures before executing the unknown statement.

2.2.4 The SQL Descriptor Area

SQL provides a data structure called the **SQL Descriptor Area** (SQLDA) that provides a means for programs to communicate with SQL about parameter markers and select list items. To use the SQLDA, host languages must support pointer variables that provide indirect access to memory by storing the address of data instead of directly storing data in the variable. Declarations for the SQLDA structure in various languages can be found in include files that are provided with SQL.

When SQL processes a DESCRIBE statement, it writes information about select list items (for a DESCRIBE . . . SELECT LIST statement) or parameter markers (for a DESCRIBE . . . MARKERS statement) of a prepared statement into an SQLDA.

The host language program examines the SQLDA to determine how many select list items (DESCRIBE . . . SELECT LIST) or parameter markers (DESCRIBE . . . MARKERS) are present and the data type of each. The program must provide memory (static or dynamic) for each parameter marker or select list item, and write the address of each memory location into the SQLDA.

For parameter markers, the program writes values into the SQLDA before dynamically executing the SQL statement. For select list items, the program reads the data written into the SQLDA by subsequent FETCH statements.

Section 7.5 describes the SQLDA in detail. In addition, the *VAX Rdb/VMS SQL Reference Manual* contains an appendix on the SQLDA and a section on the DESCRIBE statement that discusses the MARKERS and SELECT LIST clauses of the DESCRIBE statement in more detail.

2.2.5 The SQL Communications Area

The **SQL Communications Area** (SQLCA) is a collection of parameters that SQL uses to provide information about the execution of SQL statements to application programs. SQL updates the contents of the SQLCA after completion of every executable SQL statement. The only fields of interest in the SQLCA are the SQLCODE field and the third element of the SQLERRD array.

The SQLCODE field shows whether a statement was successful, and for some errors, the particular error when a statement is not successful.

SQL puts a value in the third element of the SQLERRD array after successful execution of the following statements:

- INSERT: the number of rows stored by the statement
- UPDATE: the number of rows modified by the statement
- DELETE: the number of rows deleted by the statement
- FETCH: the number of the row on which the cursor is currently positioned
- OPEN: zero
- SELECT: the number of rows in the result table formed by the SELECT statement (Note: SQLERRD is not updated for dynamic SELECT statements)

Otherwise, the value of SQLERRD is undefined.

Section 7.3 describes the SQLCA in detail. In addition, the *VAX Rdb/VMS SQL Reference Manual* contains an appendix on the SQLCA.

3

Overview of Routines and Data Structures

This chapter provides overviews of the SQL/Services routines and data structures.

3.1 Overview of API Routines

The SQL/Services Application Programming Interface (API) is a set of callable routines that the client uses to access SQL/Services functions. The API routines are grouped according to function and summarized in Section 3.1.1 through Section 3.1.4.

3.1.1 Association Routines

Association routines create and terminate client/server associations and control the association environment (context).

- sqlsrv_associate

  Creates a client/server association. Makes the remote connection to the server process and negotiates association values. For more information, see Section 6.4.

- sqlsrv_release

  Terminates a client/server association in an orderly fashion. Sends a message to the server requesting termination of the association, disconnects the network link, and releases all client resources related to the association. For more information, see Section 6.14.

- sqlsrv_abort

  Terminates a client/server association immediately. Disconnects from the server and releases all client resources related to the association. For more information, see Section 6.2.

- sqlsrv_set_environment

  Sets new values for environment variables on the server. Environment variables control date, time, and numeric output formats, and string-matching modes. For more information, see Section 6.16.

- sqlsrv_get_environment

  Gets current values of environment variables. For more information, see Section 6.11.

3.1.2 SQL Statement Routines

SQL statement routines prepare and execute SQL statements, and release prepared SQL statement resources. These routines map directly to the dynamic SQL interface.

- sqlsrv_prepare

  Prepares (compiles) a dynamic SQL statement. It returns a statement identifier and SQLDA metadata information (fields that describe parameter markers and select list items). This routine maps to the dynamic SQL PREPARE and DESCRIBE statements. For more information, see Section 6.13.

- sqlsrv_execute

  Executes a prepared SQL statement. This routine maps to the dynamic SQL EXECUTE statement. For more information, see Section 6.6.

- sqlsrv_execute_immediate

  Prepares and executes an SQL statement. This routine cannot be used if the SQL statement contains parameter markers. This routine maps to the dynamic SQL EXECUTE IMMEDIATE statement. For more information, see Section 6.7.

- sqlsrv_release_statement

  Releases client and server statement resources associated with a prepared statement. This routine maps to the dynamic SQL RELEASE statement. For more information, see Section 6.15.

3.1.3 Result Table Routines

Result table routines allow the caller to fetch data from the server by providing calls to open a cursor, associate a filter expression with a cursor, fetch from an open cursor, and close an open cursor.

- sqlsrv_open_cursor

  Opens a cursor by associating a cursor name with a prepared statement identifier. The cursor name is used in each reference to the cursor. An SQL DECLARE CURSOR statement is implicit within the sqlsrv_open_cursor call. For more information, see Section 6.12.

- sqlsrv_set_filter

  Associates a Boolean expression with a cursor to filter out unwanted rows from the result table before they are sent to the client. For more information, see Section 6.17.

- sqlsrv_fetch

  Fetches one row of data from an open cursor. Can be used to fetch rows of information from within an sqlsrv_fetch_many context. For more information, see Section 6.8.

- sqlsrv_fetch_many

  Requests that multiple rows of data be fetched and transmitted to the client in one message. For more information, see Section 6.9.

- sqlsrv_close_cursor

  Closes an open cursor. For more information, see Section 6.5.

3.1.4 Utility Routines

Utility routines provide local services to the caller.

- sqlsrv_allocate_sqlda_data

  Allocates memory for the SQLDA data buffer and indicator variable fields. For more information, see Section 6.3.

- sqlsrv_free_sqlda_data

  Frees memory for the SQLDA data buffer and indicator variable fields. For more information, see Section 6.10.

3.2 Overview of Data Structures

The API routines use the following data structures.

- ASSOCIATE_STR

  This structure is passed as a parameter to sqlsrv_associate to enable or disable various API functions. The sqlsrv_associate routine opens the communications link between client and server and creates an association context. For more information, see Section 7.2.

- SQLDA

  The SQLDA (SQL Descriptor Area) is used to exchange database metadata and data for parameter markers (input) and select lists (output). Parameter markers are required when the SQL statement refers to data not defined at compile time. The SQL/Services SQLDA is identical to that used by dynamic SQL. For more information, see Section 2.2.4 and Section 7.5.

- SQLCA

  The SQLCA (SQL Communications Area) is used to store error messages and SQL statement information returned by SQL/Services. When an API routine returns a non-zero value, the SQLCA contains additional error information. For more information, see Section 7.3.

- SQLSRV_ENV_STR

  This structure provides a mechanism for requesting and receiving environment variable values. An array of these structures is passed to the API with one element for each environment variable. For more information, see Section 7.7.

4
Programming Guidelines

This chapter describes how to develop application programs using SQL/Services.

4.1 Building SQL/Services Application Programs

The process of building SQL/Services application programs consists of these steps:

1 Compile your code using the following `#include` compiler directives:

```
#include <sqlsrvda.h>    /* SQLDA */
#include <sqlsrvca.h>    /* SQLCA */
#include <sqlsrv.h>      /* other structures */
```

On most operating systems, include files are kept in a standard location, indicated in C by placing angle brackets around the name of the file. If these directives do not work on your system, ask the person who installed the SQL/Services API where the include files are located.

2 Link your object module with the SQL/Services API. Linking procedures are system dependent and are thus discussed in separate sections.

4.1.1 Building Applications on the VMS Operating System

The VMS include files are installed in SYS$LIBRARY. Their names are SQLSRVCA.H, SQLSRVDA.H, and SQLSRV.H.

To link your program, enter the command:

```
$ LINK object.OBJ,SYS$LIBRARY:options_file/OPT
```

Replace *object* with the name of your object module and *options_file* with either SQLSRV$API (D_float) or SQLSRV$APIG (G_float) depending on how you compiled your source code. See the *Introduction to VMS System Routines* for more information about VMS data types.

4.1.2 Building Applications on the MS-DOS Operating System

The MS-DOS include files are installed in a directory created by the installer; for example, C:\SQLSRV. Their names are SQLSRVCA.H, SQLSRVDA.H, and SQLSRV.H.

To link your program, enter the command:

```
> LINK object,/STACK=n,,apilib+decnetlib+libc/NOD/NOE
```

Replace *object* with the name of your object module, *n* with the desired stack size (1000 bytes plus whatever is required by your application), *apilib* with one of the libraries shown in Table 4–1, *decnetlib* with the name of the DECnet-DOS Programming Interface Library, and *libc* with the name of the C run-time support library.

Table 4–1 MS-DOS API Libraries

Library	Memory Model
SQSAPIL.LIB	large
SQSAPIM.LIB	medium
SQSAPIS.LIB	small

Note *The DECnet-DOS V2.1 Programming Interface Library contains a reference to the undefined symbol* ***dnet_ask_for_password****. Ignore any linker error messages about this symbol.*

You may find it useful to examine the procedures that build the MS-DOS API Installation Verification Procedure (SQSIVP.BAT and SQSIVP.MAK) and the sample application SQLSRV$DYNAMIC (see Section 4.2.2).

4.1.3 Building Applications on the ULTRIX Operating System

The ULTRIX include files are installed in /usr/include or (if the installer did not have superuser privileges) in a directory created by the installer. Their names are sqlsrvca.h, sqlsrvda.h, and sqlsrv.h.

By default, the ULTRIX C compiler compiles and links your program in one command:

```
% cc file sqsapi.a -o name
% chmod +x name
```

Replace *file* with the name of your source file and *name* with the name you wish for the executable file.

You may find it useful to examine the *make* file that builds the ULTRIX API Installation Verification Procedure (sqsivpu.mak) and the *make* file that builds the sample application SQLSRV$DYNAMIC (see Section 4.2.3).

4.2 Sample Application: SQLSRV$DYNAMIC

This section describes a sample program written in C that illustrates a general type of SQL/Services application. The sample, SQLSRV$DYNAMIC, was derived from SQL$DYNAMIC, the dynamic SQL sample program in the *VAX Rdb/VMS Guide to Using SQL*, which is written in Ada and uses the SQL module processor. The conversion involved recoding in portable C and converting the SQL module language procedures to SQL/Services API routine calls. Complete source listings are provided in Example B–1 and Example B–2.

SQLSRV$DYNAMIC creates an association, accepts SQL statements from the terminal, and executes them by calling routines in the SQL/Services API. In other words, the program resembles in some respects a portable implementation of interactive SQL.

Like interactive SQL, SQLSRV$DYNAMIC recognizes the semicolon (;) as a line terminator and thus accepts multiple-line statements. Input lines beginning with an exclamation point (!) are considered comments and are not executed.

For input statements that contain parameter markers, the program describes the data required and prompts for user input. For SELECT statements, the program creates a cursor, and fetches and displays each row in the result table.

The source code for SQLSRV$DYNAMIC is included with the SQL/Services distribution so you can compile, link, and run it on your own system.

4.2.1 Building the Sample Application on the VMS Operating System

The source code for SQLSRV$DYNAMIC is available on line in the directory SYS$EXAMPLES. To compile, link, and run SQLSRV$DYNAMIC, enter the following commands:

```
$ cc sys$examples:sqlsrv$driver,sys$examples:sqlsrv$dynamic
$ link/exe=sqlsrv$dynamic sqlsrv$driver,sqlsrv$dynamic -
_$ sys$library:sqlsrv$api/opt
$ run sqlsrv$dynamic
```

4.2.2 Building the Sample Application on the MS-DOS Operating System

The source code for SQLSRV$DYNAMIC is available on line in the directory in which the MS-DOS API was installed. If you have the *MAKE* utility on your system, enter the following command:

```
> CD C:\SQLSRV
> MAKE SQSDYN.MAK
> SQSDYN
```

Otherwise, to compile and link the sample application, follow the instructions in Section 4.1.2. The names of the source files are SQSDRV.C and SQSDYN.C.

4.2.3 Building the Sample Application on the ULTRIX Operating System

The source code for SQLSRV$DYNAMIC is available on line. To compile, link, and run SQLSRV$DYNAMIC, enter the following command:

```
% cp /usr/sqlsrv/* .
% make -f sqsdynu.mak
% sqsdynu
```

Replace /usr/sqlsrv with the name of the directory in which the ULTRIX API was installed. The names of the sample application source files are sqsdrvu.c and sqsdynu.c.

4.2.4 Running the Sample Application

When SQLSRV$DYNAMIC starts up, it prompts for the information required to create an association with (establish a DECnet connection with the server process on) a remote system. When the association is made, the program prints instructions and prompts for SQL statements to execute. For example, on the VMS operating system:

```
$ run sqlsrv$dynamic
VMS server node: MYNODE
VMS server account name: MYNAME
VMS server account password: MYPASSWORD

Enter any dynamically executable SQL statement,
continuing it on successive lines.
Terminate the statement with a semicolon.
Built-in commands are: [no]echo and exit.

SQL> DECLARE SCHEMA FILENAME SQL_PERSONNEL;
SQL> SELECT * FROM EMPLOYEES WHERE FIRST_NAME = ?;
Enter value for:   FIRST_NAME
Maximum length is: 11
DATA> Norman
```

```
------ BEGIN RESULT TABLE ------
EMPLOYEE_ID          : 00168
LAST_NAME            : Nash
FIRST_NAME           : Norman
MIDDLE_INITIAL       :
ADDRESS_DATA_1       : 87 West Rd.
ADDRESS_DATA_2       :
CITY                 : Meadows
STATE                : NH
POSTAL_CODE          : 03587
SEX                  : M
BIRTHDAY             : 1932102300000000
STATUS_CODE          : 1
---------- END OF ROW ----------
    .
    .
    .
---------- END OF ROW ----------
EMPLOYEE_ID          : 00245
LAST_NAME            : Roberts
FIRST_NAME           : Norman
MIDDLE_INITIAL       : U
ADDRESS_DATA_1       : 162 Tenby Dr.
ADDRESS_DATA_2       :
CITY                 : Chocorua
STATE                : NH
POSTAL_CODE          : 03817
SEX                  : M
BIRTHDAY             : 1949061100000000
STATUS_CODE          : 1
---------- END OF ROW ----------
------- END RESULT TABLE -------
SQL> EXIT;
$
```

4.2.5 Sample Program Structure

The sample application SQLSRV$DYNAMIC consists of the following modules:

- The SQLSRV$DRIVER module accepts a string from the user (ostensibly containing a dynamic SQL statement) and passes it to the SQLSRV$DYNAMIC module.
- The SQLSRV$DYNAMIC module processes the statement, executing non-SELECT statements and displaying result tables from SELECT statements on the terminal.

4.2.6 The Driver Module

When a user runs SQLSRV$DYNAMIC, it executes the main function in the SQLSRV$DRIVER.C module, which does the following:

- Calls a routine to create an association. Although SQLSRV$DRIVER creates only one association, SQL/Services allows an application to have several associations active at any given time.
- Enters a loop that inputs dynamic SQL statements and passes them to the function execute_statement for processing.
- Calls a routine to close the association.

The implementation of the terminal input/output in SQLSRV$DRIVER is unimportant. The module is intended to be easily replaced. It does, however, demonstrate how to declare the variables that are "global" to a client/server association:

```
char            *assoc_id;
struct SQLCA    sqlca_str;
char            long_error[512];
```

- The variable assoc_id identifies (provides a **handle** for) an active client/server association. Every SQL/Services API routine has an association identifier in its parameter list.

  Assoc_id is declared as a pointer to a character object. The choice of char as the data type is arbitrary because SQLSRV$DYNAMIC does not allocate the object that assoc_id points to, nor does it ever directly access that object. When SQLSRV$DYNAMIC calls the sqlsrv_associate routine, it passes the address of assoc_id (a pointer to a pointer). The API allocates the object and writes its address into assoc_id.
- The variable sqlca_str is real memory that is used as the communications area for an active client/server association. It is declared as an instance of the structure SQLCA, which is defined in the include file SQLSRVCA.H. When SQLSRV$DYNAMIC calls the sqlsrv_associate routine, it passes the address of the SQLCA structure. Then, whenever an API routine call returns a status value other than SQL_SUCCESS, the application can examine the SQLCA structure for error information. In addition, SQL/Services uses the SQLCA to return various types of status information, as described in Section 7.3.
- The variable long_error is real memory that is used as an alternative error message text buffer. The SQLCA field that is intended for error message text is only 70 bytes, which is too short for some error messages. Long_error is 512 bytes, which is sufficient for all possible messages. For more information, see Section 4.2.8.9 and Section 7.2.

4.2.7 Creating and Releasing an Association

The module SQLSRV$DYNAMIC contains a function named create_association that does the following:

- Declares the variables required for an association, including the message protocol buffers and sizes.
- Gets the node name, user name, and password for the server system from the argument vector; if any of these are missing, the create_association function prompts the user.
- Sets up the sizes (in bytes) of the read and write message protocol buffers.

  ```
  read_size = 1024;   /* protocol buffer size value */
  write_size = 1024;  /* protocol buffer size value */
  ```

  Buffer size is a tradeoff between message throughput, memory usage, and maximum number of possible simultaneous associations. Larger buffers result in fewer messages that must be transmitted between client and server when you use the sqlsrv_fetch_many routine to fetch multiple rows (see Section 4.3.2) or the sqlsrv_execute routine to send multiple rows (see Section 4.3.1). You may have to fine tune the buffer sizes to optimize your application for a specific platform.
- Sets up the association structure. This structure is described in detail in Section 7.2.

  ```
  associate_str.CLIENT_LOG = 0;             /* disable client logging.      */
  associate_str.SERVER_LOG = 0;             /* disable server logging.      */
  associate_str.LOCAL_FLAG = 0;             /* this is a remote session.    */
  associate_str.MEMORY_ROUTINE = NULL;      /* use default alloc routine.   */
  associate_str.FREE_MEMORY_ROUTINE = NULL; /* use default free routine.    */
  associate_str.ERRBUFLEN = 512;
  associate_str.ERRBUF = long_error;        /* use alternative error string */
  ```
- Calls the API routine sqlsrv_associate to create the association.

4.2.7.1 Passing the Association Identification Variable

If you are an experienced C programmer and are familiar with multiple levels of indirection, you may prefer to skip this section and go to Section 4.2.8.

The sqlsrv_associate routine is one of two API routines (the other is sqlsrv_prepare) that require addresses to be passed by reference. In other words, one of the arguments (assoc_id) is the address of an address, as in the following example.

```
create_association() {
        .
        .
        .
    char *assoc_id;             /* pointer variable internal to function */
        .
        .
        .
    status = sqlsrv_associate(  /* API routine call */
                     .
                     .
                     .
                &assoc_id);     /* address of pointer variable */
}
```

When the association identifier is declared in the *calling* function (as in SQLSRV$DYNAMIC), make sure not to add an extra level of indirection. In the following example, assoc_id is declared in the main program and passed as a parameter to a function that calls the sqlsrv_associate routine:

```
main () {
        .
        .
        .
    char    *assoc_id;             /* pointer variable */
        .
        .
        .
    create_association(&assoc_id); /* call with address of pointer */
}
```

The function that calls the sqlsrv_associate routine is as follows:

```
create_association(assoc_id)          /* function declaration */
char **assoc_id;                      /* formal parameter */
{
    status = sqlsrv_associate(        /* API routine call */
                     .
                     .
                     .
                assoc_id);  /* argument contains address of pointer */
/*    wrong--> &assoc_id);  would add an extra level of indirection */
}
```

For clarity, the formal association id parameter is defined as a pointer to a pointer. A long integer would work as well because the parameter is an address.

4.2.8 Processing the Dynamic SQL Statement

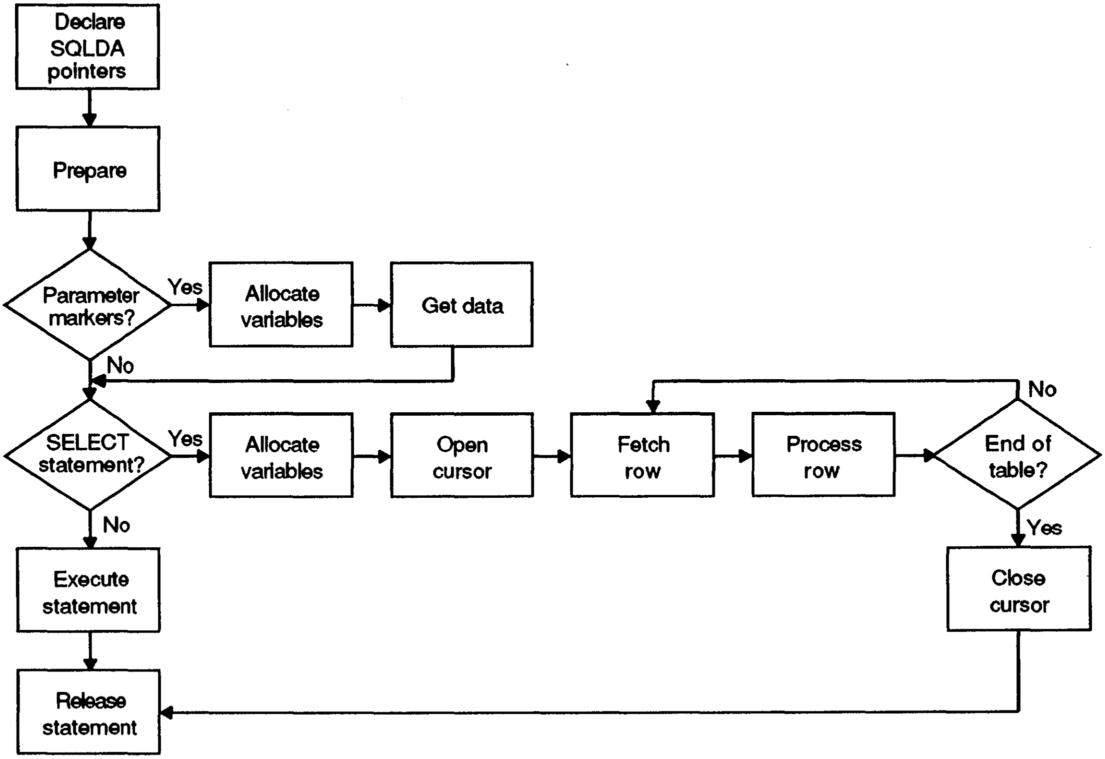
The module SQLSRV$DYNAMIC contains a function named execute_statement that processes the statement string passed to it by the driver module. As shown in Figure 4–1, the execute_statement function does the following:

- Declares SQLDA pointers and other variables.
- Calls the sqlsrv_prepare routine, which prepares (compiles) the statement and returns a statement identification variable.

- Tests the SQLDA pointers to determine whether the statement contains parameter markers or is a SELECT statement.
- If the statement string contains parameter markers, allocates data and indicator variables for the parameter marker SQLDA and calls the get_params function to get data values from the user.
- Calls the sqlsrv_execute routine to execute the statement, unless the statement is a SELECT. In that case, SQLSRV$DYNAMIC:
  - Allocates data and indicator variables for the select list SQLDA
  - Opens a cursor
  - Fetches and displays the rows in the result table
  - Closes the cursor
- Releases the prepared statement.

Section 4.2.8.1 through Section 4.2.8.9 explain the workings of the execute_ statement and get_params functions in more detail.

Figure 4–1 Statement Execution Flow



ZK–0998A–GE

4.2.8.1 Declaring and Allocating SQLDA Structures The SQLDA structure contains SQL parameter marker and select list metadata as well as pointers to data and indicator variables. Thus, the SQLDA is the means by which your application and the SQL/Services API communicate about the SQL statement being prepared for execution.

SQL/Services applications must allocate variables that point to SQLDA structures. The execute_statement function contains the following declarations:

```
struct SQLDA    *param_sqlda;
struct SQLDA    *select_sqlda;
```

The include file SQLSRVDA.H defines the SQLDA structure as follows:

```
/*
 * SQLDA: SQL Description Area data structure.
 */
struct SQLDA {
    char        SQLDAID[8];
    long int    SQLDABC;
    short int   SQLN;       /* Total # of occurrences in SQLVAR */
    short int SQLD;         /* # of select list items or parameter
                             * markers in prepared statement    */
    struct SQLVAR SQLVARARY[1];/* Variable length SQLVARARY.    */
};
```

Your application can either allocate its own SQLDA structures or request SQL/Services to dynamically allocate them. Existing applications written for the Rdb/VMS SQL interface or other ANSI dynamic SQL implementations may use preallocated SQLDA structures. In new SQL/Services applications, however, you may find that the dynamic allocation approach has two major advantages in terms of efficient memory usage:

- One field in the SQLDA, the SQLVARARY, is an array of SQLVAR structures, each of which contains metadata about one parameter marker or one select list item.

  ```
  /*
   * SQLVAR: Variable portion of the SQLDA structure.
   */
  struct SQLVAR {
          short int   SQLTYPE;    /* SQL data type.                  */
          short int   SQLLEN;     /* SQL data length.                */
          char        *SQLDATA;   /* ptr: SQL data.                  */
          short int   *SQLIND;    /* ptr: SQL indicator var.         */
          short int   SQLNAME_LEN;/* length of SQL name.             */
          char        SQLNAME[30];/* SQL name.                       */
  };
  ```

  The length of the SQLVARARY array can vary because it is impossible to predict exactly how many parameter markers or select list items will be present in any given SQL statement. If the API allocates an SQLDA structure, the SQLVARARY can be the exact size needed for any particular statement. If you choose to allocate your own SQLDA structures, you must make sure that the SQLVARARY is large enough for all of the parameter markers or select list items that can be present in a statement.

- By calling the sqlsrv_release_statement or sqlsrv_release routine, you can request the API to deallocate the structures when they are no longer needed. However, the API cannot deallocate structures that it did not allocate.

4.2.8.2 Testing for Parameter Markers

When your application calls the sqlsrv_prepare routine, it passes two SQLDA pointer variables. The sqlsrv_prepare routine is one of two API routines (sqlsrv_associate is the other, as described in Section 4.2.7.1) that require addresses to be passed by reference. In other words, an argument is the address of an address.

```
select_sqlda = NULL;
param_sqlda = NULL;

sts = sqlsrv_prepare(
                    assoc_id,           /* association handle.          */
                    database_id,        /* database_id, must be zero.   */
                    sql_statement,      /* SQL statement.               */
                    &statement_id,      /* Prepared statement id.       */
                    &param_sqlda,
                    &select_sqlda);
```

The param_sqlda pointer can be NULL or can contain the address of a valid SQLDA structure. If you supply a NULL pointer (as in SQLSRV$DYNAMIC) and the SQL statement contains parameter markers, the API dynamically allocates a parameter marker SQLDA and writes the address of the structure into the param_sqlda pointer. In other words, the API allocates the parameter marker SQLDA structure only when the structure is needed. Thus, your application can test the pointer and branch based on the presence or absence of the structure.

```
if (param_sqlda) {
    .
    .
    .
}
```

If you supply a param_sqlda pointer containing the address of a valid SQLDA structure, the API uses that structure to store parameter marker metadata. Applications using preallocated SQLDA structures can branch on the value that the API writes into the SQLD field, which is the number of parameter markers in the SQL statement:

```
if (param_sqlda.SQLD > 0) {
    .
    .
    .
}
```

A nonzero value in the SQLD field indicates the presence of parameter markers.

4.2.8.3 Allocating Indicator and Data Variables If parameter markers are present in the SQL statement, the prepare_statement function calls the API routine sqlsrv_allocate_sqlda_data (which also can be used with select list SQLDAs). If you prefer, your application can allocate and deallocate its own data and indicator variables.

```
sts = sqlsrv_allocate_sqlda_data(assoc_id, param_sqlda);
```

This routine dynamically allocates a data variable of the appropriate type and an indicator variable for each parameter marker and writes the addresses of those variables into the SQLVAR. The length of each data variable matches the SQLVAR.SQLLEN field.

A symmetric routine, sqlsrv_free_sqlda_data, deallocates the variables; however, the API cannot deallocate variables that it did not allocate.

4.2.8.4 Processing Parameter Markers The SQLSRV$DYNAMIC module includes a function named get_params that obtains values for parameter markers. As in the SQLSRV$DRIVER module, the implementation of the terminal input/output is unimportant. As demonstrated in the get_params function, your application must perform the following steps:

1 Allocate data and indicator variables for the parameter markers, as described in Section 4.2.8.3.

```
sts = sqlsrv_allocate_sqlda_data(assoc_id, param_sqlda);
```

2 Execute a loop that iterates once for each parameter marker in the SQL statement. The API places that number in the SQLD field when it executes the sqlsrv_prepare routine.

```
for (i = 0; i < param_sqlda->SQLD; i++) {
      .
      .
      .
} /* for */
```

3 Within the loop, set up a dispatch table based on the data type of the column.

```
switch(param_sqlda->SQLVARARY[i].SQLTYPE) {
    case SQLSRV_ASCII_STRING:
    case SQLSRV_GENERALIZED_NUMBER:
    case SQLSRV_GENERALIZED_DATE:
             .
             .
             .
        gets(param_sqlda->SQLVARARY[i].SQLDATA);
             .
             .
             .
        break;
    case SQLSRV_VARCHAR:    /* counted string */
            .
            .
            .
        break;
} /* switch */
```

For null-terminated ASCII strings (data types other than SQLSRV_VARCHAR), access the SQLDATA field of the appropriate SQLVAR element using the loop counter as an index into the SQLVARARY. Because it uses terminal input/output to obtain data, the get_params function calls the library routine *gets* to write directly into the data variable.

4 For counted strings (SQLSRV_VARCHAR), which are typically used to store binary data, your application must:

a Write a signed word integer into the first word of the SQLDATA field of the appropriate SQLVAR element. That integer represents the number of 8-bit bytes of data to follow. If you are programming in C, you can use a cast operator to coerce the data variable into an integer so that you can write into the first word.

```
char *p;
     .
     .
     .
p = param_sqlda->SQLVARARY[i].SQLDATA;
*(short int *)p = len;
```

b Copy the data into the second and subsequent words of the SQLDATA field of the appropriate SQLVAR element. If you are programming in C, you can use a char pointer to write individual bytes of data into the variable. Use the *sizeof* operator to set the pointer to the first data byte.

```
p += sizeof(short int);
strncpy(p,s,len);
```

Because the get_params function uses terminal input/output to obtain data, it demonstrates the SQLSRV_VARCHAR type by calling the library routine *strncpy* to copy in ASCII data.

4.2.8.5 Executing Non-SELECT Statements For non-SELECT statements, the execute_statement function calls the API routine sqlsrv_execute.

```
sts = sqlsrv_execute(
            assoc_id,               /* association handle.          */
            database_id,            /* database_id, must be zero.   */
            statement_id,           /* Prepared statement id.       */
            execute_flag,           /* Execute mode.                */
            param_sqlda             /* Parameter marker SQLDA.      */
            );
```

4.2.8.6 Testing for SELECT Statements The test for the presence of a SELECT statement is the same as that for parameter markers. When your application calls the sqlsrv_prepare routine, it passes two SQLDA pointer variables.

```
select_sqlda = NULL;
param_sqlda = NULL;

sts = sqlsrv_prepare(
                    assoc_id,           /* association handle.          */
                    database_id,        /* database_id, must be zero.   */
                    sql_statement,      /* SQL statement.               */
                    &statement_id,      /* Prepared statement id.       */
                    &param_sqlda,
                    &select_sqlda);
```

The select_sqlda pointer can be NULL or can contain the address of a valid SQLDA structure. If you supply a NULL pointer (as in SQLSRV$DYNAMIC) and the SQL statement is a SELECT, the API dynamically allocates a select list SQLDA and writes the address of the structure into the select_sqlda pointer. In other words, the API allocates the select list SQLDA structure only when the structure is needed. Thus, your application can test the pointer and branch based on the presence or absence of the structure.

```
if (select_sqlda) {
     .
     .
     .
}
```

If you supply a select_sqlda pointer containing the address of a valid SQLDA structure, the API uses that structure to store select list metadata. Applications using preallocated SQLDA structures can branch on the value that the API writes into the SQLD field, which is the number of select list items in the SQL statement.

```
if (select_sqlda.SQLD > 0) {
     .
     .
     .
}
```

A nonzero value in the SQLD field indicates the presence of select list items.

4.2.8.7 Processing a Result Table

If the SQL statement is a SELECT statement, the execute_statement function emulates interactive SQL by printing out each row in the result table. The steps are:

1 Allocate data and indicator variables for the select list items, as described in Section 4.2.8.3.

```
sts = sqlsrv_allocate_sqlda_data(assoc_id, select_sqlda);
```

2 Open a cursor.

```
sts = sqlsrv_open_cursor(
                  assoc_id,        /* association id                    */
                  cursor_name,     /* handle for cursor                 */
                  statement_id,    /* handle for SELECT statement       */
                  param_sqlda      /* parameter marker SQLDA            */
                  );
```

3 Execute a loop that iterates at least once and stops when the sqlsrv_fetch routine returns a status code indicating that the end of the result table has been reached.

```
do {
    sts = sqlsrv_fetch(
            assoc_id,        /* association id      */
            cursor_name,     /* handle for cursor   */
            0,               /* direction           */
            0L,              /* row number          */
            select_sqlda     /* select list SQLDA   */
            );
    .
    .
    .
} while (sts != SQL_EOS);
```

4 Within the loop, set up a dispatch table based on the status code.

```
switch (sts) {
    case SQL_SUCCESS:
        /* process the data */
          .
          .
          .
        break;
    case SQL_EOS:
        printf("------- END RESULT TABLE -------\n");
        break;
    default:
        return report_error(assoc_id, sqlca_str, long_error);
        break;
} /* switch */
```

5 When sqlsrv_fetch returns a status code of SQL_SUCCESS, the select list SQLDA contains metadata and data for one row of the result table. The SQLDA.SQLD field contains the number of columns in the row. Set up another loop that iterates once for each column.

```
for (i = 0; i < select_sqlda->SQLD; i++) {
    .
    .
    .
} /* for */
```

6 Within the inner loop, check the indicator variable for a NULL value. If a non-NULL value is present, set up a dispatch table based on the data type of the column.

```
if (*select_sqlda->SQLVARARY[i].SQLIND < 0)
    printf("NULL\n");
else
    switch (select_sqlda->SQLVARARY[i].SQLTYPE) {
    case SQLSRV_ASCII_STRING:
    case SQLSRV_GENERALIZED_NUMBER:
    case SQLSRV_GENERALIZED_DATE:
        printf("%s\n", select_sqlda->SQLVARARY[i].SQLDATA);
        break;
    case SQLSRV_VARCHAR:
          .
          .
          .
        break;
    } /* switch */
```

Again, the execute_statement function uses the loop variable as an index into the SQLVARARY.

7 For counted strings (SQLSRV_VARCHAR), which are typically used to store binary data, your application must:

a Read the signed word integer from the first word of the SQLDATA field of the appropriate SQLVAR element. That integer represents the number of 8-bit bytes of data that follow. If you are programming in C, you can use a cast operator to coerce the data variable into an integer so that you can access the first word.

```
char *p;
     .
     .
     .
p = select_sqlda->SQLVARARY[i].SQLDATA;
len = *(short int *)p;
```

b Use the data in the second and subsequent words of the SQLDATA field of the appropriate SQLVAR element. If you are programming in C, you can use a char pointer to read individual bytes of data from the variable. Use the *sizeof* operator to set the pointer to the first data character.

```
p += sizeof(short int);
printf("%-*.*s\n", len, len, p);
```

Because the execute_statement function uses terminal input/output, it demonstrates the SQLSRV_VARCHAR type by calling the *printf* routine to display ASCII data.

4.2.8.8 Releasing Prepared Statements When a prepared statement is no longer needed, the execute_statement function calls the API routine sqlsrv_release_statement to release the resources allocated for that statement.

```
sts = sqlsrv_release_statement(
                    assoc_id,           /* association handle.      */
                    1,                  /* no. of statement id's.   */
                    &statement_id       /* statement id array.      */
                    );
```

If your application prepares several statements at one time, you can release any or all of them together by passing an array of multiple statement identifiers to the API routine sqlsrv_release_statement. (The sample application prepares only one statement at a time.) In C, an array is a pointer, so by passing a pointer, the execute_statement function is actually passing an array of one element.

4.2.8.9 Error Handling It is a good programming practice to check the status value returned by each call to an API routine.

```
if (sts != SQL_SUCCESS)
    return report_error(assoc_id, sqlca_str, long_error);
```

If an API routine call fails, the sample application calls the function report_error, which contains a dispatch table based on the SQLCODE field of the SQLCA structure.

```
switch (sqlca_str->SQLCODE) {
    .
    .
    .
case SQLSRV_NETERR:
    printf("DECnet returned an error.\n");
    printf("SQLERRD[0]: x%lx\n", sqlca_str->SQLERRD[0]);
    printf("SQLERRD[2]: %d.\n", sqlca_str->SQLERRD[2]);
    sqlsrv_release(assoc_id,stats);
    exit(2);
    break;
    .
    .
    .
case SQL_EOS:
    printf("SELECT or cursor got to end of stream\n");
    break;
    .
    .
    .
} /* switch */
```

When a DECnet error or a server error occurs, the report_error function:

- Prints out the specific error code in SQLERRD[0] (see Table 7–2)
- Prints out the contents of SQLERRD[2], which represents different things depending on the API routine, and in some cases the SQL statement that was executing, as shown in Table 7–3
- Releases the association

The report_error function also prints out error messages returned in the alternative error text buffer (see Section 7.2) by VMS, Rdb/VMS, or dynamic SQL.

```
if (strlen(long_error) != 0)
    printf("%s\n", long_error);
```

4.3 Performance Enhancements

This section describes how to enhance the performance of your application by reducing the number of client/server network messages required to perform operations.

4.3.1 Batched Execution

When your application executes a prepared INSERT, UPDATE, or DELETE statement that contains parameter markers, it can control whether the API sends one row of data at a time to the server for processing or several rows at a time. Frequently, batched execution reduces the number of messages required to complete the operation.

The mechanism for controlling batched execution is the execute_flag parameter in the sqlsrv_execute routine, which is described in Section 6.6. The values of the execute_flag parameter are shown in Table 6–4.

In normal (nonbatched) execution, the API places each set of parameter marker values (rows) in the message buffer and sends the message to the server for execution.

In batched execution, the API stores sets of parameter marker values (rows) in the message buffer but does not send the message to the server until your application signals the end of the batched execution.

If the message buffer becomes full during batched execution, the API sends the message to the server and begins a new message in a manner that is transparent to your application. In that case, when the batched parameter marker values arrive on the server, it stores them in a buffer until your application signals the end of the batched execution. If your application aborts the batched execution, the API clears the buffers on both the client and the server. Thus, the database remains consistent and there is no need to roll back the transaction.

4.3.2 Fetching Multiple Rows

When your application fetches rows from a result table, it can control whether the server sends one row of data at a time to the API or several rows at a time. Fetching multiple rows at a time generally reduces the number of client/server messages required to complete the operation.

The mechanism for fetching multiple rows is the sqlsrv_fetch_many routine, which is described in Section 6.9. The repeat_count parameter specifies the number of rows that the server can send to the API the next time your application calls sqlsrv_fetch. A repeat_count value of 0 gets the entire result table.

When the call to sqlsrv_fetch_many completes, the next call to sqlsrv_fetch causes the API to get multiple rows of data and store them in the message buffer. Then, subsequent calls to sqlsrv_fetch can fetch rows without client/server messages.

For example:

```
status = sqlsrv_fetch( . . .  ); /* gets 1 row  */
status = sqlsrv_fetch_many( . . .  3 . . .  );
status = sqlsrv_fetch( . . .  ); /* gets 3 rows */
status = sqlsrv_fetch( . . .  ); /* gets 0 rows */
status = sqlsrv_fetch( . . .  ); /* gets 0 rows */
status = sqlsrv_fetch( . . .  ); /* gets 1 row  */
status = sqlsrv_fetch( . . .  ); /* gets 1 row  */
               .
               .
               .
```

When the specified number of rows have been fetched, the API returns to the default behavior (one row at a time), which is necessary when executing the SQL statements UPDATE . . . WHERE CURRENT OF cursor-name and DELETE . . . WHERE CURRENT OF cursor-name.

If a sqlsrv_fetch_many operation requests more rows than can fit in the message buffer at one time, the API clears and refills the message buffer in a manner that is transparent to your application.

4.4 Filtering Result Tables

This section describes how your application can instruct the server to discard unwanted rows from a result table before sending them to the client, reducing the number of client/server messages required to complete the operation.

The sqlsrv_set_filter routine (Section 6.17) allows your application to define a Boolean (true/false) expression and to associate that **filter expression** with a cursor. When your application fetches rows from the result table, the server evaluates the expression for each row and filters out (discards) those rows for which the expression returns a value of false.

4.4.1 Elements of Filter Expressions

The syntax of filter expressions is similar to that of most high-level programming languages.

The operands that can be used to form filter expressions are:

- *Constants*, as described in Section 4.4.2
- *Placeholders*, as described in Section 4.4.3
- *Functions*, as described in Appendix A

The operators that can be used to form filter expressions are:

- *Mathematical* operators, as described in Section 4.4.4
- *Relational* operators, as described in Section 4.4.5

- *Logical* operators, as described in Section 4.4.6
- *String* operators, as described in Section 4.4.7

The precedence of the operators is described in Section 4.4.8.

4.4.2 Constants

The following types of constants can be used in filter expressions:

Character	ASCII string delimited by double quotes, single quotes, or brackets.
Numeric	Decimal or E notation. The internal representation of numeric data is floating-point.
Date	Character string in the format {mm/dd/yy} (see Section 5.2 and Section A.13).

4.4.3 Placeholders

Variables in filter expressions are represented by placeholders (question marks) that correspond to columns in the result table. An index array maps the placeholders to values in the select list SQLDA. Although they are ASCII strings, the SQL/Services data types (see Chapter 5) behave as if they were binary; SQLSRV_GENERALIZED_NUMBER data behave as floating-point numeric data, and SQLSRV_GENERALIZED_DATE data behave as date type data.

For example, suppose that your application prepares the following SELECT statement in which columns A, B, and C are numeric data:

```
SELECT A,B,C FROM NUMBERS
```

The only useful rows from the result table are those for which the following algebraic expression is true:

```
SIN(C + A) + 12 > B
```

Your application would specify the following filter expression, replacing the variables with "?" placeholders:

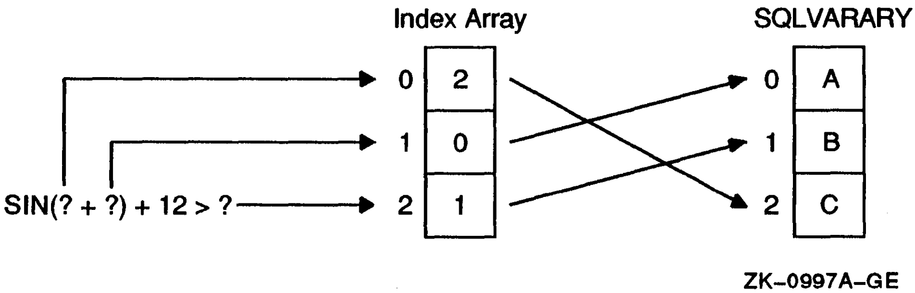
```
SIN(? + ?) + 12 > ?
```

When your application calls sqlsrv_set_filter, it associates the placeholders with columns in the result table by passing an **index array** into the select list SQLDA. The first element of the index array corresponds to the leftmost placeholder, and so forth. You would set up the index array as shown:

```
sqlda_index_array[0] = 2; /* "C" */
sqlda_index_array[1] = 0; /* "A" */
sqlda_index_array[2] = 1; /* "B" */
```

The values of the array elements are zero-based indexes into the array of SQLVAR structures, each element of which represents a column, as shown in Figure 4–2. The first placeholder corresponds to column C, the data and metadata for which is in SQLDA.SQLVARARY[2].

Figure 4–2 Placeholders in Filter Expressions



4.4.4 Mathematical Operators

Mathematical operators in filter expressions generate numeric results.

Operator	Description	Precedence
( )	Grouping	1
+	Unary Positive	2
–	Unary Negative	2
** or ^	Exponentiation	3
*	Multiplication	4
/	Division	4
+	Addition	5
–	Subtraction	5

4.4.5 Relational Operators

Relational operators in filter expressions generate logical results; that is, true (.T.) or false (.F.). You can use relational operators with character, numeric, date, or logical operands. However, both operands in a relational expression must be of the same type. Relational operators have only one level of precedence and are performed in order from left to right.

Operator	Description
<	Less than
>	Greater than
=	Equal to
<> or #	Not equal to
<=	Less than or equal to
>=	Greater than or equal to
$	Substring comparison. (For example, if A and B are character strings, A$B returns a logical true if A is either identical to B or contained within B.)

4.4.6 Logical Operators

Logical operators in filter expressions obtain a logical result from comparing two expressions.

Operator	Description	Precedence
( )	Grouping	1
.NOT.	Logical not	2
.AND.	Logical and	3
.OR.	Logical or	4

4.4.7 String Operators

String operators in filter expressions concatenate two or more character strings into a single character string. String operators have only one level of precedence and are performed in order from left to right.

Operator	Description
( )	Grouping
+	Trailing spaces between the strings are left intact when the strings are joined.
–	Trailing spaces between the strings are moved to the end of the last string.

4.4.8 Precedence of Operators

When several of the four types of operators are used in the same filter expression, the precedence levels are:

1 Mathematical or string

2 Relational

3 Logical

All operations of the same precedence level are performed in order from left to right. Parentheses override the order in which operations are performed. Operations within nested parentheses are performed first.

4.5 Execution Logging

This section describes how to use various types of execution logging to help debug and monitor the performance of SQL/Services applications.

The mechanism for enabling or disabling logging is the association structure (see Section 7.2). It contains two fields, CLIENT_LOG and SERVER_LOG, into which you place one or more of the values defined in the include file SQLSRV.H, which are:

SQLSRV_LOG_DISABLED	Disables logging (default)
SQLSRV_LOG_ASSOCIATION	Enables association logging
SQLSRV_LOG_ROUTINE	Enables API routine logging
SQLSRV_LOG_PROTOCOL	Enables message protocol logging
SQLSRV_LOG_SCREEN	Sends logging output to the video display on the client system as well as to the log file

All types of logging are valid on the client system; on the server system, however, only message protocol logging is valid.

To enable more than one type of logging, add the appropriate constants. For example:

```
associate_str.CLIENT_LOG = SQLSRV_LOG_ROUTINE + SQLSRV_LOG_SCREEN;
```

When you enable client logging, the API writes information into the file CLIENT.LOG in the SQL/Services application program's current working directory. When you enable server logging, the server process writes information into the file SQLSRV.LOG in the default directory of the association's UIC.

4.5.1 Association Logging

Association logging occurs whenever a client/server association is created, terminated, or aborted. Use this type of logging to debug server access in application programs.

Depending on the API routine called, association log entries include some or all of the following items:

❶ A header that identifies the entry as ASSOCIATE LEVEL LOG

❷ The name of the API routine

❸ The association identifier

❹ The name of the server node

❺ The name of the user account on the server

❻ The error status for the API routine

❼ The detailed error code for network or server errors

For example:

```
ASSOCIATE LEVEL LOG ❶
----SQLSRV_ASSOCIATE ❷
--------SQLSRV_ASSOCIATE ID: 106520 ❸
--------NODE: abcdef, ❹ USERNAME: xxxxxx, ❺ SQLCODE: 0, ❻ SQLERRD[0] 0 ❼
```

These messages indicate that an association with a server system was created and terminated normally.

4.5.2 Routine Logging

Routine logging occurs whenever your application calls an SQL/Services API routine. Use this type of logging to debug execution flow in application programs.

Routine log entries include some or all of the following items:

❶ A header that identifies the entry as ROUTINE LEVEL LOG

❷ The name of the API routine

❸ The length in bytes of the SQL statement string

❹ The SQL statement string

❺ The name of the cursor

❻ The SQL statement identifier

❼ The execution flag

For example:

```
ROUTINE LEVEL LOG ❶
----SQLSRV_PREPARE ❷
--------SQL STATEMENT
------------len: 45, ❸ value: select * from sqlsrv_table where USERNAME = ? ❹

ROUTINE LEVEL LOG
----SQLSRV_OPEN_CURSOR
--------CURSOR NAME
------------sqlsrv_cursor ❺
--------STATEMENT ID
             1199896 ❻

ROUTINE LEVEL LOG
----SQLSRV_EXECUTE
--------STATEMENT ID
------------1199896
--------EXECUTE FLAG
             0 ❼
      .
      .
      .
```

Routine log entries that follow the sqlsrv_prepare routine also include metadata:

❶ The type of SQLDA (parameter marker or select list)

❷ The number of parameter markers or select list items

❸ The SQL/Services data type

❹ For non-numeric data, the length of the data variable

❺ For numeric data, the length of the data variable and the scale factor (see Section 7.6)

❻ The name of the column

For example:

```
ROUTINE LEVEL LOG
----SELECT LIST SQLDA ❶
--------SQLDA: SQLD 4 ❷
--------[0].SQLTYPE: SQLSRV_ASCII_STRING, ❸ SQLLEN: 33 ❹
------------SQLNAME: USERNAME
--------[1].SQLTYPE: SQLSRV_GENERALIZED_NUMBER, SQLLEN[0] 12, SQLLEN[1] 0 ❺
------------SQLNAME: INTEGER_VALUE ❻
--------[2].SQLTYPE: SQLSRV_GENERALIZED_NUMBER, SQLLEN[0] 24, SQLLEN[1] 0
------------SQLNAME: DOUBLE_VALUE
--------[3].SQLTYPE: SQLSRV_GENERALIZED_DATE, SQLLEN: 17
------------SQLNAME: DATE_VALUE
```

Routine log entries that follow the sqlsrv_fetch, sqlsrv_open_cursor, and sqlsrv_execute routines also include data:

❶ The type of SQLDA (parameter marker or select list)

❷ The number of parameter markers or select list items

❸ The SQL/Services data type

❹ The value of the indicator variable

❺ The length of the value of the data variable

❻ The value of the data variable

For example:

```
ROUTINE LEVEL LOG
----SELECT LIST SQLDA ❶
--------SQLDA: SQLD 4 ❷
------------[0].SQLTYPE: SQLSRV_ASCII_STRING, ❸ SQLIND: 0 ❹
----------------len: 32, ❺ value: xxxxxx ❻
------------[1].SQLTYPE: SQLSRV_GENERALIZED_NUMBER, SQLIND: 0
----------------len: 11, value: 1
------------[2].SQLTYPE: SQLSRV_GENERALIZED_NUMBER, SQLIND: 0
----------------len: 23, value:  1.280000000000000E+002
------------[3].SQLTYPE: SQLSRV_GENERALIZED_DATE, SQLIND: 0
----------------len: 16, value: 1988070100000000
```

4.5.3 Message Protocol Logging

Message protocol logging occurs whenever a message is transmitted between the client API and the server process. Use this type of logging to verify that the SQL/Services client/server communications protocol is working as expected.

Protocol log entries include some or all of the following items:

❶ A header that identifies the entry as PROTOCOL LEVEL

❷ The word CLIENT or SERVER to indicate where the log file was written

❸ The word “read” or “write” to indicate whether the packet was received or transmitted, respectively

❹ The packet identification number, which is incremented from 0 from the beginning of the association

❺ The packet sequence number, which is used in the following instances:

- Batched execution
- Multiple row fetches
- Any message that is too large for a single packet

❻ The message tag, which either specifies a routine to be executed on the server, an acknowledgment (ACK) that the routine was executed, or an error (ERROR) message

❼ Tags that represent routine parameters, including:

  ❽ The SQL/Services data type

  ❾ The total length in bytes of the data

  ❿ The number of bytes of data in this packet

  ⓫ The data value

  ⓬ Subtags that describe SQLDA structures

For example:

```
PROTOCOL LEVEL LOG ❶ CLIENT: ❷  write ❸
----PACKET ID: 11, ❹  PACKET SEQUENCE: 0 ❺
--------SQLSRV_FETCH ❻
------------CURSOR NAME ❼
----------------SQLSRV_ASCII_STRING, ❽ len: 13 ❾
--------------------len: 13,  ❿ value: sqlsrv_cursor ⓫
--------END OF MESSAGE

PROTOCOL LEVEL LOG CLIENT: read
----PACKET ID: 11, PACKET SEQUENCE: 0
--------SQLSRV_FETCH ACK
------------FETCH ROW NUMBER
----------------SQLSRV_GENERALIZED_NUMBER, len: 1
--------------------len: 1, value: 1
------------SELECT LIST DATA ❼
----------------len: 2, value: 4
------------SQLVAR ⓬
----------------len: 2, value: 0
------------SQLDATA ⓬
----------------SQLSRV_ASCII_STRING, len: 32
--------------------len: 32, value: xxxxxx
------------SQLIND ⓬
----------------len: 2, value: 0
       .
       .
       .
--------END OF MESSAGE
```

5

Data Types and Environment Variables

SQL/Services supports a subset of the SQL data types. Declarations for data type names and constant values are provided in the include file SQLSRVDA.H.

In filter expressions, SQL/Services uses environment variables to control the format of date type data and the way that string matching works.

5.1 Data Types

The SQL data types are listed in Table 5–1 with their SQL/Services representation.

Table 5–1 Data Types

SQL Data Type	SQL/Services Data Type
SQL_INTEGER	SQLSRV_GENERALIZED_NUMBER
SQL_SMALLINT	SQLSRV_GENERALIZED_NUMBER
SQL_FLOAT	SQLSRV_GENERALIZED_NUMBER
SQL_CHAR	SQLSRV_ASCII_STRING
SQL_VARCHAR	SQLSRV_VARCHAR
SQL_DATE	SQLSRV_GENERALIZED_DATE
SQL_DECIMAL	SQLSRV_GENERALIZED_NUMBER
SQL_QUADWORD	SQLSRV_GENERALIZED_NUMBER
SQL_NUMERIC	SQLSRV_GENERALIZED_NUMBER

5.1.1 SQLSRV_ASCII_STRING

The SQLSRV_ASCII_STRING data type is an array of 8-bit bytes containing ASCII characters. A byte containing 0 (the *null* character) indicates the end of the data. This data type is commonly known as an ASCIZ or null-terminated string.

5.1.2 SQLSRV_GENERALIZED_NUMBER

The SQLSRV_GENERALIZED_NUMBER data type is an SQLSRV_ASCII_STRING that is used to represent all numeric values. The format is:

[-][NNN][.DD][E[-][xx]]

–	unary minus
NNN	integer portion of the number
.DD	decimal portion of the number
E	exponent identifier
–	unary minus for exponent value
xx	exponent value

The brackets indicate the optional syntax. The one requirement is that either the integer or decimal portion of the number must be specified.

5.1.3 SQLSRV_GENERALIZED_DATE

The SQLSRV_GENERALIZED_DATE data type is an SQLSRV_ASCII_STRING that is used to represent all dates. The format is:

ccyymmdd[hh[mi[ss[ff]]]]

cc	century
yy	year
mm	month
dd	day
hh	hour (24-hour format)
mi	minute
ss	second
ff	fractions of a second

If you omit any of the optional fields, SQL/Services pads the string with zeros. Thus, the default time is exactly midnight.

For example: May 4, 1989 11:04 a.m. would be represented as: 198905041104.

5.1.4 SQLSRV_VARCHAR

The SQLSRV_VARCHAR data type is a signed word integer followed by an array of 8-bit bytes that can be used to store any sort of data, including binary. The signed word contains the number of bytes that contain data. This type is commonly known as a *counted string*. The maximum length of an SQLSRV_VARCHAR is 16,383 bytes.

5.2 Environment Variables

Environment variables (SQLSRV_ENV_DATE, SQLSRV_ENV_CENTURY, and SQLSRV_ENV_SET_EXACT) control the format of date type data and the way that string matching works in filter expressions. For more information, see:

- filter expressions (Section 4.4)
- sqlsrv_get_environment (Section 6.11)
- sqlsrv_set_environment (Section 6.16)
- sqlsrv_env_str (Section 7.7)

5.2.1 SQLSRV_ENV_DATE

The SQLSRV_ENV_DATE variable controls the format of the date values used in filter expressions. The settings are shown in Table 5–2.

Table 5–2 Settings for the SQLSRV_ENV_DATE Variable

Setting	Value	Result	
SQLSRV_ENV_DATE_AMERICAN	0	mm/dd/yy	Default
SQLSRV_ENV_DATE_BRITISH	1	dd/mm/yy	
SQLSRV_ENV_DATE_GERMAN	2	dd.mm.yy	
SQLSRV_ENV_DATE_JAPAN	3	yy/mm/dd	
SQLSRV_ENV_DATE_ANSI	4	yy.mm.dd	
SQLSRV_ENV_DATE_FRENCH	5	dd/mm/yy	
SQLSRV_ENV_DATE_ITALIAN	6	dd-mm-yy	
SQLSRV_ENV_DATE_USA	7	mm-dd-yy	

5.2.2 SQLSRV_ENV_CENTURY

The SQLSRV_ENV_CENTURY variable controls whether the century prefix is included as part of the date format. The settings are shown in Table 5–3.

Table 5–3 Settings for the SQLSRV_ENV_CENTURY Variable

Setting	Value	Result	
SQLSRV_ENV_CENTURY_OFF	0	Century is OFF.	Default
SQLSRV_ENV_CENTURY_ON	1	Century is ON.	

5.2.3 SQLSRV_ENV_SET_EXACT

The SQLSRV_ENV_SET_EXACT variable controls whether a comparison between two character strings requires the strings to be the same length. The settings are shown in Table 5–4.

Table 5–4 Settings for the SQLSRV_ENV_SET_EXACT Variable

Setting	Value	Result	
SQLSRV_ENV_SET_EXACT_OFF	0	Comparisons between character strings begin with the left character in each string and continue character-by-character to the end of the string on the right of the relational operator. If the two strings are equivalent up to that point, the comparison returns a value of true.	Default
SQLSRV_ENV_SET_EXACT_ON	1	The comparison of characters in each string is the same except that both character strings must be the same length for the comparison to return a value of true.	

6
API Routines

This chapter describes the routines in the SQL/Services client Application Programming Interface (API).

6.1 Documentation Format

Each SQL/Services API routine is documented using a structured format called the routine template. The sections of the routine template are listed in Table 6–1, along with the information that is presented in each section and the format used to present the information. Some sections require no further explanation beyond what is given in Table 6–1. Those that require additional explanation are discussed in the remaining subsections of this section.

Table 6–1 Sections in the Routine Template

Section	Description
Routine Name	Appears at the top of the page, followed by the English name of the routine
Overview	Appears directly below the routine name and explains, usually in one or two sentences, what the routine does
VAX Format	Gives the routine entry point name and the routine argument list; also specifies whether arguments are required or optional
C Format	Shows the C function prototype from the include file SQLSRV.H
Parameters	Gives detailed information about each parameter

(continued on next page)

Table 6–1 (Cont.) Sections in the Routine Template

Section	Description
Description	Contains detailed information about specific actions taken by the routine, interaction between routine arguments, operation of the routine within the context of a specific operating system, and resources used by the routine
Notes	Contains additional pieces of information related to applications programming
Errors	Lists the SQL/Services errors that can occur in the routine
SQL Errors	Lists the SQL errors (if any) that can occur in the routine

6.1.1 Routine Name

The SQL/Services API routine names are shown in the form sqlsrv_xxx throughout the manual. In most Digital software documentation, the routine template is language-independent but quite dependent on the VMS operating system. Because the SQL/Services API must be portable across all supported platforms, the routine template in this manual is intended for C programmers who are concerned with portability.

Digital requires that all callable products that run on the VMS operating system have routine names in the format facility_name$routine_name. Thus, the VAX Format section of the template shows the routine name in the format SQLSRV$routine_name.

However, the dollar sign character ($) is not portable to all supported platforms. Some C compilers return a syntax error when they encounter a dollar sign character. Thus, SQL/Services automatically maps routine calls in the portable C format to the dollar sign format in a manner that is transparent to your application.

6.1.2 Return Values

The SQL/Services routine template does not include a "Returns" section. Except where explicitly noted, the SQL/Services API routines return a signed longword integer containing one of the values shown in Table 6–2.

Table 6–2 API Return Values

Return Value	Description
n = SQL_SUCCESS[1]	The routine completed successfully.
n < SQL_SUCCESS	An error occurred during processing. Refer to the SQLCA.SQLCODE for the specific error.
n > SQL_SUCCESS	A warning was issued during processing. Refer to the SQLCA for additional information.

[1] The symbol SQL_SUCCESS is defined as 0 in the include file SQLSRVCA.H.

6.1.3 VAX Format Section

In the VAX Format section:

- The entry point name is shown in uppercase letters.
- The argument names are shown in lowercase letters.
- One or more spaces are used between the entry point name and the first argument, and between each argument and the next.
- Brackets surround optional arguments. In SQL/Services, optional arguments cannot be omitted; a value of 0, passed by value, indicates that the API is to ignore the parameter.
- Commas precede arguments instead of following them.

6.1.4 C Format Section

The C Format section shows the function prototypes for the SQL/Services API routines exactly as they are declared in the include file SQLSRV.H. If you are using a compiler that does not support function prototypes, such as the ULTRIX C compiler, alternative declarations are also provided in SQLSRV.H.

For example, the following is the function prototype for the sqlsrv_execute_immediate routine:

```
extern int sqlsrv_execute_immediate(
                char *associate_id,
                long int database_id,
                char *sql_statement);
```

The following is the alternative function declaration for the same routine:

```
extern int sqlsrv_execute_immediate(associate_id, database_id,
                                    sql_statement)
        char *associate_id;
        long int database_id;
        char *sql_statement;
```

To avoid repetition, #include compiler directives are not repeated in each routine template. When you write SQL/Services programs, use the following #include directives:

```
#include <sqlsrvda.h>    /* SQLDA structure definition.          */
#include <sqlsrvca.h>    /* SQLCA structure definition.          */
#include <sqlsrv.h>      /* SQL/SERVICES structure definitions.  */
```

6.1.5 Parameters Section

The Parameters section contains detailed information about each parameter listed in the call format. Parameters are described in the order in which they appear in the call format.

The following format is used to describe each parameter:

name

data type: the data type of the data specified by the parameter (see Section 6.1.5.1)

access: the way in which the called routine accesses the data specified by the parameter (see Section 6.1.5.2)

mechanism: the way in which a parameter specifies the data to be used by the called routine (see Section 6.1.5.3)

In addition, the Parameters section contains at least one paragraph of text describing the purpose of the parameter.

6.1.5.1 Data Type Entry A parameter does not have a data type; rather, the data specified by the parameter has a data type. The parameter is the vehicle for passing of data to the called routine. However, the term *parameter data type* is used to describe the data type of the data specified by the parameter. Table 6–3 lists the data types used in SQL/Services API routine calls and structures.

Table 6–3 API Parameter Data Types

Data Type	Description
character string	Array of unsigned 8-bit integers
word (signed)	16-bit signed integer
word (unsigned)	16-bit unsigned integer
longword (signed)	32-bit signed integer
longword (signed) array	Array of signed 32-bit integers
longword (unsigned)	32-bit unsigned integer

(continued on next page)

Table 6–3 (Cont.) API Parameter Data Types

Data Type	Description
pointer	32-bit unsigned integer that contains an address
structure	Named collection of variables (*record* in some languages)
structure array	Array of structures
undefined	Memory that is allocated and used by the API but never accessed directly by the application (see the description of the associate_id parameter in Section 6.4)

Regardless of the passing mechanism (described in Section 6.1.5.3), *the data type entry always refers to the data type of the data specified by the parameter.*

6.1.5.2 Access Entry The access entry describes the way in which the called routine accesses the data specified by the parameter. The following three access methods are used:

- Read. Data needed by the called routine to perform its operation is read but not returned.
- Write. Data that the called routine returns to the calling routine is written into a location accessible to the calling routine.
- Modify. Data that is both read and returned by the called routine; input data specified by the parameter is overwritten.

6.1.5.3 Mechanism Entry The parameter passing mechanism is the way in which a parameter specifies the data to be used by the called routine. SQL/Services uses two passing mechanisms:

- By value. The parameter contains a copy of the data to be used by the routine.
- By reference. The parameter contains the address of the data to be used by the routine. In other words, the parameter is a pointer to the data.

  Because C supports only call by value, write parameters other than arrays and structures must be passed by means of pointers (variables that contain the addresses of objects). References to names of arrays and structures are automatically converted by the compiler to pointer expressions.

6.2 sqlsrv_abort—Disconnect Association

The sqlsrv_abort routine drops the network link between the client and server, frees client association resources, and rolls back active transactions on the server.

VAX Format

SQLSRV$ABORT associate_id

C Format

```
extern int sqlsrv_abort(
                    char *associate_id);
```

Parameters

associate_id

data type:	**undefined**
access:	**read**
mechanism:	**by reference**

Handle used to identify the active association.

Errors

SQLSRV_INTERR	Internal error.
SQLSRV_INVASC	Invalid association identifier.
SQLSRV_NETERR	DECnet returned an error.

6.3 sqlsrv_allocate_sqlda_data—Allocate Variables

The sqlsrv_allocate_sqlda_data routine dynamically allocates data and indicator variables. Your application passes an SQLDA structure to sqlsrv_allocate_sqlda_data, which allocates variables of the appropriate data type and writes the addresses of the newly allocated variables into the SQLDATA and SQLIND fields in the SQLVAR array.

VAX Format

SQLSRV$ALLOCATE_SQLDA_DATA associate_id ,sqlda_str

C Format

```
extern int sqlsrv_allocate_sqlda_data(
                char *associate_id,
                struct SQLDA *sqlda_str);
```

Parameters

associate_id

data type:	**undefined**
access:	**read**
mechanism:	**by reference**

Handle used to identify the active association.

sqlda_str

data type:	**structure**
access:	**modify**
mechanism:	**by reference**

An SQLDA structure into whose SQLVAR array the API writes the address of the newly allocated SQLDATA and SQLIND fields. You can pass any valid SQLDA structure; it does not matter how the structure was allocated.

sqlsrv_allocate_sqlda_data—Allocate Variables

Notes

You can free variables allocated by sqlsrv_allocate_sqlda_data explicitly by calling sqlsrv_free_sqlda_data, or implicitly by calling sqlsrv_release_statement or sqlsrv_release.

Errors

SQLSRV_INTERR	Internal error.
SQLSRV_INVARG	Invalid routine parameter.
SQLSRV_INVASC	Invalid association identifier.
SQLSRV_INVSQLDA	Invalid SQLDA structure.
SQLSRV_NO_MEM	API memory allocation failed.

6.4 sqlsrv_associate—Create Client/Server Association

The sqlsrv_associate routine creates a DECnet link between your application and a server process, using the node name, user name, and password input parameters. It creates an association *handle* (identification structure) used in subsequent routine calls and binds specific variables (message protocol buffers and an SQLCA structure) to the association.

VAX Format

SQLSRV$ASSOCIATE node_name ,[user_name] ,[password] ,read_buffer ,write_buffer ,read_buffer_size ,write_buffer_size ,sqlca_str ,associate_str ,associate_id

C Format

```
extern int sqlsrv_associate(
                char *node_name,
                char *user_name,
                char *password,
                char *read_buffer,
                char *write_buffer,
                long int read_buffer_size,
                long int write_buffer_size,
                struct SQLCA *sqlca_str,
                struct ASSOCIATE_STR *associate_str,
                char **associate_id);
```

Parameters

node_name

data type:	**character string**
access:	**read**
mechanism:	**by reference**

A null-terminated string containing the DECnet node name of the VAX system on which the server resides.

user_name (optional)

data type:	**character string**
access:	**read**
mechanism:	**by reference**

A null-terminated string containing the user name within whose context the server session runs. If this parameter is NULL, and a default user name is defined on your system, the API attempts to access the server by means of proxy. If proxy access is disabled on the server, you must supply a user name; otherwise the association fails. (See the *Guide to DECnet-VAX Networking* for information on proxy access and the DECnet documentation for your system for information on setting default access control data.)

password (optional)

data type:	**character string**
access:	**read**
mechanism:	**by reference**

A null-terminated string containing the password for the account within whose context the server session runs.

read_buffer

data type:	**character string**
access:	**modify**
mechanism:	**by reference**

The buffer used by the API to receive messages from the server.

write_buffer

data type:	**character string**
access:	**modify**
mechanism:	**by reference**

The buffer used by the API to build messages to send to the server.

read_buffer_size

data type:	**longword (signed)**
access:	**read**
mechanism:	**by value**

The size in bytes of the API buffer used to receive messages. The maximum value is 65,535 bytes, the minimum value is 256 bytes.

write_buffer_size

data type:	**longword (signed)**
access:	**read**
mechanism:	**by value**

The size in bytes of the API buffer used to send messages. The maximum value is 65,535 bytes, the minimum value is 256 bytes.

sqlca_str

data type:	**structure**
access:	**modify**
mechanism:	**by reference**

An SQLCA (SQL Communications Area) structure (see Section 7.3). Your application must declare an instance of this structure and can refer to it when any API routine called in the context of this association returns a status value other than SQL_SUCCESS. (The SQLCA structure is defined in the include file SQLSRVCA.H, along with all valid SQL/Services error codes.)

associate_str

data type:	**structure**
access:	**modify**
mechanism:	**by reference**

An ASSOCIATE_STR structure, used to define optional association characteristics (see Section 7.2). The ASSOCIATE_STR structure is defined in the include file SQLSRV.H.

associate_id

data type:	**pointer**
access:	**write**
mechanism:	**by reference**

A pointer variable into which the API writes the address of the newly allocated associate_id (an undefined structure never accessed directly by your application). This handle is used by all succeeding routines to identify the active association.

Notes

In selecting buffer sizes for applications that will run on the MS-DOS operating system, you must take into account the limitations of the *small* and *medium* standard memory models in which the data segment is 64K bytes.

Errors

SQLSRV_INTERR	Internal error.
SQLSRV_INVARG	Invalid routine parameter.
SQLSRV_INVASCSTR	Invalid parameter in ASSOCIATE_STR.
SQLSRV_INVBUFSIZ	Invalid read or write buffer size.
SQLSRV_INVSQLCA	Invalid SQLCA structure.
SQLSRV_NETERR	DECnet returned an error.
SQLSRV_NO_MEM	API memory allocation failed.
SQLSRV_OPNLOGFIL	Unable to open log file.

6.5 sqlsrv_close_cursor—Release Result Table

The sqlsrv_close_cursor routine closes an open cursor.

VAX Format

SQLSRV$CLOSE_CURSOR associate_id ,cursor_name

C Format

```
extern int sqlsrv_close_cursor(
                char *associate_id,
                char *cursor_name);
```

Parameters

associate_id

data type:	**undefined**
access:	**read**
mechanism:	**by reference**

Handle used to identify the active association.

cursor_name

data type:	**character string**
access:	**read**
mechanism:	**by reference**

A null-terminated string used to identify the open cursor.

Errors

SQLSRV_INTERR	Internal error.
SQLSRV_INVASC	Invalid association identifier.
SQLSRV_INVCURNAM	Invalid cursor name.

SQLSRV_MULTI_ACT	A batched sqlsrv_execute or sqlsrv_fetch_many context is active.
SQLSRV_NETERR	DECnet returned an error.

SQL Errors

SQL_RDBERR	Rdb/VMS returned an error.

6.6 sqlsrv_execute—Execute Prepared Statement

The sqlsrv_execute routine executes a prepared SQL statement and, if rows were modified, updates the SQLCA.

VAX Format

SQLSRV$EXECUTE associate_id ,database_id ,statement_id ,execute_flag ,parameter_marker_sqlda

C Format

```
extern int sqlsrv_execute(
                char *associate_id,
                long int database_id,
                long int statement_id,
                short int execute_flag,
                struct SQLDA *parameter_marker_sqlda);
```

Parameters

associate_id

data type:	**undefined**
access:	**read**
mechanism:	**by reference**

Handle used to identify the active association.

database_id

data type:	**longword (signed)**
access:	**read**
mechanism:	**by value**

This parameter must be 0. Databases are referenced within the SQL statement syntax.

statement_id

data type:	**longword (signed)**
access:	**read**
mechanism:	**by value**

Variable identifying a previously prepared statement. When batching is enabled, this parameter must remain the same. In other words, before changing this parameter, you must first call the sqlsrv_execute routine and pass an execute_flag parameter with a value of 0 or 2 (signaling that the current batch is finished).

execute_flag

data type:	**word (signed)**
access:	**read**
mechanism:	**by value**

For a prepared INSERT, UPDATE, or DELETE statement that contains parameter markers and is executed more than once, this parameter specifies whether the API sends single or multiple sets of parameter marker values to the server for processing (see Section 4.3.1). For all other prepared SQL statements, this value must be 0. The values of the execute_flag parameter are shown in Table 6–4.

Table 6–4 Values of the execute_flag Parameter

Value	Function	Description
0	Nonbatched execution	Sends the contents of the message buffer to the server for execution, including the current parameter marker values.
1	Begins batched execution	Stores the current parameter marker values in the message buffer but does not send the contents of the buffer to the server.
2	Ends batched execution	Sends the contents of the message buffer to the server for execution, *not* including the current parameter marker values.
3	Aborts batched execution	Clears the contents of the message buffer and clears all parameter marker values waiting to execute on the server.

parameter_marker_sqlda

data type:	**longword (unsigned)**
access:	**read**
mechanism:	**by reference**

An SQLDA structure defining the parameter marker values for the SQL statement to be executed.

Notes

- When you execute an UPDATE or DELETE statement, a single set of parameter marker values can affect many rows. Thus, when your application requests execution by calling the sqlsrv_execute routine with an execute_flag parameter of 0 or 2, the API places the following status information in the SQLCA structure:
  - The SQLERRD[1] contains the number of statements (sets of parameter marker values) successfully executed.
  - The SQLERRD[2] contains the number of rows inserted, updated, or deleted.

  See Section 7.3 for more information about the SQLCA structure.
- Batched execution stops (the sqlsrv_execute routine returns) if there is an error.
- If batched execution would result in a message buffer overflow, the API sends the contents of the buffer to the server but does not request execution.
- During batched execution, you cannot call API routines other than sqlsrv_execute; you must complete the batched execution before calling other routines.

Errors

SQLSRV_INTERR	Internal error.
SQLSRV_INVARG	Invalid routine parameter.
SQLSRV_INVASC	Invalid association identifier.
SQLSRV_INVEXEFLG	Invalid execute flag.

SQLSRV_INVSQLDA	Invalid SQLDA structure.
SQLSRV_INVSTMID	Invalid statement identifier.
SQLSRV_MULTI_ACT	A batched sqlsrv_execute or sqlsrv_fetch_many context is active.
SQLSRV_NETERR	DECnet returned an error.

SQL Errors

SQL_BAD_TXN_STATE	Invalid transaction state.
SQL_DEADLOCK	Deadlock encountered.
SQL_INTEG_FAIL	Constraint failed.
SQL_LOCK_CONFLICT	Lock conflict.
SQL_NOT_VALID	Valid-if failed.
SQL_NO_DUP	Duplicate on index.
SQL_RDBERR	Rdb/VMS returned an error.
SQL_ROTXN	Read/write operation in read-only transaction.
SQL_UDCURNOPE	Cursor in update or delete is not open.
SQL_UDCURNPOS	Cursor in update or delete is not positioned on a record.

6.7 sqlsrv_execute_immediate—Prepare and Execute Statement

The sqlsrv_execute_immediate routine prepares and executes an SQL statement that does not contain parameter markers, and updates the SQLCA with a value representing the number of rows modified as a result of the SQL statement execution.

VAX Format

SQLSRV$EXECUTE_IMMEDIATE associate_id ,database_id ,sql_statement

C Format

```
extern int sqlsrv_execute_immediate(
                char *associate_id,
                long int database_id,
                char *sql_statement);
```

Parameters

associate_id

data type:	**undefined**
access:	**read**
mechanism:	**by reference**

Handle used to identify the active association.

database_id

data type:	**longword (signed)**
access:	**read**
mechanism:	**by value**

This parameter must be 0. Databases are referenced within the SQL statement syntax.

sql_statement

data type:	**character string**
access:	**read**
mechanism:	**by reference**

A null-terminated string containing the SQL statement to be prepared and executed by dynamic SQL.

Errors

SQLSRV_INTERR	Internal error.
SQLSRV_INVARG	Invalid routine parameter.
SQLSRV_INVASC	Invalid association identifier.
SQLSRV_MULTI_ACT	A batched sqlsrv_execute or sqlsrv_fetch_many context is active.
SQLSRV_NETERR	DECnet returned an error.

SQL Errors

SQL_BAD_TXN_STATE	Invalid transaction state.
SQL_INTEG_FAIL	Constraint failed.
SQL_LOCK_CONFLICT	Lock conflict.
SQL_NOT_VALID	Valid-if failed.
SQL_NO_DUP	Duplicate on index.
SQL_RDBERR	Rdb/VMS returned an error.
SQL_ROTXN	Read/write operation in read-only transaction.
SQL_UDCURNOPE	Cursor in update or delete is not open.
SQL_UDCURNPOS	Cursor in update or delete is not positioned on a record.

6.8 sqlsrv_fetch—Get Row from Result Table

The sqlsrv_fetch routine fetches a row of data into a select list SQLDA.

VAX Format

SQLSRV$FETCH associate_id ,cursor_name ,direction ,row_number ,select_list_sqlda

C Format

```
extern int sqlsrv_fetch(
                char *associate_id,
                char *cursor_name,
                short int direction,
                long int row_number,
                struct SQLDA *select_list_sqlda);
```

Parameters

associate_id

data type:	**undefined**
access:	**read**
mechanism:	**by reference**

Handle used to identify the active association.

cursor_name

data type:	**character string**
access:	**read**
mechanism:	**by reference**

A null-terminated string used to identify the open cursor.

direction

data type:	**word (signed)**
access:	**read**
mechanism:	**by value**

This parameter is reserved and must be 0.

row_number

data type:	**longword (signed)**
access:	**read**
mechanism:	**by value**

This parameter is reserved and must be 0.

select_list_sqlda

data type:	**longword (unsigned)**
access:	**modify**
mechanism:	**by reference**

The select list SQLDA structure in which to store the row.

Notes

- A return value of SQL_EOS indicates *end of data*, that is, no more rows appear in the result table. A call to the sqlsrv_fetch routine that returns a status code of SQL_EOS does not return any data in the SQLDA. All rows in the result table were returned by the preceding fetches.
- Although it modifies only one SQLDA structure per call, the sqlsrv_fetch routine can download several rows of data when called within a sqlsrv_fetch_many context. See Section 4.3.2 and Section 6.9.

Errors

SQLSRV_CNDERR	Filter run-time error.
SQLSRV_INTERR	Internal error.
SQLSRV_INVARG	Invalid routine parameter.
SQLSRV_INVASC	Invalid association identifier.
SQLSRV_INVCURNAM	Invalid cursor name.
SQLSRV_INVSQLDA	Invalid SQLDA structure.
SQLSRV_MULTI_ACT	A batched sqlsrv_execute or sqlsrv_fetch_many context is active.
SQLSRV_NETERR	DECnet returned an error.

SQL Errors

SQL_CURNOTOPE	Cursor is not open.
SQL_DEADLOCK	Deadlock encountered.
SQL_EOS	SELECT or cursor got to end of stream.
SQL_LOCK_CONFLICT	Lock conflict.
SQL_NULLNOIND	NULL value and no indicator variable.

6.9 sqlsrv_fetch_many—Get Multiple Rows from Result Table

The sqlsrv_fetch_many routine causes the sqlsrv_fetch routine to transfer multiple rows of data from the server, as described in Section 4.3.2. Frequently, this reduces the number of client/server messages required to complete the operation. By default, sqlsrv_fetch gets one row of data at a time.

VAX Format

SQLSRV$FETCH_MANY associate_id ,cursor_name ,direction ,repeat_count

C Format

```
extern int sqlsrv_fetch_many(
                    char *associate_id,
                    char *cursor_name,
                    short int direction,
                    short int repeat_count);
```

Parameters

associate_id

data type:	**undefined**
access:	**read**
mechanism:	**by reference**

Handle used to identify the active association.

cursor_name

data type:	**character string**
access:	**read**
mechanism:	**by reference**

A null-terminated string used to identify the open cursor.

direction

data type:	**word (signed)**
access:	**read**
mechanism:	**by value**

This parameter is reserved and must be 0.

repeat_count

data type:	**word (signed)**
access:	**read**
mechanism:	**by value**

The number of rows to fetch. A value of 0 fetches the entire result table. A value other than 0 fetches that number of rows. For example, an application might fetch enough rows to fill one screen.

Notes

- When you specify a repeat_count other than 0, your application must call the sqlsrv_fetch_many routine again once the specified number of rows have been fetched. Otherwise, the API returns to the default behavior (one row for each call to the sqlsrv_fetch routine).
- During an sqlsrv_fetch_many operation, you cannot call API routines other than sqlsrv_fetch. In other words, you must complete the operation before calling other routines.
- A call to the sqlsrv_close_cursor routine aborts an sqlsrv_fetch_many operation.
- SQL/Services prevents buffer overflow on the client in a manner that is transparent to your application.
- By default, the sqlsrv_fetch routine downloads only one row of data. That way, your application can execute the SQL statements UPDATE . . . WHERE CURRENT OF cursor-name and DELETE . . . WHERE CURRENT OF cursor-name without having to reset the context.

sqlsrv_fetch_many—Get Multiple Rows from Result Table

Errors

SQLSRV_INTERR	Internal error.
SQLSRV_INVARG	Invalid routine parameter.
SQLSRV_INVASC	Invalid association identifier.
SQLSRV_INVCURNAM	Invalid cursor name.
SQLSRV_INVREPCNT	Invalid repeat count.
SQLSRV_MULTI_ACT	A batched sqlsrv_execute or sqlsrv_fetch_many context is active.

6.10 sqlsrv_free_sqlda_data—Release Variables

The sqlsrv_free_sqlda_data routine frees data and indicator variables that were dynamically allocated by the sqlsrv_allocate_sqlda_data routine. Your application passes an SQLDA structure to the API, which frees the variables and writes zeros into the SQLDATA and SQLIND fields of the SQLVAR array.

VAX Format

SQLSRV$FREE_SQLDA_DATA associate_id ,sqlda_str

C Format

```
extern int sqlsrv_free_sqlda_data(
                char *associate_id,
                struct SQLDA *sqlda_str);
```

Parameters

associate_id

data type:	**undefined**
access:	**read**
mechanism:	**by reference**

Handle used to identify the active association.

sqlda_str

data type:	**longword (unsigned)**
access:	**modify**
mechanism:	**by reference**

An SQLDA structure to modify.

Errors

SQLSRV_INTERR	Internal error.
SQLSRV_INVASC	Invalid association identifier.
SQLSRV_INVSQLDA	Invalid SQLDA structure.
SQLSRV_MULTI_ACT	A batched sqlsrv_execute or sqlsrv_fetch_many context is active.
SQLSRV_SQLDA_NOTALL	Attempt to deallocate static memory.

6.11 sqlsrv_get_environment—Return Environment Variable Values

The sqlsrv_get_environment routine returns the values of environment variables (as described in Section 5.2).

VAX Format

SQLSRV$GET_ENVIRONMENT associate_id ,env_str_array_count ,env_str_array

C Format

```
extern int sqlsrv_get_environment(
                char *associate_id,
                unsigned short int env_str_array_count,
                struct SQLSRV_ENV_STR *env_str_array);
```

Parameters

associate_id

data type:	**undefined**
access:	**read**
mechanism:	**by reference**

Handle used to identify the active association.

env_str_array_count

data type:	**word (unsigned)**
access:	**read**
mechanism:	**by value**

Specifies the number of env_str_array entries.

env_str_array

data type:	**structure array**
access:	**modify**
mechanism:	**by reference**

Array of SQLSRV_ENV_STR structures (described in Section 7.7), each of which contains the information necessary to get an environment variable.

sqlsrv_get_environment—Return Environment Variable Values

Description

Your application allocates an array of SQLSRV_ENV_STR structures and sets the values of the ENV_TAG fields, which identify specific environment variables. To request information on all environment variables, set the env_str_array[0].ENV_TAG field to SQLSRV_ENV_ALL. The env_str_array must be large enough to receive all of the values. The number of values returned is placed in the SQLCA.SQLERRD[2] field.

Errors

SQLSRV_INTERR	Internal error.
SQLSRV_INVARG	Invalid routine parameter.
SQLSRV_INVASC	Invalid association identifier.
SQLSRV_INVENVTAG	Invalid environment tag.
SQLSRV_INVENVVAR	Invalid environment variable.
SQLSRV_MULTI_ACT	A batched sqlsrv_execute or sqlsrv_fetch_many context is active.
SQLSRV_NETERR	DECnet returned an error.

6.12 sqlsrv_open_cursor—Create Result Table

The sqlsrv_open_cursor routine opens a cursor for a prepared SELECT statement. In SQL/Services Version 3.1, sqlsrv_open_cursor reduces network traffic by implicitly invoking the dynamic SQL statement DECLARE CURSOR.

VAX Format

```
SQLSRV$OPEN_CURSOR  associate_id ,cursor_name ,statement_id
                    ,parameter_marker_sqlda
```

C Format

```
extern int sqlsrv_open_cursor(
                char *associate_id,
                char *cursor_name,
                long int statement_id,
                struct SQLDA *parameter_marker_sqlda);
```

Parameters

associate_id

data type:	**undefined**
access:	**read**
mechanism:	**by reference**

Handle used to identify the active association.

cursor_name

data type:	**character string**
access:	**read**
mechanism:	**by reference**

A null-terminated string containing the result table identifier. All cursor operations, including positional UPDATE and DELETE statements, must use the cursor_name to identify the cursor.

statement_id

data type:	**longword (signed)**
access:	**read**
mechanism:	**by value**

The identifier of the prepared SELECT statement. The sqlsrv_open_cursor routine maps the cursor_name to the prepared statement.

parameter_marker_sqlda

data type:	**longword (unsigned)**
access:	**read**
mechanism:	**by reference**

An SQLDA structure defining the parameter marker values for the prepared SELECT statement.

Errors

SQLSRV_INTERR	Internal error.
SQLSRV_INVARG	Invalid routine parameter.
SQLSRV_INVASC	Invalid association identifier.
SQLSRV_INVCURNAM	Invalid cursor name.
SQLSRV_INVSQLDA	Invalid SQLDA structure.
SQLSRV_INVSTMID	Invalid statement identifier.
SQLSRV_MULTI_ACT	A batched sqlsrv_execute or sqlsrv_fetch_many context is active.
SQLSRV_NETERR	DECnet returned an error.

SQL Errors

SQL_CURALROPE	Cursor is already open.
SQL_LOCK_CONFLICT	Lock conflict.
SQL_RDBERR	Rdb/VMS returned an error.

6.13 sqlsrv_prepare—Compile Statement and Initialize Structures

The sqlsrv_prepare routine prepares (compiles) the input SQL statement and returns a value that identifies the prepared statement. It also initializes SQLDA structures describing the parameter markers and select list items in the SQL statement (it implicitly invokes the dynamic SQL DESCRIBE statement to reduce message traffic).

VAX Format

```
SQLSRV$PREPARE  associate_id ,database_id ,sql_statement ,statement_id
                ,parameter_marker_sqlda, select_list_sqlda
```

C Format

```
extern int sqlsrv_prepare(
                char *associate_id,
                long int database_id,
                char *sql_statement,
                long int *statement_id,
                struct SQLDA **parameter_marker_sqlda,
                struct SQLDA **select_list_sqlda);
```

Parameters

associate_id

data type:	**undefined**
access:	**read**
mechanism:	**by reference**

Handle used to identify the active association.

database_id

data type:	**longword (signed)**
access:	**read**
mechanism:	**by value**

This parameter must be 0. Databases are referenced within the SQL statement syntax.

sql_statement

data type:	**character string**
access:	**read**
mechanism:	**by reference**

A null-terminated string containing the SQL statement to be prepared.

statement_id

data type:	**longword (signed)**
access:	**write**
mechanism:	**by reference**

The identifier used in all subsequent references to the prepared statement.

parameter_marker_sqlda

data type:	**longword (unsigned)**
access:	**modify/write**
mechanism:	**by reference**

An SQLDA structure used for parameter markers. If the value passed by the caller is the address of an existing SQLDA structure, the API writes metadata into that structure. If the SQL statement contains one or more parameter markers ("?" placeholders), there must be at least one SQLVAR structure for each parameter marker.

If the value passed by the caller is NULL, the API determines whether an SQLDA structure is needed. If an SQLDA is needed, the API performs the following operations; otherwise it leaves the value NULL:

- Dynamically allocates an SQLDA structure containing the requisite number of SQLVAR structures
- Writes parameter marker metadata into the SQLDA
- Returns the address of the SQLDA

select_list_sqlda

data type:	**longword (unsigned)**
access:	**modify/write**
mechanism:	**by reference**

An SQLDA structure used for select list items. If the value passed by the caller is the address of an existing SQLDA structure, the API writes metadata into that structure. If the SQL statement is a SELECT, there must be at least one SQLVAR structure for each select list item.

If the value passed by the caller is NULL, the API determines whether an SQLDA structure is needed. If an SQLDA is needed, the API performs the following operations; otherwise it leaves the value NULL:

- Dynamically allocates an SQLDA structure containing the requisite number of SQLVAR structures
- Writes select list metadata into the SQLDA
- Returns the address of the SQLDA

Description

In an SQLDA structure returned by the sqlsrv_prepare routine, the SQLVARARY[ ].SQLDATA (address of data variable) and SQLVARARY[ ].SQLIND (address of indicator variable) fields are NULL. Before calling the sqlsrv_execute routine, your application must allocate data and indicator variables and must write the addresses of those variables into SQLVARARY[ ].SQLDATA and SQLVARARY[ ].SQLIND, respectively.

Your application can perform those functions itself, or can call the sqlsrv_allocate_sqlda_data routine to dynamically allocate the variables and to write the addresses into the SQLDA.

Typically, an application that finishes processing one SQL statement before preparing the next SQL statement would use the sqlsrv_prepare routine to allocate SQLDA structures and the sqlsrv_allocate_sqlda_data routine to allocate data and indicator variables. An application that prepares more than one SQL statement at a time and thus must use several different SQLDA structures at the same time, can allocate as many as required and pass them to the sqlsrv_prepare routine. Note, however, that you cannot use the sqlsrv_release_statement or sqlsrv_free_sqlda_data routines to free memory explicitly allocated by your application.

Notes

You must supply valid values for the parameter_marker_sqlda and select_list_sqlda parameters. If the SQL statement is known not to contain parameter markers or not to be a SELECT statement, supply NULL values.

Errors

SQLSRV_INTERR	Internal error.
SQLSRV_INVARG	Invalid routine parameter.
SQLSRV_INVASC	Invalid association identifier.
SQLSRV_INVSQLDA	Invalid SQLDA structure.
SQLSRV_MULTI_ACT	A batched sqlsrv_execute or sqlsrv_fetch_many context is active.
SQLSRV_NETERR	DECnet returned an error.
SQLSRV_NO_MEM	API memory allocation failed.

SQL Errors

SQL_RDBERR	Rdb/VMS returned an error.

6.14 sqlsrv_release—Release Client/Server Association

The sqlsrv_release routine commits active transactions on the server and requests an orderly termination of the association, which disconnects the network link and frees client association resources.

VAX Format

SQLSRV$RELEASE associate_id [,stats]

C Format

```
extern int sqlsrv_release(
                char *associate_id,
                char *stats);
```

Parameters

associate_id

data type:	**undefined**
access:	**read**
mechanism:	**by reference**

Handle used to identify the active association.

stats (optional)

data type:	**undefined**
access:	**modify**
mechanism:	**by reference**

This parameter must be 0 (a null pointer).

Errors

SQLSRV_INTERR	Internal error.
SQLSRV_INVASC	Invalid association identifier.
SQLSRV_MULTI_ACT	A batched sqlsrv_execute or sqlsrv_fetch_many context is active.
SQLSRV_NETERR	DECnet returned an error.

6.15 sqlsrv_release_statement—Release Statement Resources

The sqlsrv_release_statement routine frees all resources associated with one or more prepared statements (including dynamically allocated SQLDA structures) for both the client and server, and updates SQLERR[2] with the number of statements that were released. Processing stops when an error is encountered.

VAX Format

SQLSRV$RELEASE_STATEMENT associate_id ,statement_id_count ,statement_id_array

C Format

```
extern int sqlsrv_release_statement(
                char *associate_id,
                short int statement_id_count,
                long int *statement_id_array);
```

Parameters

associate_id

data type:	**undefined**
access:	**read**
mechanism:	**by reference**

Handle used to identify the active association.

statement_id_count

data type:	**word (signed)**
access:	**read**
mechanism:	**by value**

The number of statement identifiers passed in the statement_id_array.

sqlsrv_release_statement—Release Statement Resources

statement_id_array

data type:	**longword (signed) array**
access:	**read**
mechanism:	**by reference**

An array containing the identifiers (statement_id parameters returned by the sqlsrv_prepare routine) of the statements to free.

Errors

SQLSRV_INTERR	Internal error.
SQLSRV_INVARG	Invalid routine parameter.
SQLSRV_INVASC	Invalid association identifier.
SQLSRV_INVSTMID	Invalid statement identifier.
SQLSRV_MULTI_ACT	A batched sqlsrv_execute or sqlsrv_fetch_many context is active.
SQLSRV_NETERR	DECnet returned an error.

6.16 sqlsrv_set_environment—Set Environment Variable Values

The sqlsrv_set_environment routine sets the values of environment variables (as described in Section 5.2).

VAX Format

SQLSRV$SET_ENVIRONMENT associate_id ,env_str_array_count ,env_str_array

C Format

```
extern int sqlsrv_set_environment(
                    char *associate_id,
                    unsigned short env_str_array_count,
                    struct SQLSRV_ENV_STR *env_str_array);
```

Parameters

associate_id

data type:	**undefined**
access:	**read**
mechanism:	**by reference**

Handle used to identify the active association.

env_str_array_count

data type:	**word (unsigned)**
access:	**read**
mechanism:	**by value**

The number of elements in the env_str_array.

env_str_array

data type:	**longword (unsigned)**
access:	**read**
mechanism:	**by reference**

An array of SQLSRV_ENV_STR structures (described in Section 7.7), each of which contains the information necessary to set an environment variable.

sqlsrv_set_environment—Set Environment Variable Values

Description

Your application allocates an array of SQLSRV_ENV_STR structures, each of which describes an environment variable, and sets the values of the ENV_TAG and ENV_VALUE fields.

Errors

SQLSRV_INTERR	Internal error.
SQLSRV_INVARG	Invalid routine parameter.
SQLSRV_INVASC	Invalid association identifier.
SQLSRV_INVENVTAG	Invalid environment tag.
SQLSRV_INVENVVAR	Invalid environment variable.
SQLSRV_MULTI_ACT	A batched sqlsrv_execute or sqlsrv_fetch_many context is active.
SQLSRV_NETERR	DECnet returned an error.

6.17 sqlsrv_set_filter—Define Filter for Result Table

The sqlsrv_set_filter routine defines a Boolean filter expression (as described in Section 4.4) and associates the expression with a result table. When your application calls sqlsrv_fetch, the server applies the specified filter to each row and eliminates from the result table those rows for which the expression returns a value of false.

VAX Format

SQLSRV$SET_FILTER associate_id ,cursor_name ,filter_expression ,sqlda_index_count ,sqlda_index_array ,filter_precedence

C Format

```
extern int sqlsrv_set_filter(
                char *associate_id,
                char *cursor_name,
                char *filter_expression,
                short int sqlda_index_count,
                short int *sqlda_index_array,
                short int filter_precedence);
```

Parameters

associate_id

data type:	**undefined**
access:	**read**
mechanism:	**by reference**

Handle used to identify the active association.

cursor_name

data type:	**character string**
access:	**read**
mechanism:	**by reference**

A null-terminated string used to identify the open cursor.

sqlsrv_set_filter—Define Filter for Result Table

filter_expression

data type:	**character string**
access:	**read**
mechanism:	**by reference**

A null-terminated string containing the filter expression applied to the result table by the server when your application fetches a row.

sqlda_index_count

data type:	**word (signed)**
access:	**read**
mechanism:	**by value**

The number of "?" placeholders in the filter expression.

sqlda_index_array

data type:	**word (signed) array**
access:	**read**
mechanism:	**by value**

An array of zero-based indices into the select list SQLDA structure associated with cursor_name. The first array element corresponds to the first "?" placeholder in the filter expression, and so forth.

filter_precedence

data type:	**word (signed)**
access:	**read**
mechanism:	**by value**

This parameter must be 0.

Notes

- You can associate only one filter expression with a cursor.
- You can use environment variables to control the way that dates in filter expressions are parsed (see sqlsrv_set_environment).

Errors

SQLSRV_FTRSYNERR	Syntax error in filter expression.
SQLSRV_INTERR	Internal error.
SQLSRV_INVARG	Invalid routine parameter.
SQLSRV_INVASC	Invalid association identifier.
SQLSRV_INVCURNAM	Invalid cursor name.
SQLSRV_INVIDX	Invalid sqlda_index_array.
SQLSRV_MULTI_ACT	A batched sqlsrv_execute or sqlsrv_fetch_many context is active.
SQLSRV_NETERR	DECnet returned an error.

7
Data Structures

This chapter describes the data structures that SQL/Services uses to communicate with the client application. Some of the data structures (the SQLDA and SQLCA) are identical in allocation but not in usage with those in dynamic SQL. Those structures are described in detail in the *VAX Rdb/VMS SQL Reference Manual*. This manual provides relatively brief descriptions and points out the differences in usage.

7.1 Documentation Format

Each SQL/Services data structure is documented using a structured format called a template. The sections of the template are shown in Table 7–1, along with the information that is presented in each section and the format used to present the information.

Table 7–1 Sections in the Data Structure Template

Section	Description
Structure Name	Appears at the top of the page, followed by the English equivalent.
Overview	Appears directly below the structure name. The overview explains, usually in one or two sentences, the purpose of the structure.
Diagram	Shows the layout of the structure on a 32-bit machine architecture.
Fields	Gives detailed information about each field.

The Fields section contains detailed information about each field in the data structure. Fields are described in the order in which they appear in the structure.

Documentation Format

The following format is used to describe each field:

field-name

data type: the data type of the specific field (see Table 6–3)

C declaration: how that field is declared in the SQL/Services include files

set by: whether the value of the field is set by the API, the application program, or both

used by: whether the value of the field is used by the API, the application program, or both

In addition, the Fields section contains at least one paragraph of text describing the purpose of the field.

7.2 ASSOCIATE_STR—Association Structure

The association structure is a parameter that is passed to the sqlsrv_associate routine to enable or disable API functions such as execution logging, user-defined memory allocation, local input/output, and alternative error message buffering. The ASSOCIATE_STR is defined in the include file SQLSRV.H.

SERVER_LOG	CLIENT_LOG	0
VERSION	LOCAL_FLAG	0
MEMORY_ROUTINE		0
FREE_MEMORY_ROUTINE		0
ERRBUFLEN	RESERVED	0
ERRBUF		0

Fields

CLIENT_LOG
data type: **word (unsigned)**
C declaration: **unsigned short int CLIENT_LOG**
set by: **program**
used by: **API**

Specifies the type of execution logging to be enabled or disabled on the client system (see Section 4.5). The following constants are defined in the include file SQLSRV.H:

SQLSRV_LOG_DISABLED	Disables logging (default)
SQLSRV_LOG_ASSOCIATION	Enables association logging
SQLSRV_LOG_ROUTINE	Enables API routine logging
SQLSRV_LOG_PROTOCOL	Enables message protocol logging
SQLSRV_LOG_SCREEN	Sends logging output to the video display on the client system as well as to the log file

To enable more than one type of logging, add the appropriate constants.

SERVER_LOG

data type:	**word (unsigned)**
C declaration:	**unsigned short int SERVER_LOG**
set by:	**program**
used by:	**API**

Enables or disables message protocol logging on the server system (see Section 4.5). The following constants are defined in the include file SQLSRV.H:

SQLSRV_LOG_DISABLED	Disables logging (default)
SQLSRV_LOG_PROTOCOL	Enables message protocol logging

LOCAL_FLAG

data type:	**word (signed)**
C declaration:	**short int LOCAL_FLAG**
set by:	**program**
used by:	**API**

Specifies whether SQL/Services can use local input/output instead of DECnet input/output in the association and subsequent messages. Local input/output is valid (and preferred) only when the server is on the same VAX system as the application. However, a process can have only one local association at a time. The user name and password parameters to the sqlsrv_associate call are ignored; those associated with the current process are used instead.

0	DECnet input/output (default)
1	local input/output

VERSION

data type:	**word (signed)**
C declaration:	**short int VERSION**
set by:	**reserved**
used by:	**unused**

Must be 0.

MEMORY_ROUTINE

data type:	**pointer**
C declaration:	**char *(*MEMORY_ROUTINE) ()**
set by:	**program**
used by:	**API**

A pointer to the entry point of a user-specified routine to be called by the API for memory allocation. This feature is for client environments in which a limited amount of memory is available. The default value is NULL, which causes the API to use the portable C routine *malloc()* for all memory allocation.

FREE_MEMORY_ROUTINE

data type:	**pointer**
C declaration:	**char (*FREE_MEMORY_ROUTINE) ()**
set by:	**program**
used by:	**API**

A pointer to the entry point of a user-specified routine to be called by the API for memory deallocation. The default value is NULL, which causes the API to use the portable C routine *free()* for all memory deallocation.

RESERVED

data type:	**word (signed)**
C declaration:	**short int RESERVED**
set by:	**program**
used by:	**unused**

This field is reserved.

ERRBUFLEN

data type:	**word (signed)**
C declaration:	**short int ERRBUFLEN**
set by:	**program**
used by:	**API**

The length in bytes of ERRBUF. The recommended length is 512 bytes if sufficient memory is available.

ERRBUF

data type:	**pointer**
C declaration:	**char *ERRBUF**
set by:	**API**
used by:	**program**

The address of a buffer in which to store ASCII error messages from SQL/Services, SQL, Rdb, or VMS. If you supply a valid address, the API writes error messages into this buffer instead of the SQLCA.SQLERRM.SQLERRMC buffer, which is only 70 bytes long and may be too small to contain the entire message. If you supply a NULL value, the API writes error messages into the SQLCA.SQLERRM.SQLERRMC buffer.

7.3 SQLCA—SQL Communications Area

The SQLCA structure is used to store information when an error occurs. This structure is defined in the include file SQLSRVCA.H along with the error codes generated by SQL/Services.

SQLCAID[3] "C"	SQLCAID[2] "L"	SQLCAID[1] "Q"	SQLCAID[0] "S"	0
SQLCAID[7] res	SQLCAID[6] res	SQLCAID[5] 0	SQLCAID[4] "A"	0
SQLCABC				0
SQLCODE				0
		SQLERRM.SQLERRML		0
SQLERRM.SQLERRMC[ ] (70 bytes)				
				0
SQLERRD[0]				0
SQLERRD[1]				0
SQLERRD[2]				0
SQLERRD[3]				0
SQLERRD[4]				0
SQLERRD[5]				0
SQLWARN3	SQLWARN2	SQLWARN1	SQLWARN0	0
SQLWARN7	SQLWARN6	SQLWARN5	SQLWARN4	4
SQLEXT[3]	SQLEXT[2]	SQLEXT[1]	SQLEXT[0]	0
SQLEXT[7]	SQLEXT[6]	SQLEXT[5]	SQLEXT[4]	0

SQLCA—SQL Communications Area

The SQL/Services SQLCA is based on the SQL SQLCA, which is described in detail in the *VAX Rdb/VMS SQL Reference Manual.*

Fields

SQLCAID

data type:	**character string**
C declaration:	**char SQLCAID (8)**
set by:	**API**
used by:	**unused**

Structure identification field, present only for compatibility with SQL. Contains the null-terminated string “SQLCA” followed by two reserved bytes.

SQLCABC

data type:	**longword (signed)**
C declaration:	**long int SQLCABC**
set by:	**API**
used by:	**program**

Contains the size, in bytes, of the SQLCA structure. The value of this field is always 128.

SQLCODE

data type:	**longword (signed)**
C declaration:	**long int SQLCODE**
set by:	**API**
used by:	**program**

Contains the error status for the most recently invoked SQL/Services routine. A positive value indicates a warning, a negative value indicates an error, and a 0 value indicates success. The include file SQLSRVCA.H contains the error messages that correspond to all of the possible values of SQLCODE. The file SQLSRV$MSG.DOC contains explanations of the errors and suggests user actions.

SQLERRM.SQLERRML

data type:	**word (signed)**
C declaration:	**short int SQLERRML**
set by:	**API**
used by:	**program**

The length, in bytes, of the error message text returned in SQLERRMC.

SQLERRM.SQLERRMC

data type:	**character string**
C declaration:	**char SQLERRMC (70)**
set by:	**API**
used by:	**program**

An ASCII string that describes the error (which may be from SQL/Services, SQL, Rdb, or VMS) in more detail. Because some error messages are longer than 70 bytes, you can use the ASSOCIATE_STR.ERRBUF field to define a longer buffer (see Section 7.2).

SQLERRD

data type:	**array of longword (signed)**
C declaration:	**long int SQLERRD (6)**
set by:	**API**
used by:	**program**

An array of six integers as described in Section 7.4.

SQLWARNn

data type:	**character**
C declaration:	**char SQLWARN0 . . . SQLWARN7**
set by:	**unused**
used by:	**unused**

A series of eight 1-character fields that SQL and the API do not use.

SQLEXT

data type:	**character string**
C declaration:	**char SQLEXT (8)**
set by:	**unused**
used by:	**unused**

Not used by the API.

7.4 SQLERRD—Part of SQLCA

The SQLERRD array contains six elements. SQL/Services uses only the first three elements.

SQLERRD Elements

SQLERRD[0]

Contains the detailed error code when the SQLCODE field is SQLSRV_NETERR or SQLSRV_SRVERR, as defined in the include file SQLSRVCA.H. Information about these error codes can be found at the locations listed in Table 7–2.

Table 7–2 Error Code Files

Operating System	File Specification	Description
VMS	SYS$LIBRARY:SSDEF.H	System service return status code definitions
MS-DOS	DERRNO.H	DECnet error codes (provided with the DECnet-DOS software)
ULTRIX	/usr/include/errno.h	DECnet error codes (provided with the DECnet-ULTRIX software)

Note *This feature is an extension to Rdb / VMS SQL and ANSI SQL.*

SQLERRD[1]

The number of rows processed successfully in a batched execution.

Note *This feature is an extension to Rdb / VMS SQL and ANSI SQL.*

SQLERRD[2]

The value placed in the SQLERRD[2] field depends on the type of SQL statement executed, as shown in Table 7–3.

SQLERRD—Part of SQLCA

Table 7–3 Values Placed in the SQLCA.SQLERRD(2) Field

SQL Statement	API Routine	Value
INSERT[1]	sqlsrv_execute or sqlsrv_execute_immediate	The number of rows stored.
UPDATE[1]	sqlsrv_execute or sqlsrv_execute_immediate	The number of rows modified.
DELETE[1]	sqlsrv_execute or sqlsrv_execute_immediate	The number of rows deleted.
FETCH	sqlsrv_fetch	The number of the row on which the cursor is currently positioned. This is maintained within a sqlsrv_fetch_many context with the restriction that positional SQL statements cannot be invoked.
OPEN	sql_open_cursor	0
RELEASE	sqlsrv_release_statement	The number of statements released.
n/a	sqlsrv_get_environment	The number of environment variable values returned in sql_str_array.

[1] For INSERT, UPDATE, and DELETE statements that operate on multiple rows of data ("batched" execution), the value of SQLDERR[2] reflects the total number of rows modified.

7.5 SQLDA—SQL Descriptor Area

The SQLDA structure contains SQL parameter marker and select list metadata as well as pointers to data and indicator variables. It is defined in the include file SQLSRVDA.H.

The SQL/Services SQLDA is identical to the SQLDA structure in SQL. For additional information on the SQLDA, read the dynamic SQL chapter in the *VAX Rdb/VMS Guide to Using SQL* and the SQLDA appendix in the *VAX Rdb/VMS SQL Reference Manual.*

SQLDAID[3] "D"	SQLDAID[2] "L"	SQLDAID[1] "Q"	SQLDAID[0] "S"	0
SQLDAID[7] res	SQLDAID[6] res	SQLDAID[5] 0	SQLDAID[4] "A"	0
SQLABC				0
SQLD		SQLN		0
≈ SQLVARARY[0. . .n] (44 bytes) ≈				44

Fields

SQLDAID
data type: **character string**
C declaration: **char SQLDAID(8)**
set by: **API**
used by: **unused**

Structure identification field, present only for compatibility with dynamic SQL. Contains the null-terminated string "SQLDA" followed by two reserved bytes.

SQLABC
data type: **longword (signed)**
C declaration: **long int SQLABC**
set by: **API**
used by: **unused**

The size, in bytes, of the SQLDA structure.

SQLN

data type:	**word (signed)**
C declaration:	**short int SQLN**
set by:	**see following text**
used by:	**API**

The number of elements in the SQLVARARY. If the API allocated the SQLDA structure, this value is the same as the SQLD field. If your application allocated its own SQLDA structure, it must supply this value. In that case, the SQLN field specifies the maximum number of select list items or parameter marker items that can exist in an SQL statement that is prepared with a particular SQLDA; a call to the sqlsrv_prepare routine with an SQLVARARY that is too small returns an error.

SQLD

data type:	**word (signed)**
C declaration:	**short int SQLD**
set by:	**API**
used by:	**program**

The number of parameter markers or select list items in a prepared SQL statement. In an SQLDA structure that was allocated by the API, this value is the same as the SQLN field (the number of elements in the SQLVARARY).

SQLVARARY

data type:	**array of structures**
C declaration:	**struct SQLVAR SQLVARARY(1)**
set by:	**see Section 7.6**
used by:	**see Section 7.6**

An array of SQLVAR structures (see Section 7.6), each of which describes one select list item or one parameter marker item.

7.6 SQLVAR—Parameter Marker or Select List Item

Each SQLVAR structure describes one select list item or parameter marker.

SQLLEN		SQLTYPE		0
SQLDATA				0
SQLIND				0
SQLNAME[1]	SQLNAME[0]	SQLNAME_LEN		0
SQLNAME[5]	SQLNAME[4]	SQLNAME[3]	SQLNAME[2]	0
SQLNAME[9]	SQLNAME[8]	SQLNAME[7]	SQLNAME[6]	0
SQLNAME[13]	SQLNAME[12]	SQLNAME[11]	SQLNAME[10]	0
SQLNAME[17]	SQLNAME[16]	SQLNAME[15]	SQLNAME[14]	0
SQLNAME[21]	SQLNAME[20]	SQLNAME[19]	SQLNAME[18]	0
SQLNAME[25]	SQLNAME[24]	SQLNAME[23]	SQLNAME[22]	0
SQLNAME[29]	SQLNAME[28]	SQLNAME[27]	SQLNAME[26]	0

Fields

SQLTYPE

data type:	**word (signed)**
C declaration:	**short int SQLTYPE**
set by:	**API**
used by:	**program**

The SQL data type for the SQLVAR entry. This value represents the SQL/Services data type as defined in the include file SQLSRVDA.H.

```
#define SQLSRV_ASCII_STRING                     129
#define SQLSRV_GENERALIZED_NUMBER               130
#define SQLSRV_GENERALIZED_DATE                 131
#define SQLSRV_VARCHAR                          132
```

SQLLEN

data type:	**word (signed)**
C declaration:	**short int SQLLEN**
set by:	**see following text**
used by:	**program**

For SQLSRV_ASCII_STRING, SQLSRV_GENERALIZED_DATE, and SQLSRV_VARCHAR data, the length, in bytes, of the variable pointed to by the SQLDATA field.

For SQLSRV_GENERALIZED_NUMBER, the SQLLEN field is split in half. The low-order byte of SQLLEN indicates the size of the data variable. The high-order byte indicates the **scale factor** (the number of digits to the right of the decimal point). Thus, a scale factor of 0 indicates that the value is either an integer or a floating-point number in E notation. A non-zero scale factor indicates that the value is a decimal number.

SQLDATA

data type:	**pointer**
C declaration:	**char *SQLDATA**
set by:	**program or API**
used by:	**program and API**

The address of a variable used to store data (select list items or parameter markers). If your application allocates data variables by calling the sqlsrv_allocate_sqlda_data routine, the API initializes this field. If your application allocates its own data variables, it must write the address of each variable into an SQLDATA field. In that case, the API returns an error if an SQLLEN value is less than the length of the associated data value.

SQLIND

data type:	**pointer**
C declaration:	**short int *SQLIND**
set by:	**program or API**
used by:	**program and API**

The address of an indicator variable for the data. (A value of –1 in the indicator variable indicates a null data value.) If your application calls the sqlsrv_allocate_sqlda_data routine, the API initializes this field. Otherwise, your application must allocate its own indicator variables and write the address of each variable into an SQLIND field.

SQLNAME_LEN

data type:	**word (signed)**
C declaration:	**short int SQLNAME_LEN**
set by:	**API**
used by:	**program**

The length, in bytes, of the name stored in the SQLNAME field.

SQLNAME

data type:	**character string**
C declaration:	**char SQLNAME(30)**
set by:	**API**
used by:	**program**

The column name of the select list or parameter marker entry. The maximum length of a column name is 30 characters. If the actual name is less than 30 characters, the API returns a null-terminated string.

7.7 SQLSRV_ENV_STR—Environment Variable Structure

The SQLSRV_ENV_STR structure contains the value of an environment variable, as described in Section 5.2. Your application passes an array of SQLSRV_ENV_STR structures to the sqlsrv_set_environment and sqlsrv_get_environment routines.

The SQLSRV_ENV_STR, environment variable names, and environment variable settings are defined in the include file SQLSRV.H. The abbreviation "env" is used in the include file for convenience.

ENV_RESERVED	ENV_TAG	0
ENV_VALUE		0
ENV_OPT_VALUE		0

Fields

ENV_TAG

data type:	**word (unsigned)**
C declaration:	**unsigned short int ENV_TAG**
set by:	**program**
used by:	**API**

Identifies the environment variable to be set or returned (SQLSRV_ENV_DATE or SQLSRV_ENV_CENTURY).

ENV_RESERVED

data type:	**word (signed)**
C declaration:	**short int ENV_RESERVED**
set by:	**program**
used by:	**unused**

This field is reserved (must be 0).

ENV_VALUE

data type:	**longword (signed)**
C declaration:	**long int ENV_VALUE**
set by:	**program or API**
used by:	**API or program**

The value of the environment variable. For SQLSRV_ENV_DATE, see Table 5–2. For SQLSRV_ENV_CENTURY, see Table 5–3.

ENV_OPT_VALUE

data type:	**pointer**
C declaration:	**char *ENV_OPT_VALUE**
set by:	**unused**
used by:	**unused**

This field is reserved.

A

Filter Expression Functions

This appendix describes the functions that can be used to evaluate or convert data in filter expressions.

As described in Section 6.17, SQL/Services applications can call the sqlsrv_set_filter routine to associate Boolean filter expressions with result tables. When your application calls sqlsrv_fetch, the API applies the specified filter to each row and eliminates from the result table those rows for which the expression returns FALSE.

The conventions used in Appendix A are:

<>	Angle brackets indicate that you supply a data value of the type required for the item in the brackets
<expC>	Angle brackets enclosing expC indicate character data.
<expN>	Angle brackets enclosing expN indicate numeric data.
<expD>	Angle brackets enclosing expD indicate date type data.
[ ]	Brackets enclose optional items
/	The slash indicates an either/or choice

A.1 ABS

The ABS function returns the absolute value of a numeric expression. The returned value is always a positive number.

Syntax

```
ABS(<expN>)
```

Examples

The following expression returns the difference between two numbers without regard to their sign (0).

```
ABS(3) + ABS(-3)
```

The following expression returns the number of days between two dates (268).

```
ABS(CTOD("12/25/88") - CTOD("04/01/88"))
```

A.2 ACOS

The ACOS arccosine function calculates and returns the angle size in radians for any given cosine value.

Syntax

ACOS(<expN>)

Arguments

<expN>
A numeric expression that is the cosine of a particular angle. The value of the numeric expression must be between –1.0 and +1.0 inclusive.

Usage

The response is always a number that represents an angle size in radians between zero and pi (π).

Examples

The following expression returns 0.7854.

```
ACOS(0.7071)
```

See Also

ASIN	Section A.4
ATAN	Section A.6
ATN2	Section A.7
COS	Section A.12
DTOR	Section A.19
PI	Section A.33
RTOD	Section A.38
SIN	Section A.41
TAN	Section A.48

A.3 ASC

The ASC function returns the ASCII decimal code of the first character from a character expression.

Syntax

ASC(<expC>)

Examples

The following expression returns 78.

```
ASC("Nestle")
```

A.4 ASIN

The ASIN arcsine function calculates and returns the angle size (in radians) for any given sine value.

Syntax

ASIN(<expN>)

Arguments

<expN>
A numeric expression that is the sine of a particular angle. The value of the numeric expression must be between –1.0 and +1.0 inclusive.

Usage

The value returned is always a floating-point number that represents an angle size (in radians) between $-\pi/2$ and $+\pi/2$.

Examples

The following expression returns .5236.

```
ASIN(.5000)
```

See Also

ACOS	Section A.2
ATAN	Section A.6
ATN2	Section A.7
COS	Section A.12
DTOR	Section A.19
RTOD	Section A.38
SIN	Section A.41
TAN	Section A.48

A.5 AT

The AT function returns a number that shows the starting position of a character string within a second string, counting from 1.

Syntax

AT(<expC>,<expC>)

Usage

The contained character string is called a substring. If the substring is not contained within the second expression, the function returns a zero.

Examples

The following expression returns 4.

```
AT("b", "aaabaaa")
```

See Also

LEFT	Section A.24
RIGHT	Section A.36
SUBSTR	Section A.47

A.6 ATAN

The ATAN arctangent function calculates and returns the angle size (in radians) for any given tangent value.

Syntax

```
ATAN(<expN>)
```

Arguments

<expN>
A numeric expression that is the tangent of a particular angle. The range is between +π/2 and −π/2.

Examples

The following expression returns 0.7854.

```
ATAN(1.000)
```

See Also

ACOS	Section A.2
ASIN	Section A.4
ATN2	Section A.7
COS	Section A.12
DTOR	Section A.19
RTOD	Section A.38
SIN	Section A.41
TAN	Section A.48

A.7 ATN2

The ATN2 arctangent function calculates and returns the angle size (in radians) when the cosine and sine of a given point are specified.

Syntax

ATN2(<expN1>,<expN2>)

Arguments

<expN1>
The sine of a particular angle

<expN2>
The cosine of that same angle

Usage

The value of the expression <expN1>/<expN2> must fall within the range of +π and −π.

This function returns values in all four quadrants, and is equivalent to ATAN(x/y). It is easier to use than ATAN(x/y) because it eliminates divide-by-zero errors.

The returned value is always a number that represents an angle size (in radians) between +π and −π.

Examples

The following expression, which shows an integrated usage of trigonometric functions, returns 30.00.

```
RTOD(ATN2(SIN(DTOR(30)),COS(DTOR(30))))
```

See Also

ATAN	Section A.6
COS	Section A.12
DTOR	Section A.19
RTOD	Section A.38
SIN	Section A.41
TAN	Section A.48

A.8 CDOW

The CDOW function returns the name of the day of the week from a date expression.

Syntax

CDOW(<expD>)

Arguments

<expD>
A placeholder or any function that returns date type data

Examples

The following expression returns “Monday”.

```
CDOW({02/29/88})
```

See Also

CTOD	Section A.13
DAY	Section A.14
DOW	Section A.17
DTOC	Section A.18

A.9 CEILING

The CEILING function calculates and returns the smallest integer that is greater than or equal to the value specified in the numeric expression.

Syntax

CEILING(<expN>)

Usage

Use this function to find the smallest integer that is greater than or equal to a given value. The value returned is the same data type as the specified numeric expression.

Examples

The following expression returns 13.00.

```
CEILING(12.3)
```

The following expression returns –5.00. Unlike ROUND, CEILING always returns an integer closer to zero. ROUND(–5.556,0) returns –6.00.

```
CEILING(-5.556)
```

See Also

FLOOR	Section A.22
ROUND	Section A.37

A.10 CHR

The CHR function converts an ASCII decimal code to a character.

Syntax

CHR(<expN>)

Arguments

<expN>
An integer numeric expression in the range 1 to 255

Examples

The following expression returns capital A.

```
CHR(65)
```

The following expression returns false.

```
CHR(0) = "abc"
```

The following expression returns true. When you use CHR in comparisons, CHR(0) must be on the left side of the equation. When the expression evaluator performs character string comparisons, it reads what is on the right side first. Because CHR(0) is a null string, if it is on the right side, the evaluator reads no further and returns a value of true.

```
"abc" = CHR(0)
```

See Also

ASC	Section A.3

A.11 CMONTH

The CMONTH function returns the name of the month from a date expression.

Syntax

CMONTH(<expD>)

Arguments

<expD>
A placeholder or any function that returns date type data

Examples

The following expression returns "May".

```
CMONTH({05/15/88})
```

See Also

MONTH	Section A.32

A.12 COS

The cosine COS function calculates and returns the cosine value for any angle size expressed in radians.

Syntax

```
COS(<expN>)
```

Arguments

<expN>
A numeric expression that is the size of an angle measured (in radians). There are no limits on this numeric expression.

Examples

The following expression returns 0.7071.

```
COS(.7854)
```

See Also

ACOS	Section A.2
ASIN	Section A.4
ATAN	Section A.6
ATN2	Section A.7
DTOR	Section A.19
RTOD	Section A.38
SIN	Section A.41
TAN	Section A.48

A.13 CTOD

The CTOD function converts a date stored as a character string to date type data.

Syntax

CTOD(<expC>)

{expC}

Arguments

<expC>
The format of the character string is normally mm/dd/yy, but this format can be changed by the environment variables SQLSRV_ENV_DATE and SQLSRV_ENV_CENTURY (see Section 5.2).

Usage

The character expression used by CTOD can range from “01/01/0100” to “12/31/9999”. A twentieth century date is assumed if you use only two numbers for the year.

You can also use braces {mm/dd/yy} to create a date type data from a literal value.

See Also

DTOC	Section A.18
DTOS	Section A.20

A.14 DAY

The DAY function returns the numeric value of the day of the month from a date expression.

Syntax

DAY(<expD>)

Arguments

<expD>
A placeholder or any function that returns date type data

Examples

The following expression returns 15.

```
DAY({05/15/88})
```

See Also

CDOW	Section A.8
DOW	Section A.17

A.15 DIFFERENCE

The DIFFERENCE function converts two literal strings to SOUNDEX codes and returns a value representing the difference between the two strings.

Syntax

DIFFERENCE(<expC>, <expC>)

Arguments

<expC>
Must be a character expression. Placeholders can be used.

Usage

The DIFFERENCE function returns an integer between 0 and 4. Two closely matched codes return a difference of 4, and two codes that have no letters in common return a code of 0. One common letter in each string returns a 1.

Examples

To find names with similar SOUNDEX codes:

The following expression returns 3.

```
DIFFERENCE("Sandra","Kimbrelee")
```

The following expression returns 4.

```
DIFFERENCE("Kimberly","Kimbrelee")
```

See Also

SOUNDEX	Section A.42

A.16 DMY

The DMY function converts the date to a day/month/year format from any valid date expression.

Syntax

DMY(<expD>)

Arguments

<expD>
A placeholder or any function that returns date type data

Usage

This function converts the date to the following format:

DD Month YY

The day is shown without a leading zero as one or two digits. The month is spelled in full, and the year is shown with the two last digits.

If the environment variable SQLSRV_ENV_CENTURY (see Section 5.2) is ON, the format is:

DD Month YYYY

Examples

The following expression returns “29 February 88”.

```
DMY({02/29/88})
```

See Also

CDOW	Section A.8
CMONTH	Section A.11
DOW	Section A.17
MDY	Section A.30
MONTH	Section A.32
YEAR	Section A.54

A.17 DOW

The DOW function returns a number that represents the day of the week from a date expression, starting with Sunday as day 1.

Syntax

DOW(<expD>)

Arguments

<expD>
A placeholder, or any function that returns date type data

Examples

The following expression returns 6.00.

```
DOW({05/13/88})
```

See Also

CDOW	Section A.8
DAY	Section A.14

A.18 DTOC

The DTOC function converts a date expression to a character string.

Syntax

DTOC(<expD>)

Usage

This function is used to store a date as character data or to compare a date to a character string.

Examples

The following expression returns “05/13/88”.

```
DTOC({05/13/88})
```

See Also

CTOD	Section A.13

A.19 DTOR

The DTOR function converts degrees to radians.

Syntax

DTOR(<expN>)

Arguments

<expN>
The size of the angle measured in degrees

Usage

The DTOR function returns the angle size (in radians).

Convert minutes and seconds to decimal fractions of a degree before using this function.

Examples

The following expression returns 3.14.

```
DTOR(180)
```

See Also

ACOS	Section A.2
ATAN	Section A.6
ATN2	Section A.7
COS	Section A.12
RTOD	Section A.38
SIN	Section A.41

A.20 DTOS

The DTOS function converts a date expression to a character string of the form CCYYMMDD regardless of SQLSRV_ENV_CENTURY or SQLSRV_ENV_DATE.

Syntax

DTOS(<expD>)

Usage

Use this function when you need a date expression in a character string that has the same format regardless of environment variables.

Examples

The following expression returns “19880229”.

```
DTOS({02/29/88})
```

See Also

CTOD	Section A.13
DTOC	Section A.18

A.21 EXP

The EXP function returns the value that results from raising the constant e to the power of <expN>.

Syntax

EXP(<expN>)

Usage

Given the equation $y = e^x$, <expN> is the value of x. For any exponent x to the base e, the function returns the value of y from the equation. The returned value is a real number.

Examples

The following expression returns 625.00.

```
EXP(LOG(25) + LOG(25))
```

See Also

LOG	Section A.26

A.22 FLOOR

The FLOOR function calculates and returns the largest integer that is less than or equal to the value of the specified numeric expression. The returned value is the same data type as the argument.

Syntax

```
FLOOR(<expN>)
```

Examples

The following expression returns 12.00.

```
FLOOR(12.99)
```

See Also

CEILING	Section A.9
INT	Section A.23
ROUND	Section A.37

A.23 INT

The INT function truncates any numeric expression to an integer.

Syntax

INT(<expN>)

Usage

You can discard all digits to the right of the decimal point in a numeric expression by using INT.

Examples

The following expression returns 10.

```
INT(10.23)
```

See Also

CEILING	Section A.9
FLOOR	Section A.22
ROUND	Section A.37

A.24 LEFT

The LEFT function returns a specified number of characters from a character expression, starting from the first character on the left.

Syntax

LEFT(<expC>,<expN>)

Usage

The LEFT function lets you retrieve the first part of a character string. This is the same as defining the SUBSTR function with a starting position of one, and the number of characters to extract with <expN>.

The numeric expression defines the number of characters to extract from the character string. If the numeric expression is zero, a null string is returned.

If the numeric expression is greater than the length of the character string, LEFT returns the entire string.

Examples

The following expression returns "abc".

```
LEFT("abcdef",3)
```

See Also

AT	Section A.5
LTRIM	Section A.29
RIGHT	Section A.36
RTRIM	Section A.39
STUFF	Section A.46
SUBSTR	Section A.47
TRIM	Section A.50

A.25 LEN

The LEN function returns a numeric value indicating the number of characters in a specified character expression.

Syntax

LEN(<expC>)

Usage

Use this function to determine the number of characters in a placeholder. This function returns a zero if the associated data variable contains a null string.

Examples

The following expression returns 6.

```
LEN("Bailey")
```

See Also

TRIM	Section A.50

A.26 LOG

The LOG function returns the natural logarithm of a specified number.

Syntax

LOG(<expN>)

Usage

The natural logarithm has a base of e. The LOG function returns the exponent in the equation $y = e^x$ where x is the numeric expression used by the LOG function. This must be a positive integer for the value of <expN>. LOG returns the value of y.

Examples

The following expression returns 1.00000.

```
LOG(2.71828)
```

See Also

EXP	Section A.21
LOG10	Section A.27

A.27 LOG10

The LOG10 function returns the common log to the base 10 of a specified number.

Syntax

LOG10(<expN>)

Usage

The LOG10 function returns the value for y in the equation $y = LOG10(x)$ where x is the numeric expression used by the LOG10 function. This must be a positive integer for the value of <expN>. LOG10 returns the value of y.

Examples

The following expression returns 0.3010.

```
LOG10(2.0000)
```

See Also

EXP	Section A.21
LOG	Section A.26

A.28 LOWER

The LOWER function converts uppercase letters to lowercase letters.

Syntax

LOWER(<expC>)

Examples

The following expression returns “this is a nice day”.

```
LOWER("THIS IS A NICE DAY")
```

See Also

UPPER	Section A.51

A.29 LTRIM

The LTRIM function removes leading blanks from a character string.

Syntax

LTRIM(<expC>)

Usage

Use this function to remove leading blanks.

Examples

The following expression returns “Bailey”.

```
LTRIM("      Bailey")
```

See Also

LEFT	Section A.24
RIGHT	Section A.36
RTRIM	Section A.39
STR	Section A.45
SUBSTR	Section A.47

A.30 MDY

The MDY function converts the date format to month day, year.

Syntax

MDY(<expD>)

Usage

The MDY function returns the date as a character expression in a month (full name of month) day (two digits), year (two digits) format. If the environment variable SQLSRV_ENV_CENTURY is ON, four digits are displayed for the year.

Examples

If SQLSRV_ENV_CENTURY is ON, the following expression returns "February 29, 1988".

```
MDY({02/29/88})
```

See Also

DMY	Section A.16

A.31 MOD

The MOD function returns the remainder from a division of two numeric expressions. MOD is particularly useful for converting units, such as inches to yards where the division often leaves a remainder.

Syntax

MOD(<expN1>, <expN2>)

Usage

The MOD function returns a whole number, the modulus, which is the remainder of the division of <expN1> by <expN2>.

MOD returns a positive number if <expN2> is positive and a negative number if <expN2> is negative.

The modulus formula is:

<expN1> - FLOOR(<expN1>/<expN2>) * <expN2>

where FLOOR is a mathematical function that returns the greatest integer less than or equal to its argument.

Examples

The following expression returns 2.

```
MOD(14,12)
```

The following expression returns 0.

```
MOD(0,32)
```

The following expression returns –2.

```
MOD(1,-3)
```

See Also

FLOOR	Section A.22
INT	Section A.23

A.32 MONTH

The MONTH function returns a number representing the month from a date expression.

Syntax

MONTH(<expD>)

Usage

The date expression is a placeholder or any function that returns date type data.

Examples

The following expression returns 5.00.

```
MONTH({05/15/87})
```

See Also

CMONTH	Section A.11
DAY	Section A.14
YEAR	Section A.54

A.33 PI

The PI function returns the irrational number 3.14159, which is an approximation of the constant pi (π), the ratio of the circumference of a circle to its diameter.

Syntax

PI()

Usage

The constant pi (π) is used in mathematical and engineering calculations.

Examples

The following expression returns 3.14.

```
PI()
```

A.34 RAND

The RAND function generates a random number. [1]

Syntax

RAND([<expN>])

Arguments

<expN>
An optional numeric expression used as the seed to generate a new random number. If the expression is a negative number, the seed is taken from the system clock.

Usage

The RAND function computes a random number with or without a numeric argument. You can repeat the function without an argument in order to get subsequent random numbers in that sequence.

This function returns numbers between 0 and 0.999999 inclusive.

The default seed number is 100001. To reset the seed to the default value, use RAND(100001).

Examples

The following expression returns 0.13.

```
RAND(23)
```

The following expression returns the next random number.

```
RAND()
```

[1] Although this description uses the word "random," the value returned by the RAND function is a pseudorandom number, that is, one of a very large but finite sequence of numbers. Computers cannot generate truly random numbers.

A.35 REPLICATE

The REPLICATE function repeats a character expression a specified number of times.

Syntax

REPLICATE(<expC>, <expN>)

Arguments

<expC>
The character string to repeat

<expN>
The number of times to repeat <expC>

Usage

The output string must not exceed 254 characters (<expN> must be a number less than 254 divided by the number of characters in <expC>). Thus, when you use the REPLICATE function to create histograms, you may need to use a weighting factor.

Examples

The following expression returns "*****"

```
REPLICATE("*",5)
```

A.36 RIGHT

The RIGHT function returns a specified number of characters from a character expression, starting from the last character on the right.

Syntax

RIGHT(<expC>, <expN>)

Usage

The RIGHT function allows you to retrieve the last part of a character string or a variable. The numeric expression defines the number of characters to extract from the character string or variable.

If the numeric expression is zero or negative, RIGHT returns an empty string.

If the numeric expression is greater than the length of the character string, RIGHT returns the entire string.

Examples

The following expression returns "def".

```
RIGHT("abcdef",3)
```

See Also

AT	Section A.5
LEFT	Section A.24
LTRIM	Section A.29
RTRIM	Section A.39
STUFF	Section A.46
SUBSTR	Section A.47

A.37 ROUND

The ROUND function rounds fractions off to a specified number of decimal places. Negative numbers round as if they were positive.

Syntax

ROUND(<expN1>, <expN2>)

Arguments

<expN1>
The number or numeric expression you want to round

<expN2>
The number of decimal places you want to retain. If <expN2> is negative, ROUND returns a rounded whole number.

Examples

The following expression returns 14.75.

```
ROUND(14.746321,2)
```

The following expression returns 11.

```
ROUND(10.7654321,0)
```

The following expression returns 15000.

```
ROUND (14911,-3)
```

The following expression returns –6.

```
ROUND(-5.8,0)
```

The following expression returns –5.

```
ROUND(-5.2,0)
```

See Also

CEILING	Section A.9
FLOOR	Section A.22
INT	Section A.23
STR	Section A.45
VAL	Section A.53

A.38 RTOD

The RTOD function converts radians to degrees.

Syntax

RTOD(<expN>)

Arguments

<expN>
A number representing an angle size in degrees

Usage

Use this function to convert radians to degrees.

Examples

The following expression returns 270.

```
RTOD(3 * PI/2)
```

See Also

ACOS	Section A.2
ASIN	Section A.4
ATAN	Section A.6
ATN2	Section A.7
COS	Section A.12
DTOR	Section A.19
SIN	Section A.41
TAN	Section A.48

A.39 RTRIM

The RTRIM function removes all trailing blanks from a character string. This function is identical to the TRIM function.

Syntax

RTRIM(<expC>)

Usage

Use this function to trim trailing blanks from character strings. RTRIM(<expC>) followed by a comma inserts one blank space before the next string. RTRIM(<expC>) followed by a plus sign does not insert any blank space before the next string.

Examples

The following expression returns “Jones”.

```
RTRIM("Jones       ")
```

See Also

LEFT	Section A.24
LTRIM	Section A.29
RIGHT	Section A.36
TRIM	Section A.50

A.40 SIGN

The SIGN function returns a number representing the mathematical sign of a numeric expression. It returns a 1 for a positive number, a –1 for a negative number, and a 0 for zero.

Syntax

SIGN(<expN>)

Arguments

<expN>
A numeric expression

Usage

Use SIGN when the result of a calculation must have the same sign as the initial values used, but where the result of the calculation can be of either sign.

Examples

The following expression returns –1.

```
SIGN(-999)
```

See Also

ABS	Section A.1

A.41 SIN

The SIN function returns the trigonometric sine of an angle.

Syntax

SIN(<expN>)

Arguments

<expN>
Is a numeric expression representing the size of the angle (in radians)

Usage

Use this function to get the sine of an angle. No limits are placed on the argument.

Examples

The following expression returns 1.

```
SIN(PI/2)
```

The following expression returns 0.

```
SIN(PI)
```

The following expression returns –1.

```
SIN(3*PI/2)
```

The following expression returns 0.

```
SIN(2*PI)
```

See Also

ACOS	Section A.2
ASIN	Section A.4
ATAN	Section A.6
ATN2	Section A.7
COS	Section A.12
DTOR	Section A.19
PI	Section A.33
RTOD	Section A.38
TAN	Section A.48

A.42 SOUNDEX

The SOUNDEX function provides a phonetic match (sound-alike) code to find a match when the exact spelling is not known.

Syntax

SOUNDEX(<expC>)

Usage

The SOUNDEX function returns a 4-character code by using the following algorithm:

1 It retains the first letter of <expC>, the specified character expression.

2 It drops all occurrences of the letters a e h i o u w y in all positions except the first one.

3 It assigns a number to the remaining letters:

b f p v	1
c g j k q s x z	2
d t	3
l	4
m n	5
r	6

4 If two or more adjacent letters have the same code, it drops all but the first letter.

5 It provides a code of the form "letter digit digit digit". It adds trailing zeros if there are fewer than three digits. It drops all digits after the third digit on the right.

6 It stops at the first nonalphabetic character.

7 It skips over leading blanks.

8 It returns "0000" if the first nonblank character is non-alphabetic.

These steps produce a 4-character code. This code is used to find possible sound-alike matches.

Examples

The following expression returns “K516”.

```
SOUNDEX("Kimberlee")
```

The following expression returns “K516”.

```
SOUNDEX("Kimbrelea")
```

The following expression returns “K516”.

```
SOUNDEX("Kimburley")
```

See Also

DIFFERENCE Section A.15

A.43 SPACE

The SPACE function generates a character string consisting of a specified number of spaces.

Syntax

SPACE(<expN>)

Arguments

<expN>
A number less than or equal to 254

Examples

The following expression returns 20 space characters.

```
SPACE(20)
```

A.44 SQRT

The SQRT function returns the square root of a positive number.

Syntax

```
SQRT(<expN>)
```

Usage

SQRT returns a square root value of the number specified in <expN>.

Examples

The following expression returns 2.

```
SQRT(4)
```

A.45 STR

The STR function converts a number to a character string.

Syntax

STR(<expN> [,<length> [,<decimal>]])

Arguments

<expN>
A numeric expression

<length>
Specifies the number of characters in the string returned by STR, including, if applicable, the decimal point, minus sign, and the number of decimal places. The default is ten characters. If you specify a smaller <length> than there are digits to the left of the decimal in the numeric expression, STR returns asterisks in place of the number.

<decimal>
Specifies the total number of decimal places to output. If necessary, STR rounds <expN> to fit. The default is 0; that is, <expN> is rounded to an integer.

Examples

The following examples use the STR function to display the number 11.14 * 10 as a character string:

The following expression returns "111".

```
STR(111.4,5)
```

The following expression returns "111.4".

```
STR(111.4,5,1)
```

The following expression returns "111.4".

```
STR(111.4,5,2)
```

See Also

VAL	Section A.53

A.46 STUFF

The STUFF function replaces a portion of a character string with another specified character string.

Syntax

```
STUFF(<expC1>,<expN1>, <expN2>,<expC2>)
```

Arguments

<expC1>
A character expression or a variable name

<expN1>
A numeric expression

<expN2>
A numeric expression that is zero or a positive number

<expC2>
A character expression or a variable name

Usage

Use the STUFF function to change part of a character string without reconstructing the entire string. The <expC2> argument is inserted into the character expression at the position indicated by <expN1>. A number of characters indicated by <expN2> are removed from the right of the string.

If the string starting position indicated by <expN1> is zero, STUFF treats it as 0. If it exceeds the length of the variable, it concatenates to the end.

The <expN2> argument indicates how many characters you want to remove from the original string. If the number of characters is zero, the second character expression is inserted, and no characters are removed from <expC1>. The new string will not be the same size as the original string if the specified number of characters in <expN2> differs from the actual number of characters in <expN1>.

Examples

The following expression returns “axxxdef”.

```
STUFF("abcdef",3,2,"xxx")
```

See Also

LEFT	Section A.24
RIGHT	Section A.36
SUBSTR	Section A.47

A.47 SUBSTR

The SUBSTR function extracts a specified number of characters from a character expression or a variable.

Syntax

SUBSTR(<expC>,<starting position>[,<number of characters>])

Usage

If you omit the number of characters, the function returns a substring that begins with the starting position and ends with the last character of the original character string.

If the number of characters you enter is greater than the number of characters between the starting position and the end of the original character expression, the function returns a substring that begins at the specified starting position and ends with the last character of the original character expression. The starting position must be positive.

Examples

The following expression returns 59.

```
SUBSTR("1958 1959 1960",8,2)
```

See Also

AT	Section A.5
LEFT	Section A.24
LTRIM	Section A.29
RIGHT	Section A.36
STR	Section A.45
STUFF	Section A.46

A.48 TAN

The TAN function returns the trigonometric tangent of an angle.

Syntax

TAN(<expN>)

Arguments

<expN>
The size of the angle expressed in radians

Usage

This trignometric function increases from zero to infinity between 0 to $\pi/2$ radians.

Examples

The following expression returns 0.

```
TAN(PI)
```

See Also

ACOS	Section A.2
ASIN	Section A.4
ATAN	Section A.6
ATN2	Section A.7
COS	Section A.12
SIN	Section A.41

A.49 TIME

The TIME function returns the system time as a character string in the format hh:mm:ss.

Syntax

```
TIME( )
```

Usage

To use TIME in calculations, convert the value returned to a numeric value using SUBSTR and VAL.

A.50 TRIM

The TRIM function removes all trailing blanks from a character string. This function is identical to the RTRIM function.

Syntax

TRIM(<expC>)

Usage

Use this function to trim trailing blanks from character strings. TRIM(<expC>) followed by a comma inserts one blank space before the next string. TRIM(<expC>) followed by a plus sign does not insert any blank space before the next string.

Examples

The following expression returns “Jones”.

```
TRIM("Jones      ")
```

See Also

LEFT	Section A.24
LTRIM	Section A.29
RIGHT	Section A.36
RTRIM	Section A.39

A.51 UPPER

The UPPER function converts lowercase letters to uppercase letters.

Syntax

UPPER(<expC>)

Examples

The following expression returns “THIS IS A NICE DAY”.

```
UPPER("This is a nice day")
```

See Also

LOWER	Section A.28

A.52 USER

The USER function returns the user name of the currently active association.

Syntax

```
USER()
```

A.53 VAL

The VAL function converts numbers that are defined as characters into a numeric expression.

Syntax

VAL(<expC>)

Usage

If the specified character expression consists of leading non-numeric characters other than blanks, VAL returns a value of zero.

The VAL function operates from left to right, converting characters to numeric values until a non-numeric character is encountered. Leading blanks are ignored if the argument contains both numeric and non-numeric characters. The leading numeric characters are converted to a numeric value. Trailing blanks are treated as non-numeric characters and, when encountered, terminate the conversion process.

Examples

The following expression returns 0.

```
VAL("ABC")
```

The following expression returns 0.

```
VAL("A=123")
```

The following expression returns 123.

```
VAL("123=A")
```

See Also

STR	Section A.45

A.54 YEAR

The YEAR function returns the numeric value of the year from a date expression. The result is always a 4-digit number.

Syntax

YEAR(<expD>)

Examples

The following expression returns 1988.

```
YEAR({02/29/88})
```

B

SQL/Services Sample Application

This appendix gives complete source code listings for the two modules that comprise the SQLSRV$DYNAMIC program. SQLSRV$DRIVER.C is listed in Example B–1. SQLSRV$DYNAMIC.C is listed in Example B–2.

Example B–1 The SQLSRV$DRIVER.C Module

```
/*   SQLSRV$DRIVER.C                                                     */
/*                                                                       */
/*   This module is part of an application program that demonstrates    */
/*   SQL/Services.  It is provided for instructional purposes only.      */
/*                                                                       */
/*   This module accepts a string from the terminal that contains an SQL  */
/*   statement and then calls the other module (SQLSRV$DYNAMIC) to process */
/*   it.                                                                 */
/*                                                                       */
/*   You can substitute your own module for this driver. Instead of using */
/*   terminal I/O, your module could construct an SQL statement from     */
/*   parameters passed by a calling module. For example, your module could */
/*   parse a non-SQL statement from a front-end system and build an SQL  */
/*   statement from it.                                                  */
/*                                                                       */
/*   However the module generates an SQL statement, it can be passed to a */
/*   module similar to SQLSRV$DYNAMIC for processing.                    */

#include <stdio.h>          /* Standard input/output.                    */
#include <sqlsrvda.h>       /* SQLDA structure definition.               */
#include <sqlsrvca.h>       /* SQLCA structure, error definition.        */
#include <sqlsrv.h>         /* SQL Services structure definitions.       */
```

(continued on next page)

Example B–1 (Cont.) The SQLSRV$DRIVER.C Module

```
main(argc,argv)
int     argc;
char    *argv[];
{
    /* Variables for association */

    char            *assoc_id;              /* Association handle.          */
    struct SQLCA    sqlca_str;              /* SQL Context Area.            */
    char            long_error[512];        /* Alternative error buffer.    */

    /* Other variables */

    char            sql_statement[1024];    /* SQL statement text           */
    int             sts, echo = 0;

    /* The definitions of the create_association and release_association    */
    /* functions are in SQLSRV$DYNAMIC.                                     */

    sts = create_association(argc, argv, &assoc_id, &sqlca_str, long_error);
    if (sts != SQL_SUCCESS)
        return sts;

    /* Print user instructions once.                                        */

    printf(" \n");
    printf("Enter any dynamically executable SQL statement, \n");
    printf("continuing it on successive lines.\n");
    printf("Terminate each statement with a semicolon.\n");
    printf("Built-in commands are: [no]echo and exit.\n");
    printf(" \n");

    while (1) {
        get_statement(sql_statement, echo);

        /* these string comparisons are case-sensitive */

        if (!strcmp(sql_statement, "echo"))
            echo = 1;
        else if (!strcmp(sql_statement, "noecho"))
            echo = 0;
        else if (!strcmp(sql_statement, "exit"))
            break;
        else
            execute_statement(assoc_id,&sqlca_str,sql_statement,long_error);
    } /* while */

    release_association(assoc_id,&sqlca_str,long_error);

} /* main */
```

(continued on next page)

Example B–1 (Cont.) The SQLSRV$DRIVER.C Module

```
get_statement(sql_statement,echo)
char    *sql_statement;
int     echo;
{
    /* Get SQL statement from user, concatenating partial statements using  */
    /* one space character as a separator.                                  */

    char    part_stmt[256];         /* temporaries                          */
    int     end_of_stmt = 0;        /* flag for end of statement            */

    printf("SQL> ");
    sql_statement[0] = '\0';        /* init statement string                */
    while (!end_of_stmt) {
        get_partial(part_stmt,&end_of_stmt,echo);
        if (strlen(sql_statement) != 0)
            strcat(sql_statement," ");    /* add separator character        */
        if (strlen(part_stmt) > 0)
            strcat(sql_statement,part_stmt);
        if (!end_of_stmt)
            printf ("cont> ");
    } /* while */
} /* get_statement */

get_partial(part_stmt,end_of_stmt,echo)
char    *part_stmt;
int     *end_of_stmt;
int     echo;
{
    /* Get partial statement from user. Accept semicolon as line terminator */
    /* and exclamation point as comment line.                               */

    int     len;

    *end_of_stmt = 0;
    gets(part_stmt);
    if (echo)
        printf("%s\n",part_stmt);
    len = strlen(part_stmt);
    if (len > 0) {
        trim(&part_stmt[len-1]);    /* delete trailing white space          */
        len = strlen(part_stmt);
        if (len > 0) {
            if (part_stmt[0] == '!') /* delete comments                     */
                part_stmt[0] = '\0';
            else
                *end_of_stmt = (part_stmt[len-1] == ';');
            if (*end_of_stmt) {
                part_stmt[len-1] = '\0'; /* delete semicolon                */
                if (len > 1)
                    trim(&part_stmt[len-2]); /* delete white space          */
            } /* if */
        } /* if */
    } /* if */
} /* get_partial */
```

(continued on next page)

Example B-1 (Cont.) The SQLSRV$DRIVER.C Module

```
trim(string)
char *string;
{
    if (*string == ' ' || *string == '\t') {
        *string = '\0';
        trim(--string);
    }
}
```

Example B-2 The SQLSRV$DYNAMIC.C Module

```
/*  SQLSRV$DYNAMIC.C                                                       */
/*                                                                         */
/*  This module is part of an application program that demonstrates       */
/*  SQL/Services.  It is provided for instructional purposes only.         */
/*                                                                         */
/*  This module contains the following routines:                           */
/*                                                                         */
/*  create_association                                                     */
/*                                                                         */
/*  Creates an SQL/SERVICES client/server association. Checks command line */
/*  argument vector for names of server system, account, and password. If  */
/*  not present, prompts user.                                             */
/*                                                                         */
/*  release_association                                                    */
/*                                                                         */
/*  Terminates an SQL/SERVICES client/server association.                  */
/*                                                                         */
/*  execute_statement                                                      */
/*                                                                         */
/*  Accepts a string containing a dynamically executable SQL statement from */
/*  the other module (SQLSRV$DRIVER). If parameter markers are present, it  */
/*  calls get_params. If the statement is a SELECT, it opens a cursor,      */
/*  fetches rows, and displays them. If the statement is not a SELECT, it   */
/*  executes the statement.                                                */
/*                                                                         */
/*  get_params                                                             */
/*                                                                         */
/*  For each parameter marker in the SQL statement, get_params checks the   */
/*  data type and inputs data from the terminal.                           */
/*                                                                         */
/*  report_error                                                           */
/*                                                                         */
/*  Prints out the message that corresponds to the error code in the SQLCA. */
/*  Also prints out error messages text if present. Aborts on DECnet       */
/*  errors.                                                                */

#include <stdio.h>      /* Standard input/output.              */
#include <sqlsrvda.h>   /* SQLDA structure definition.         */
#include <sqlsrvca.h>   /* SQLCA structure, error definition.   */
#include <sqlsrv.h>     /* SQL/Services structure definitions.  */
```

(continued on next page)

Example B–2 (Cont.) The SQLSRV$DYNAMIC.C Module

```
create_association(argc,argv,assoc_id,sqlca_str,long_error)
int             argc;           /* argument count                   */
char            *argv[];        /* argument vector                  */
char            **assoc_id;     /* address of association id used   */
                                /* in all SQL/Services calls.       */
struct SQLCA    *sqlca_str;     /* context structure                */
char            *long_error;    /* alternative error buffer         */
{
    /* Variables and structures for SQL/Services API */

    struct ASSOCIATE_STR    associate_str;  /* Association structure.       */
    char        node_name[8];               /* VMS node name.               */
    char        user_name[32];              /* VMS user name.               */
    char        password[32];               /* VMS password.                */
    static char read_buffer[512];           /* Protocol read buffer.        */
    static char write_buffer[512];          /* Protocol write buffer.       */
    long int    read_size, write_size;      /* Protocol buffer sizes.       */

    /* Other variables                                                      */

    int         sts;                        /* return status value.         */
    int         i;                          /* loop counter.                */

    /* Get the node name, user name and password values for the server      */
    /* connection. Prompt the user if not in argument vector.               */

    switch (argc) {
    case 1:
        printf("VMS server node: ");
        gets(node_name);
        printf("VMS server account name: ");
        gets(user_name);
        printf("VMS server account password: ");
        gets(password);
        break;
    case 2:
        strcpy(node_name, argv[1]);
        printf("VMS server account name: ");
        gets(user_name);
        printf("VMS server account password: ");
        gets(password);
        break;
    case 3:
        strcpy(node_name, argv[1]);
        strcpy(user_name, argv[2]);
        printf("VMS server account password: ");
        gets(password);
        break;
    case 4:
        strcpy(node_name, argv[1]);
        strcpy(user_name, argv[2]);
        strcpy(password, argv[3]);
        break;
```

(continued on next page)

Example B–2 (Cont.) The SQLSRV$DYNAMIC.C Module

```
    default:
        for (i = 4; i < argc; i++)
            printf ("Extraneous argument ignored: %s\n", argv[i]);
        break;
    } /* switch */

    read_size = 1024;   /* protocol buffer size value */
    write_size = 1024;  /* protocol buffer size value */

    /* Set up association structure */

    associate_str.CLIENT_LOG = 0;           /* disable client logging.      */
    associate_str.SERVER_LOG = 0;           /* disable server logging.      */
    associate_str.LOCAL_FLAG = 0;           /* this is a remote session.    */
    associate_str.MEMORY_ROUTINE = NULL;    /* use default alloc routine.   */
    associate_str.FREE_MEMORY_ROUTINE = NULL; /* use default free routine.  */
    associate_str.ERRBUFLEN = 512;
    associate_str.ERRBUF = long_error;      /* use alternative error string */

    /* Connect with the server and establish an association.                */

    sts = sqlsrv_associate(
                node_name,              /* node name.               */
                user_name,              /* user name.               */
                password,               /* password.                */
                read_buffer,            /* protocol read buffer.    */
                write_buffer,           /* protocol write buffer.   */
                read_size,              /* read buffer size.        */
                write_size,             /* write buffer size.       */
                sqlca_str,              /* SQLCA structure.         */
                &associate_str,         /* Association structure.   */
                assoc_id                /* Association handle.      */
                );

    if (sts != SQL_SUCCESS)
                return report_error(*assoc_id, sqlca_str, long_error);

} /* create_association */

release_association(assoc_id, sqlca_str, long_error)
char            *assoc_id;      /* association handle        */
struct SQLCA    *sqlca_str;     /* context structure         */
char            *long_error;    /* alternative error buffer */
{
    int         sts;            /* return status value. */
    char        *stats = NULL;  /* reserved parameter */
    /*
     * release the association.
     */
    sts = sqlsrv_release(assoc_id,stats);

    if (sts != SQL_SUCCESS)
        return report_error(assoc_id, sqlca_str, long_error);

} /* release_association */
```

(continued on next page)

Example B–2 (Cont.) The SQLSRV$DYNAMIC.C Module

```
execute_statement(assoc_id, sqlca_str, sql_statement, long_error)
char            *assoc_id;      /* association handle.         */
struct SQLCA    *sqlca_str;     /* Context structure.          */
char            *sql_statement; /* SQL statement to execute    */
char            *long_error;    /* alternative error buffer    */
{
    /* Variables and structures for SQL/Services API                        */

    int             sts;                    /* return status value.         */
    short int       execute_flag;           /* Execute mode flag.           */
    long int        statement_id;           /* Prepared statement id.       */
    char            *cursor_name = "SEL";   /* Name of cursor.              */
    long int        database_id = 0L;       /* Database ID. Not in V1.0.    */
    struct SQLDA    *param_sqlda;           /* Parameter marker SQLDA.      */
    struct SQLDA    *select_sqlda;          /* Select list SQLDA.           */

    /* Other variables                                                      */

    int             i;                      /* Loop counter                 */
    int             len;                    /* temporary                    */
    char            *p;                     /* temporary                    */

    /* Call the sqlsrv_prepare routine to prepare the SQL statement and to  */
    /* write parameter marker and select list information into the SQLDA    */
    /* structures. If you pass NULL pointers to the parameter marker SQLDA  */
    /* and the select list SQLDA, sqlsrv_prepare allocates and initializes  */
    /* the structures if they are required.                                 */

    select_sqlda = NULL;
    param_sqlda = NULL;

    /* You can also pass in existing SQLDA structures, in which case the    */
    /* sqlsrv_prepare routine initializes them.                             */

    sts = sqlsrv_prepare(
                        assoc_id,           /* association handle.          */
                        database_id,        /* database_id, must be zero.   */
                        sql_statement,      /* SQL statement.               */
                        &statement_id,      /* Prepared statement id.       */
                        &param_sqlda,
                        &select_sqlda);

    if (sts != SQL_SUCCESS)
        return report_error(assoc_id, sqlca_str, long_error);
```

(continued on next page)

Example B-2 (Cont.) The SQLSRV$DYNAMIC.C Module

```
/* The call to sqlsrv_prepare succeeded. If it allocated a param_sqlda  */
/* structure, the SQL statement contains parameter markers. NOTE: if    */
/* you preallocated param_sqlda, test (param_sqlda.SQLD > 0) here.      */

if (param_sqlda) {

    /* Call routine to allocate data and indicator variables            */

    sts = sqlsrv_allocate_sqlda_data(assoc_id, param_sqlda);

    if (sts != SQL_SUCCESS)
        return report_error(assoc_id, sqlca_str, long_error);

    /* get values for parameter markers                                 */

    get_params(param_sqlda);
}

/* If the sqlsrv_prepare routine allocated a select list SQLDA, the     */
/* statement is a SELECT.  Open a cursor, fetch rows, display them on  */
/* the terminal, and close the cursor. NOTE: if you are using a         */
/* preallocated SQLDA, test (select_sqlda.SQLD > 0) here.               */

if (select_sqlda) {

    /* Call routine to allocate data and indicator variables    */

    sts = sqlsrv_allocate_sqlda_data(assoc_id, select_sqlda);

    if (sts != SQL_SUCCESS)
        return report_error(assoc_id, sqlca_str, long_error);

    sts = sqlsrv_open_cursor(
                    assoc_id,       /* association id                   */
                    cursor_name,    /* handle for cursor                */
                    statement_id,   /* handle for SELECT statement      */
                    param_sqlda     /* parameter marker SQLDA           */
                    );

    if (sts != SQL_SUCCESS)
        return report_error(assoc_id, sqlca_str, long_error);

    /* fetch and display rows */

    printf("------ BEGIN RESULT TABLE ------\n");
    do {
        sts = sqlsrv_fetch(
                assoc_id,       /* association id       */
                cursor_name,    /* handle for cursor    */
                0,              /* direction            */
                0L,             /* row number           */
                select_sqlda    /* select list SQLDA    */
                );
```

(continued on next page)

Example B–2 (Cont.)   The SQLSRV$DYNAMIC.C Module

```
        switch (sts) {
            case SQL_SUCCESS:
                for (i = 0; i < select_sqlda->SQLD; i++) {

                    /* SQLD contains number of columns */

                    /* print first 20 chars of column name */

                    printf("%-20.20s: ",select_sqlda->SQLVARARY[i].SQLNAME);

                    /* check the indicator variable for NULL value */

                    if (*select_sqlda->SQLVARARY[i].SQLIND < 0)
                        printf("NULL\n");
                    else
                        switch (select_sqlda->SQLVARARY[i].SQLTYPE) {
                        case SQLSRV_ASCII_STRING:
                        case SQLSRV_GENERALIZED_NUMBER:
                        case SQLSRV_GENERALIZED_DATE:

                            /* Null-terminated strings */

                            printf("%s\n",
                                select_sqlda->SQLVARARY[i].SQLDATA);
                            break;

                        case SQLSRV_VARCHAR:

                            /* Counted string. The first word of the    */
                            /* data buffer is the length. Set a pointer */
                            /* to the first ASCII character and print.  */

                            p = select_sqlda->SQLVARARY[i].SQLDATA;
                            len = *(short int *)p;
                            p += sizeof(short int);
                            printf("%-*.*s\n", len, len, p);

                            /* Note: SQLSRV_VARCHAR data is likely to   */
                            /* be binary. A real application wouldn't   */
                            /* print it on the terminal.                */;

                            break;
                        } /* switch */

                } /* for */
                printf("---------- END OF ROW ----------\n");
                break;
            case SQL_EOS:
                printf("------- END RESULT TABLE -------\n");
                break;
            default:
                return report_error(assoc_id, sqlca_str, long_error);
                break;
        } /* switch */
    } while (sts != SQL_EOS);

    sts = sqlsrv_close_cursor(assoc_id, cursor_name);

    if (sts != SQL_SUCCESS)
        return report_error(assoc_id, sqlca_str, long_error);
}
```

(continued on next page)

Example B–2 (Cont.) The SQLSRV$DYNAMIC.C Module

```
    else {

        /* The SQL statement is not a SELECT and can be executed now.       */

        execute_flag = 0; /* Turn batching off. */

        sts = sqlsrv_execute(
                    assoc_id,               /* association handle.          */
                    database_id,            /* database_id, must be zero.   */
                    statement_id,           /* Prepared statement id.       */
                    execute_flag,           /* Execute mode.                */
                    param_sqlda             /* Parameter marker SQLDA.      */
                    );

        if (sts != SQL_SUCCESS)
            return report_error(assoc_id, sqlca_str, long_error);

    } /* else */

    /* Release the SQL statement resources */

    sts = sqlsrv_release_statement(
                        assoc_id,           /* association handle.          */
                        1,                  /* no. of statement ids.        */
                        &statement_id       /* statement id array.          */
                        );

    /* NOTE: You can pass in multiple statement ids in array format. We're  */
    /* only passing one here.  In C, an array is a pointer, so by passing a */
    /* pointer, we pass an array of 1.                                      */

    if (sts != SQL_SUCCESS)
        return report_error(assoc_id, sqlca_str, long_error);

    return(SQL_SUCCESS);

} /* execute_statement */

get_params(param_sqlda)
struct SQLDA    *param_sqlda;   /* Parameter marker SQLDA. */
{
    int     i;          /* loop counter */
    int     len;        /* temporary */
    char    s[80],*p;   /* temporary */

    for (i = 0; i < param_sqlda->SQLD; i++) {

        /* SQLD contains the number of parameter markers */
```

(continued on next page)

Example B-2 (Cont.) The SQLSRV$DYNAMIC.C Module

```
switch(param_sqlda->SQLVARARY[i].SQLTYPE) {

    /* branch on the data type of the parameter */

    case SQLSRV_ASCII_STRING:       /* null-terminated strings */
    case SQLSRV_GENERALIZED_NUMBER:
        do {
            printf("Enter value for:   ");
            printf("%s\n", param_sqlda->SQLVARARY[i].SQLNAME);
            printf("Maximum length is: ");
            printf("%d\n", param_sqlda->SQLVARARY[i].SQLLEN);
            printf("DATA> ");
            gets(param_sqlda->SQLVARARY[i].SQLDATA);
            len = strlen(param_sqlda->SQLVARARY[i].SQLDATA);
            if (len == 0)
                printf("Value required. Please reenter.");
        } while (len == 0);
        break;
    case SQLSRV_VARCHAR:    /* counted string */
        do {
            printf("Enter value for:   ");
            printf("%s\n", param_sqlda->SQLVARARY[i].SQLNAME);
            printf("Maximum length is: ");
            printf("%d\n", param_sqlda->SQLVARARY[i].SQLLEN);
            printf("DATA> ");
            gets(s);

            /* Get the length and write it into the first word of   */
            /* the buffer. Set a pointer to the next byte and copy  */
            /* in the ASCII data.                                   */

            len = strlen(s);
            p = param_sqlda->SQLVARARY[i].SQLDATA;
            *(short int *)p = len;
            p += sizeof(short int);
            strncpy(p,s,len);
            if (len == 0)
                printf("Value required. Please reenter.");
        } while (len == 0);
        break;
    case SQLSRV_GENERALIZED_DATE:   /* null-terminated string */
        do {
            printf("Enter value for:   ");
            printf("%s\n", param_sqlda->SQLVARARY[i].SQLNAME);
            printf("Maximum length is: ");
            printf("%d\n", param_sqlda->SQLVARARY[i].SQLLEN);
            printf("Format is: ccyymmddhhmissff\n");
            printf("DATA> ");
            gets(param_sqlda->SQLVARARY[i].SQLDATA);
            len = strlen(param_sqlda->SQLVARARY[i].SQLDATA);
            if (len == 0)
                printf("Value required. Please reenter.");
        } while (len == 0);
        break;
```

(continued on next page)

Example B-2 (Cont.) The SQLSRV$DYNAMIC.C Module

```
            default:
                printf("Invalid data type: %d\n",
                    param_sqlda->SQLVARARY[i].SQLTYPE);
                gets(s); /* dispose of value */
                break;
        } /* switch */
    } /* for */
    return(SQL_SUCCESS);

} /* get_params */

report_error(assoc_id, sqlca_str, long_error)
char            *assoc_id;      /* association handle       */
struct SQLCA    *sqlca_str;     /* context structure        */
char            *long_error;    /* alternative error buffer */
{
    char        *stats = NULL;  /* reserved parameter */

    switch (sqlca_str->SQLCODE) {
    case SQLSRV_CNDERR:
        printf("Filter runtime error.\n");
        break;
    case SQLSRV_FTRSYNERR:
        printf("Syntax error in filter expression.");
        break;
    case SQLSRV_INTERR:
        printf("Internal error. Examine SQLSRV.DMP and submit SPR.\n");
        break;
    case SQLSRV_INVARG:
        printf("Invalid routine parameter.\n");
        break;
    case SQLSRV_INVASC:
        printf("Invalid association id.\n");
        break;
    case SQLSRV_INVASCSTR:
        printf("Invalid parameter in ASSOCIATE_STR.\n");
        break;
    case SQLSRV_INVBUFSIZ:
        printf("Invalid read or write buffer size.\n");
        break;
    case SQLSRV_INVCURNAM:
        printf("Invalid cursor name.\n");
        break;
    case SQLSRV_INVENVTAG:
        printf("Invalid environment tag.\n");
        break;
    case SQLSRV_INVENVVAR:
        printf("Invalid environment variable.\n");
        break;
    case SQLSRV_INVEXEFLG:
        printf("Invalid execute flag.\n");
        break;
    case SQLSRV_INVIDX:
        printf("Invalid sqlda_index_array\n");
        break;
```

(continued on next page)

Example B-2 (Cont.) The SQLSRV$DYNAMIC.C Module

```
    case SQLSRV_INVREPCNT:
        printf("Invalid repeat count.\n");
        break;
    case SQLSRV_INVSQLCA:
        printf("Invalid SQLCA structure.\n");
        break;
    case SQLSRV_INVSQLDA:
        printf("Invalid SQLDA structure.\n");
        break;
    case SQLSRV_INVSTMID:
        printf("Invalid statement id.\n");
        break;
    case SQLSRV_MULTI_ACT:
        printf("A batched sqlsrv_execute or\n");
        printf("sqlsrv_fetch_many context is active.\n");
        break;
    case SQLSRV_NETERR:
        printf("DECnet returned an error.\n");
        printf("SQLERRD[0]: x%lx\n", sqlca_str->SQLERRD[0]);
        printf("SQLERRD[2]: %d.\n", sqlca_str->SQLERRD[2]);
        sqlsrv_release(assoc_id,stats);
        exit(2);
        break;
    case SQLSRV_NO_MEM:
        printf("API memory allocation failed.\n");
        break;
    case SQLSRV_OPNLOGFIL:
        printf("Unable to open log file\n");
        break;
    case SQLSRV_PRSERR:
        printf("Fatal error in message parser\n");
        break;
    case SQLSRV_SQLDA_NOTALL:
        printf("Attempt to deallocate static memory\n");
        break;
    case SQLSRV_SRVERR:
        printf("The server returned an error.\n");
        printf("SQLERRD[0]: x%lx\n", sqlca_str->SQLERRD[0]);
        printf("SQLERRD[2]: %d.\n", sqlca_str->SQLERRD[2]);
        sqlsrv_release(assoc_id,stats);
        exit(2);
        break;

    /* SQL Errors */

    case SQL_BAD_TXN_STATE:
        printf("Invalid transaction state\n");
        break;
    case SQL_CURALROPE:
        printf("WARNING Cursor is already open\n");
        break;
    case SQL_CURNOTOPE:
        printf("Cursor not open\n");
        break;
    case SQL_DEADLOCK:
        printf("Deadlock encountered\n");
        break;
```

(continued on next page)

Example B-2 (Cont.) The SQLSRV$DYNAMIC.C Module

```
    case SQL_EOS:
        printf("SELECT or cursor at end of stream\n");
        break;
    case SQL_INTEG_FAIL:
        printf("Constraint failed\n");
        break;
    case SQL_LOCK_CONFLICT:
        printf("Lock conflict\n");
        break;
    case SQL_NO_DUP:
        printf("Duplicate on index\n");
        break;
    case SQL_NOT_VALID:
        printf("Valid-if failed\n");
        break;
    case SQL_NULLNOIND:
        printf("NULL value and no indicator variable\n");
        break;
    case SQL_OUTOFRAN:
        printf("Value is out of range for a host variable\n");
        break;
    case SQL_RDBERR:
        printf("Rdb returned an error\n");
        break;
    case SQL_ROTXN:
        printf("Read/write operation in read-only transaction\n");
        break;
    case SQL_SUCCESS:
        printf("Command completed successfully\n");
        break;
    case SQL_UDCURNOPE:
        printf("Cursor in update or delete not open\n");
        break;
    case SQL_UDCURNPOS:
        printf("Cursor in update or delete not positioned on record\n");
        break;
    default:
        printf("Unknown error\n");
        printf("SQLCA.SQLCODE: %d\n", sqlca_str->SQLCODE);
        break;
    } /* switch */

    /* Print out error message text if present */

    if (strlen(long_error) != 0)
        printf("%s\n", long_error);

    return 1;

} /* report_error */
```

C

Sample Log Files

This appendix gives listings for each of several log files generated by the SQL/Services Installation Verification Procedure. The complete association level log is shown in Example C–1. The complete routine level log is shown in Example C–2. A partial message protocol level log is shown in Example C–3.

Example C–1 Sample Association Level Log

```
ASSOCIATE LEVEL LOG
----SQLSRV_ASSOCIATE
--------SQLSRV_ASSOCIATE ID: 106520
--------NODE: abcdef, USERNAME: xxxxxx, SQLCODE: 0, SQLERRD[0] 0
        .
        .
        .
ASSOCIATE LEVEL LOG
----SQLSRV_RELEASE
--------SQLSRV_ASSOCIATE ID: 106520
```

Example C–2 Sample Routine Level Log

```
ROUTINE LEVEL LOG
----SQLSRV_EXECUTE_IMMEDIATE
--------SQL STATEMENT
------------len: 36, value: create schema filename SQLSRV_SAMPLE

ROUTINE LEVEL LOG
----SQLSRV_EXECUTE_IMMEDIATE
--------SQL STATEMENT
------------len: 119, value: create table SQLSRV_TABLE ( USERNAME       CHAR(32), INTE
------------GER_VALUE   INTEGER, DOUBLE_VALUE   DOUBLE PRECISION, DATE_VALUE   DATE )

ROUTINE LEVEL LOG
----SQLSRV_PREPARE
--------SQL STATEMENT
------------len: 102, value: insert into SQLSRV_TABLE (  USERNAME, INTEGER_VALUE
------------, DOUBLE_VALUE, DATE_VALUE ) values ( ?, ?, ?, ? )

ROUTINE LEVEL LOG
----PARAMETER MARKER SQLDA
--------SQLDA: SQLD 4
--------[0].SQLTYPE: SQLSRV_ASCII_STRING, SQLLEN: 33
------------SQLNAME: USERNAME
--------[1].SQLTYPE: SQLSRV_GENERALIZED_NUMBER, SQLLEN[0] 12, SQLLEN[1] 0
------------SQLNAME: INTEGER_VALUE
--------[2].SQLTYPE: SQLSRV_GENERALIZED_NUMBER, SQLLEN[0] 24, SQLLEN[1] 0
------------SQLNAME: DOUBLE_VALUE
--------[3].SQLTYPE: SQLSRV_GENERALIZED_DATE, SQLLEN: 17
------------SQLNAME: DATE_VALUE

ROUTINE LEVEL LOG
----SQLSRV_ALLOCATE_SQLDA_DATA

ROUTINE LEVEL LOG
----SQLSRV_EXECUTE
--------STATEMENT ID
------------1199896
--------EXECUTE FLAG
             0
--------PARAMETER MARKER SQLDA
------------SQLDA: SQLD 4
----------------[0].SQLTYPE: SQLSRV_ASCII_STRING, SQLIND: 0
--------------------len: 6, value: xxxxxx
----------------[1].SQLTYPE: SQLSRV_GENERALIZED_NUMBER, SQLIND: 0
--------------------len: 1, value: 1
----------------[2].SQLTYPE: SQLSRV_GENERALIZED_NUMBER, SQLIND: 0
--------------------len: 10, value: 128.000000
----------------[3].SQLTYPE: SQLSRV_GENERALIZED_DATE, SQLIND: 0
--------------------len: 8, value: 19880701
```

(continued on next page)

Example C–2 (Cont.) Sample Routine Level Log

```
ROUTINE LEVEL LOG
----SQLSRV_EXECUTE
--------STATEMENT ID
------------1199896
--------EXECUTE FLAG
             0
--------PARAMETER MARKER SQLDA
------------SQLDA: SQLD 4
----------------[0].SQLTYPE: SQLSRV_ASCII_STRING, SQLIND: 0
--------------------len: 6, value: xxxxxx
----------------[1].SQLTYPE: SQLSRV_GENERALIZED_NUMBER, SQLIND: 0
--------------------len: 1, value: 2
----------------[2].SQLTYPE: SQLSRV_GENERALIZED_NUMBER, SQLIND: 0
--------------------len: 12, value: 32768.000000
----------------[3].SQLTYPE: SQLSRV_GENERALIZED_DATE, SQLIND: 0
--------------------len: 8, value: 19880702

ROUTINE LEVEL LOG
----SQLSRV_EXECUTE
--------STATEMENT ID
------------1199896
--------EXECUTE FLAG
             0
--------PARAMETER MARKER SQLDA
------------SQLDA: SQLD 4
----------------[0].SQLTYPE: SQLSRV_ASCII_STRING, SQLIND: 0
--------------------len: 6, value: xxxxxx
----------------[1].SQLTYPE: SQLSRV_GENERALIZED_NUMBER, SQLIND: 0
--------------------len: 1, value: 3
----------------[2].SQLTYPE: SQLSRV_GENERALIZED_NUMBER, SQLIND: 0
--------------------len: 13, value: 524288.000000
----------------[3].SQLTYPE: SQLSRV_GENERALIZED_DATE, SQLIND: 0
--------------------len: 8, value: 19880703

ROUTINE LEVEL LOG
----SQLSRV_RELEASE_STATEMENT
--------STATEMENT ID
------------[0] 1199896

ROUTINE LEVEL LOG
----SQLSRV_FREE_SQLDA_DATA

ROUTINE LEVEL LOG
----SQLSRV_PREPARE
--------SQL STATEMENT
------------len: 45, value: Select * from sqlsrv_table where USERNAME = ?
```

(continued on next page)

Example C–2 (Cont.) Sample Routine Level Log

```
ROUTINE LEVEL LOG
----SELECT LIST SQLDA
--------SQLDA: SQLD 4
--------[0].SQLTYPE: SQLSRV_ASCII_STRING, SQLLEN: 33
------------SQLNAME: USERNAME
--------[1].SQLTYPE: SQLSRV_GENERALIZED_NUMBER, SQLLEN[0] 12, SQLLEN[1] 0
------------SQLNAME: INTEGER_VALUE
--------[2].SQLTYPE: SQLSRV_GENERALIZED_NUMBER, SQLLEN[0] 24, SQLLEN[1] 0
------------SQLNAME: DOUBLE_VALUE
--------[3].SQLTYPE: SQLSRV_GENERALIZED_DATE, SQLLEN: 17
------------SQLNAME: DATE_VALUE

ROUTINE LEVEL LOG
----PARAMETER MARKER SQLDA
--------SQLDA: SQLD 1
--------[0].SQLTYPE: SQLSRV_ASCII_STRING, SQLLEN: 33
------------SQLNAME: USERNAME

ROUTINE LEVEL LOG
----SQLSRV_ALLOCATE_SQLDA_DATA

ROUTINE LEVEL LOG
----SQLSRV_ALLOCATE_SQLDA_DATA

ROUTINE LEVEL LOG
----SQLSRV_OPEN_CURSOR
--------CURSOR NAME
------------sqlsrv_cursor
--------STATEMENT ID
             1199896

ROUTINE LEVEL LOG
----SQLSRV_FETCH
--------CURSOR NAME
------------sqlsrv_cursor

ROUTINE LEVEL LOG
----SELECT LIST SQLDA
--------SQLDA: SQLD 4
------------[0].SQLTYPE: SQLSRV_ASCII_STRING, SQLIND: 0
----------------len: 32, value: xxxxxx
------------[1].SQLTYPE: SQLSRV_GENERALIZED_NUMBER, SQLIND: 0
----------------len: 11, value: 1
------------[2].SQLTYPE: SQLSRV_GENERALIZED_NUMBER, SQLIND: 0
----------------len: 23, value:  1.280000000000000E+002
------------[3].SQLTYPE: SQLSRV_GENERALIZED_DATE, SQLIND: 0
----------------len: 16, value: 1988070100000000

ROUTINE LEVEL LOG
----SQLSRV_FETCH
--------CURSOR NAME
------------sqlsrv_cursor
```

(continued on next page)

Example C-2 (Cont.) Sample Routine Level Log

```
ROUTINE LEVEL LOG
----SELECT LIST SQLDA
--------SQLDA: SQLD 4
------------[0].SQLTYPE: SQLSRV_ASCII_STRING, SQLIND: 0
----------------len: 32, value: xxxxxx
------------[1].SQLTYPE: SQLSRV_GENERALIZED_NUMBER, SQLIND: 0
----------------len: 11, value: 2
------------[2].SQLTYPE: SQLSRV_GENERALIZED_NUMBER, SQLIND: 0
----------------len: 23, value:  3.276800000000000E+004
------------[3].SQLTYPE: SQLSRV_GENERALIZED_DATE, SQLIND: 0
----------------len: 16, value: 1988070200000000

ROUTINE LEVEL LOG
----SQLSRV_FETCH
--------CURSOR NAME
------------sqlsrv_cursor

ROUTINE LEVEL LOG
----SELECT LIST SQLDA
--------SQLDA: SQLD 4
------------[0].SQLTYPE: SQLSRV_ASCII_STRING, SQLIND: 0
----------------len: 32, value: xxxxxx
------------[1].SQLTYPE: SQLSRV_GENERALIZED_NUMBER, SQLIND: 0
----------------len: 11, value: 3
------------[2].SQLTYPE: SQLSRV_GENERALIZED_NUMBER, SQLIND: 0
----------------len: 23, value:  5.242880000000000E+005
------------[3].SQLTYPE: SQLSRV_GENERALIZED_DATE, SQLIND: 0
----------------len: 16, value: 1988070300000000

ROUTINE LEVEL LOG
----SQLSRV_FETCH
--------CURSOR NAME
------------sqlsrv_cursor

ROUTINE LEVEL LOG
----SQLSRV_CLOSE_CURSOR
--------CURSOR NAME
------------sqlsrv_cursor

ROUTINE LEVEL LOG
----SQLSRV_RELEASE_STATEMENT
--------STATEMENT ID
------------[0] 1199896

ROUTINE LEVEL LOG
----SQLSRV_FREE_SQLDA_DATA

ROUTINE LEVEL LOG
----SQLSRV_FREE_SQLDA_DATA

ROUTINE LEVEL LOG
----SQLSRV_PREPARE
--------SQL STATEMENT
------------len: 43, value: delete from SQLSRV_TABLE where USERNAME = ?
```

(continued on next page)

Example C–2 (Cont.) Sample Routine Level Log

```
ROUTINE LEVEL LOG
----PARAMETER MARKER SQLDA
--------SQLDA: SQLD 1
--------[0].SQLTYPE: SQLSRV_ASCII_STRING, SQLLEN: 33
------------SQLNAME: USERNAME

ROUTINE LEVEL LOG
----SQLSRV_ALLOCATE_SQLDA_DATA

ROUTINE LEVEL LOG
----SQLSRV_EXECUTE
--------STATEMENT ID
------------1199896
--------EXECUTE FLAG
             0
--------PARAMETER MARKER SQLDA
------------SQLDA: SQLD 1
----------------[0].SQLTYPE: SQLSRV_ASCII_STRING, SQLIND: 0
--------------------len: 6, value: xxxxxx

ROUTINE LEVEL LOG
----SQLSRV_RELEASE_STATEMENT
--------STATEMENT ID
------------[0] 1199896

ROUTINE LEVEL LOG
----SQLSRV_FREE_SQLDA_DATA

ROUTINE LEVEL LOG
----SQLSRV_EXECUTE_IMMEDIATE
--------SQL STATEMENT
------------len: 6, value: Commit

ROUTINE LEVEL LOG
----SQLSRV_EXECUTE_IMMEDIATE
--------SQL STATEMENT
------------len: 34, value: Drop Schema filename SQLSRV_SAMPLE
```

Example C–3 Sample Message Protocol Level Log

```
PROTOCOL LEVEL LOG CLIENT: write (logonly)
----PACKET ID: 1, PACKET SEQUENCE: 0
--------SQLSRV_ASSOCIATE
------------PROTOCOL VERSION
----------------len: 2, value: 1
------------READ BUFFER SIZE
----------------len: 2, value: 1024
------------WRITE BUFFER SIZE
----------------len: 2, value: 1024
--------END OF MESSAGE

PROTOCOL LEVEL LOG CLIENT: read
----PACKET ID: 1, PACKET SEQUENCE: 0
--------SQLSRV_ASSOCIATE ACK
------------PROTOCOL VERSION
----------------len: 2, value: 1
------------ASSOCIATE ID
----------------len: 2, value: 1
--------END OF MESSAGE

PROTOCOL LEVEL LOG CLIENT: write (logonly)
----PACKET ID: 2, PACKET SEQUENCE: 0
--------SQLSRV_EXECUTE_IMMEDIATE
------------SQL STATEMENT
----------------SQLSRV_ASCII_STRING, len: 36
--------------------len: 36, value: create schema filename SQLSRV_SAMPLE
--------END OF MESSAGE

PROTOCOL LEVEL LOG CLIENT: read
----PACKET ID: 2, PACKET SEQUENCE: 0
--------SQLSRV_EXECUTE_IMMEDIATE ACK
------------STATUS
----------------SQLSRV_GENERALIZED_NUMBER, len: 1
--------------------len: 1, value: 0
------------EXECUTE PARAMETER
----------------SQLSRV_GENERALIZED_NUMBER, len: 1
--------------------len: 1, value: 0
--------END OF MESSAGE

PROTOCOL LEVEL LOG CLIENT: write (logonly)
----PACKET ID: 3, PACKET SEQUENCE: 0
--------SQLSRV_EXECUTE_IMMEDIATE
------------SQL STATEMENT
----------------SQLSRV_ASCII_STRING, len: 119
--------------------len: 119, value: create table SQLSRV_TABLE ( USERNAME
  CHAR(3
--------------------2), INTEGER_VALUE   INTEGER, DOUBLE_VALUE   DOUBLE PRECISION, DA
--------------------TE_VALUE   DATE )
--------END OF MESSAGE
```

(continued on next page)

Example C–3 (Cont.) Sample Message Protocol Level Log

```
PROTOCOL LEVEL LOG CLIENT: read
----PACKET ID: 3, PACKET SEQUENCE: 0
--------SQLSRV_EXECUTE_IMMEDIATE ACK
------------STATUS
----------------SQLSRV_GENERALIZED_NUMBER, len: 1
--------------------len: 1, value: 0
------------EXECUTE PARAMETER
----------------SQLSRV_GENERALIZED_NUMBER, len: 1
--------------------len: 1, value: 0
--------END OF MESSAGE

PROTOCOL LEVEL LOG CLIENT: write (logonly)
----PACKET ID: 4, PACKET SEQUENCE: 0
--------SQLSRV_PREPARE
------------SQL STATEMENT
----------------SQLSRV_ASCII_STRING, len: 102
--------------------len: 102, value: insert into SQLSRV_TABLE (  USERNAME, INTEG
--------------------ER_VALUE, DOUBLE_VALUE, DATE_VALUE ) values ( ?, ?, ?, ? )
--------END OF MESSAGE

PROTOCOL LEVEL LOG CLIENT: read
----PACKET ID: 4, PACKET SEQUENCE: 0
--------SQLSRV_PREPARE ACK
------------STATEMENT ID
----------------SQLSRV_GENERALIZED_NUMBER, len: 7
--------------------len: 7, value: 1199896
------------PARAMETER MARKER SQLDA
----------------len: 2, value: 4
------------SQLVAR
----------------len: 2, value: 0
------------SQLTYPE
----------------len: 2, value: 129
------------SQLLEN
----------------len: 2, value: 33
------------SQLNAME
----------------SQLSRV_ASCII_STRING, len: 8
--------------------len: 8, value: USERNAME
------------SQLVAR
----------------len: 2, value: 1
------------SQLTYPE
----------------len: 2, value: 130
------------SQLLEN
----------------len: 2, value: 12
------------SQLNAME
----------------SQLSRV_ASCII_STRING, len: 13
--------------------len: 13, value: INTEGER_VALUE
------------SQLVAR
----------------len: 2, value: 2
------------SQLTYPE
----------------len: 2, value: 130
------------SQLLEN
----------------len: 2, value: 24
------------SQLNAME
----------------SQLSRV_ASCII_STRING, len: 12
--------------------len: 12, value: DOUBLE_VALUE
```

(continued on next page)

Example C–3 (Cont.) Sample Message Protocol Level Log

```
------------SQLVAR
----------------len: 2, value: 3
------------SQLTYPE
----------------len: 2, value: 131
------------SQLLEN
----------------len: 2, value: 17
------------SQLNAME
----------------SQLSRV_ASCII_STRING, len: 10
--------------------len: 10, value: DATE_VALUE
--------END OF MESSAGE

PROTOCOL LEVEL LOG CLIENT: write (logonly)
----PACKET ID: 5, PACKET SEQUENCE: 0
--------SQLSRV_EXECUTE
------------STATEMENT ID
----------------SQLSRV_GENERALIZED_NUMBER, len: 7
--------------------len: 7, value: 1199896
------------REPEAT COUNT
----------------len: 2, value: 1
------------PARAMETER MARKER DATA
----------------len: 2, value: 4
------------SQLVAR
----------------len: 2, value: 0
------------SQLDATA
----------------SQLSRV_ASCII_STRING, len: 6
--------------------len: 6, value: xxxxxx
------------SQLIND
----------------len: 2, value: 0
------------SQLVAR
----------------len: 2, value: 1
------------SQLDATA
----------------SQLSRV_GENERALIZED_NUMBER, len: 1
--------------------len: 1, value: 1
------------SQLIND
----------------len: 2, value: 0
------------SQLVAR
----------------len: 2, value: 2
------------SQLDATA
----------------SQLSRV_GENERALIZED_NUMBER, len: 10
--------------------len: 10, value: 128.000000
------------SQLIND
----------------len: 2, value: 0
------------SQLVAR
----------------len: 2, value: 3
------------SQLDATA
----------------SQLSRV_GENERALIZED_DATE, len: 8
--------------------len: 8, value: 19880701
------------SQLIND
----------------len: 2, value: 0
--------END OF MESSAGE
```

(continued on next page)

Example C-3 (Cont.) Sample Message Protocol Level Log

```
PROTOCOL LEVEL LOG CLIENT: read
----PACKET ID: 5, PACKET SEQUENCE: 0
--------SQLSRV_EXECUTE ACK
------------STATUS
----------------SQLSRV_GENERALIZED_NUMBER, len: 1
--------------------len: 1, value: 0
------------EXECUTE PARAMETER
----------------SQLSRV_GENERALIZED_NUMBER, len: 1
--------------------len: 1, value: 1
--------END OF MESSAGE

PROTOCOL LEVEL LOG CLIENT: write (logonly)
----PACKET ID: 6, PACKET SEQUENCE: 0
--------SQLSRV_EXECUTE
------------STATEMENT ID
----------------SQLSRV_GENERALIZED_NUMBER, len: 7
--------------------len: 7, value: 1199896
------------REPEAT COUNT
----------------len: 2, value: 1
------------PARAMETER MARKER DATA
----------------len: 2, value: 4
------------SQLVAR
----------------len: 2, value: 0
------------SQLDATA
----------------SQLSRV_ASCII_STRING, len: 6
--------------------len: 6, value: xxxxxx
------------SQLIND
----------------len: 2, value: 0
------------SQLVAR
----------------len: 2, value: 1
------------SQLDATA
----------------SQLSRV_GENERALIZED_NUMBER, len: 1
--------------------len: 1, value: 2
------------SQLIND
----------------len: 2, value: 0
------------SQLVAR
----------------len: 2, value: 2
------------SQLDATA
----------------SQLSRV_GENERALIZED_NUMBER, len: 12
--------------------len: 12, value: 32768.000000
------------SQLIND
----------------len: 2, value: 0
------------SQLVAR
----------------len: 2, value: 3
------------SQLDATA
----------------SQLSRV_GENERALIZED_DATE, len: 8
--------------------len: 8, value: 19880702
------------SQLIND
----------------len: 2, value: 0
--------END OF MESSAGE
```

(continued on next page)

Example C-3 (Cont.) Sample Message Protocol Level Log

```
PROTOCOL LEVEL LOG CLIENT: read
----PACKET ID: 6, PACKET SEQUENCE: 0
--------SQLSRV_EXECUTE ACK
------------STATUS
----------------SQLSRV_GENERALIZED_NUMBER, len: 1
--------------------len: 1, value: 0
------------EXECUTE PARAMETER
----------------SQLSRV_GENERALIZED_NUMBER, len: 1
--------------------len: 1, value: 1
--------END OF MESSAGE

PROTOCOL LEVEL LOG CLIENT: write (logonly)
----PACKET ID: 7, PACKET SEQUENCE: 0
--------SQLSRV_EXECUTE
------------STATEMENT ID
----------------SQLSRV_GENERALIZED_NUMBER, len: 7
--------------------len: 7, value: 1199896
------------REPEAT COUNT
----------------len: 2, value: 1
------------PARAMETER MARKER DATA
----------------len: 2, value: 4
------------SQLVAR
----------------len: 2, value: 0
------------SQLDATA
----------------SQLSRV_ASCII_STRING, len: 6
--------------------len: 6, value: xxxxxx
------------SQLIND
----------------len: 2, value: 0
------------SQLVAR
----------------len: 2, value: 1
------------SQLDATA
----------------SQLSRV_GENERALIZED_NUMBER, len: 1
--------------------len: 1, value: 3
------------SQLIND
----------------len: 2, value: 0
------------SQLVAR
----------------len: 2, value: 2
------------SQLDATA
----------------SQLSRV_GENERALIZED_NUMBER, len: 13
--------------------len: 13, value: 524288.000000
------------SQLIND
----------------len: 2, value: 0
------------SQLVAR
----------------len: 2, value: 3
------------SQLDATA
----------------SQLSRV_GENERALIZED_DATE, len: 8
--------------------len: 8, value: 19880703
------------SQLIND
----------------len: 2, value: 0
--------END OF MESSAGE
```

Index

How to Order Additional Documentation

Technical Support

If you need help deciding which documentation best meets your needs, call 800-343-4040 before placing your electronic, telephone, or direct mail order.

Electronic Orders

To place an order at the Electronic Store, dial 800-DEC-DEMO (800-332-3366) using a 1200- or 2400-baud modem. If you need assistance using the Electronic Store, call 800-DIGITAL (800-344-4825).

Telephone and Direct Mail Orders

Your Location	**Call**	**Contact**
Continental USA, Alaska, or Hawaii	800-DIGITAL	Digital Equipment Corporation P.O. Box CS2008 Nashua, New Hampshire 03061
Puerto Rico	809-754-7575	Local DIGITAL subsidiary
Canada	800-267-6215	Digital Equipment of Canada Attn: DECdirect Operations KAO2/2 P.O. Box 13000 100 Herzberg Road Kanata, Ontario, Canada K2K 2A6
International	————	Local DIGITAL subsidiary or approved distributor
Internal[1]	————	SDC Order Processing - WMO/E15 *or* Software Distribution Center Digital Equipment Corporation Westminster, Massachusetts 01473

[1] For internal orders, you must submit an Internal Software Order Form (EN-01740-07).

Reader's Comments

VAX Rdb/VMS
Guide to Using SQL/Services
AA–ND79A–TE

Please use this form to comment on this manual. If you require a written reply to a software problem and are eligible to receive one under Software Performance Report (SPR) service, submit your comments on an SPR form.

Thank you for your assistance.

I rate this manual's:	Excellent	Good	Fair	Poor
Accuracy (software works as manual says)	□	□	□	□
Completeness (enough information)	□	□	□	□
Clarity (easy to understand)	□	□	□	□
Organization (structure of subject matter)	□	□	□	□
Figures (useful)	□	□	□	□
Examples (useful)	□	□	□	□
Index (ability to find topic)	□	□	□	□
Page layout (easy to find information)	□	□	□	□

I would like to see more/less ____________________

____

What I like best about this manual is ____________________

____

What I like least about this manual is ____________________

____

I found the following errors in this manual:

Page Description

____ ____________________

____ ____________________

____ ____________________

Additional comments or suggestions to improve this manual:

____

____

____

I am using **Version** ______ of the software this manual describes.

Name/Title ____________________ Dept. __________

Company ____________________ Date __________

Mailing Address ____________________

____________________ Phone __________

Fold Here and Tape

digital™

Please
Affix Stamp
Here

DIGITAL EQUIPMENT CORPORATION
Corporate User Publications
200 Forest Street
MRO1–3/L12
Marlborough, MA 01752–9101

Fold Here

Cut Along Dotted Line

Reader's Comments

VAX Rdb/VMS
Guide to Using SQL/Services
AA–ND79A–TE

Please use this form to comment on this manual. If you require a written reply to a software problem and are eligible to receive one under Software Performance Report (SPR) service, submit your comments on an SPR form.

Thank you for your assistance.

I rate this manual's:	Excellent	Good	Fair	Poor
Accuracy (software works as manual says)	□	□	□	□
Completeness (enough information)	□	□	□	□
Clarity (easy to understand)	□	□	□	□
Organization (structure of subject matter)	□	□	□	□
Figures (useful)	□	□	□	□
Examples (useful)	□	□	□	□
Index (ability to find topic)	□	□	□	□
Page layout (easy to find information)	□	□	□	□

I would like to see more/less ______________________________

____

What I like best about this manual is ______________________________

____

What I like least about this manual is ______________________________

____

I found the following errors in this manual:
Page Description

______ ______________________________

______ ______________________________

______ ______________________________

Additional comments or suggestions to improve this manual:

____

____

____

I am using **Version** ______ of the software this manual describes.
Name/Title ______________________ Dept. ____________
Company ______________________ Date ____________
Mailing Address ______________________________
______________________ Phone ____________

Fold Here and Tape

digital™

Please
Affix Stamp
Here

DIGITAL EQUIPMENT CORPORATION
Corporate User Publications
200 Forest Street
MRO1–3/L12
Marlborough, MA 01752–9101

Fold Here

Cut Along Dotted Line