Click here to visit our Web Site

Mimer SQL

Programmer's Manual

Version 8.2

Mimer SQL version 8.2 Programmer's Manual
Second revised edition

December, 2000

Published by Mimer Information Technology AB,
P.O. Box 1713,
SE-751 47 Uppsala, Sweden.
Tel +46(0)18-18 50 00.
Fax +46(0)18-18 51 00.
Internet: http://www.mimer.com

Produced by Mimer Information Technology AB, Uppsala, Sweden.

FOREWORD

Documentation objectives

This manual describes how SQL statements may be embedded in application programs written in conventional host languages. It also describes how to create and use Stored Procedures and Functions (see Chapter 8) as well as Triggers (see Chapter 9).

Intended audience

The manual is intended for application developers working with Mimer SQL.

Prerequisites

Application developers using this manual are assumed to have a working acquaintance with the principles of the relational database model in general and of Mimer SQL in particular.

It is also assumed that programmers writing embedded SQL applications are familiar with the principles of the SQL database management language. Knowledge of Mimer SQL is of course an advantage, although experience with other standard-compliant SQL implementations will suffice. Experience of Mimer SQL is best gained through interactive use of the BSQL facility (see the Mimer SQL User's Manual).

Competence in at least one of the host programming languages supporting embedded Mimer SQL (i.e. C/C++, COBOL or FORTRAN) is assumed.

Organization of this manual

The organization of the material in this manual reflects the general requirements for writing application programs using embedded SQL:

Chapter 1 introduces embedded SQL in relation to the other Mimer SQL products.

Chapter 2 presents the basic principles of embedding SQL in application programs and describes features of preprocessing and compiling embedded SQL programs.

Chapter 3 describes the way in which embedded SQL communicates with the host program through host variables.

Chapter 4 describes how to log in to the database from application programs and how to make use of program idents.

Chapter 5 describes how to access data in the database tables; how to retrieve data and how to change table contents.

Chapter 6 describes the essential features of transaction handling.

Chapter 7 describes the special features of dynamic SQL, which allow an application program to process SQL statements entered by the user at run-time.

Chapter 8 describes stored functions, modules and procedures.

Chapter 9 describes triggers, which define a set of procedural SQL statements that are to be executed when a specified data manipulation operation occurs on a named table or view.

Chapter 10 describes how to handle errors and exception conditions.

Appendix A-D describe host language dependent aspects of embedded SQL and preprocessors, SQL return codes and program examples.

Related Mimer SQL publications

- **Mimer SQL Reference Manual** contains a complete description of the syntax and usage of all statements in Mimer SQL and is a necessary complement to this manual.
- **Mimer SQL User's Manual** contains a description of the BSQL facilities. A user-oriented guide to the SQL statements is also included, which may provide help for less experienced users in formulating statements correctly (particularly the SELECT statement, which can be quite complex).
- **Mimer SQL System Management Handbook** describes system administration functions, including export/import, backup/restore, databank shadowing and the statistics functionality. The information in this manual is used primarily by the system administrator, and is not required by application program developers. The SQL statements which are part of the System Management API are described in the Mimer SQL Reference Manual.

- **Mimer SQL platform-specific documents** containing platform-specific information. A set of one or more documents is provided, where required, for each platform on which Mimer SQL is supplied.
- **Mimer SQL Release Notes** contain general and platform-specific information relating to the Mimer SQL release for which they are supplied.

Suggestions for further reading

We can recommend to users of Mimer SQL the many works of C. J. Date. His insight into the potential and limitations of SQL, coupled with his pedagogical talents, makes his books invaluable sources of study material in the field of SQL theory and usage. In particular, we can mention:

A Guide to the SQL Standard (Fourth Edition, 1997). ISBN: 0-201-96426-0. This work contains much constructive criticism and discussion of the SQL standard, including SQL99.

For JDBC users:

JDBC information can be found on the internet at the following web addresses: http://java.sun.com/products/jdbc/ and http://www.mimer.com/jdbc/.

For information on specific JDBC methods, please see the online documentation for the java.sql package. This documentation is normally included in the Java development environment.

JDBC™ API Tutorial and Reference, 2nd edition. ISBN: 0-201-43328-1. A useful book published by JavaSoft.

For ODBC users:

Microsoft ODBC 3.0 Programmer's Reference and SDK Guide for Microsoft Windows and Windows NT. ISBN: 1-57231-516-4. This manual contains information about the Microsoft Open Database Connectivity (ODBC) interface, including a complete API reference.

Official documentation of the accepted SQL standards may be found in:

ISO/IEC 9075:1999(E) Information technology - Database languages - SQL. This document contains the standard referred to as SQL99.

ISO/IEC 9075:1992(E) Information technology - Database languages - SQL. This document contains the standard referred to as SQL92.

ISO/IEC 9075-4:1996(E) Database Language SQL - Part 4: Persistent Stored Modules (SQL/PSM). This document contains the standard which specifies the syntax and semantics of a database language for managing and using persistent routines.

CAE Specification, Data Management: Structured Query Language (SQL), Version 2. X/Open document number: C449. ISBN: 1-85912-151-9. This document contains the X/Open-95 SQL specification.

Definitions, terms and trademarks

ANSI	American National Standards Institute, Inc
API	Application Programming Interface
BSQL	The Mimer SQL facility for using SQL interactively or by running a command file
ESQL	The preprocessor for embedded Mimer SQL
IEC	International Electrotechnical Commission
ISO	International Standards Organization
JDBC	The Java database API specified by Sun Microsystems, Inc
ODBC	Microsoft's Open Database Connectivity
PSM	Persistent Stored Modules, the term used by ISO/ANSI for Stored Procedures
SQL	Structured Query Language
X/Open	X/Open is a trademark of the X/Open Company

CONTENTS

1 INTRODUCTION

2 CREATING MIMER SQL APPLICATIONS
2.1 Using the ODBC database API 2-1
2.2 Using the JDBC database API 2-1
2.3 Using embedded Mimer SQL 2-2
2.3.1 The scope of embedded Mimer SQL 2-2
2.3.2 General principles for embedding SQL statements 2-2
2.3.2.1 Host languages 2-2
2.3.2.2 Identifying SQL statements 2-3
2.3.2.3 Included code 2-3
2.3.2.4 Comments 2-4
2.3.2.5 Recommendations 2-4
2.3.3 Processing embedded SQL 2-4
2.3.3.1 Preprocessing - the ESQL command 2-4
2.3.3.2 Processing embedded SQL - the compiler 2-7
2.3.4 Essential program structure 2-7
2.4 Linking applications 2-9

3 COMMUNICATING WITH THE APPLICATION PROGRAM
3.1 Host variable usage 3-1
3.1.1 Declaring host variables 3-1
3.1.2 Using variables in statements 3-2
3.1.3 Indicator variables 3-2
3.2 The SQLSTATE variable 3-3
3.3 The diagnostics area 3-4
3.4 The SQL Descriptor Area 3-4

4 IDENTS AND DATABASE CONNECTIONS
4.1 Mimer SQL idents 4-1
4.2 Access to the database 4-2
4.3 Connecting to a database 4-3
4.3.1 Connecting 4-3
4.3.2 Changing connection 4-4
4.3.3 Disconnecting 4-4
4.3.4 Program idents - ENTER and LEAVE 4-5

5 ACCESSING DATABASE OBJECTS
5.1 Retrieving data using cursors 5-1
5.1.1 Declaring host variables 5-2
5.1.2 Declaring the cursor 5-2
5.1.3 Opening the cursor 5-3
5.1.4 Retrieving data 5-3
5.1.5 Closing the cursor 5-4
5.2 Retrieving single rows 5-5
5.3 Retrieving data from multiple tables 5-6

5.3.1 The 'Parts explosion' problem 5-7
5.4 Entering data into tables 5-8
5.4.1 Cursor-independent operations 5-8
5.4.2 Updating and deleting through cursors 5-9

6 TRANSACTION HANDLING AND DATABASE SECURITY
6.1 Transaction principles 6-1
6.1.1 Concurrency control 6-1
6.1.2 Locking 6-3
6.2 Transaction control statements 6-3
6.2.1 Starting transactions 6-3
6.2.2 Ending transactions 6-4
6.2.3 Optimizing transactions 6-5
6.2.4 Consistency within transactions 6-5
6.2.5 Exception diagnostics within transactions 6-6
6.2.6 Setting default transaction options 6-6
6.2.7 Statements in transactions 6-7
6.2.8 Cursors in transactions 6-8
6.2.9 Error handling in transactions 6-9
6.3 Transactions and logging 6-10
6.4 Protection against data loss 6-11
6.4.1 System interruptions 6-11
6.4.2 Hardware failure 6-11

7 DYNAMIC SQL
7.1 Principles of dynamic SQL 7-1
7.2 General summary of dynamic SQL processing 7-3
7.3 SQL descriptor area 7-4
7.3.1 The structure of the SQL descriptor area 7-4
7.4 Preparing statements 7-5
7.5 Extended dynamic cursors 7-5
7.6 Describing prepared statements 7-6
7.6.1 Describing output variables 7-7
7.6.2 Describing input variables 7-7
7.7 Handling prepared statements 7-8
7.7.1 Executable statements 7-8
7.7.2 Result set statements 7-9
7.8 Example framework for dynamic SQL programs 7-10

8 PERSISTENT STORED MODULES
8.1 Routines 8-1
8.1.1 Functions 8-2
8.1.2 Procedures 8-3
8.2 Syntactic components of a routine definition 8-5
8.2.1 Routine parameters 8-5
8.2.2 Routine language indicator 8-6
8.2.3 Routine deterministic clause 8-6
8.2.4 Routine access clause 8-6
8.2.5 Scope in routines - the compound SQL statement 8-6
8.2.5.1 The ATOMIC compound SQL statement 8-7
8.2.6 Declaring variables 8-9
8.2.7 The ROW data type 8-10
8.2.7.1 ROW data type syntax 8-10
8.2.7.2 Using the ROW data type 8-11
8.2.8 Row value expression 8-12
8.3 Modules 8-12
8.4 SQL constructs in routines 8-13
8.4.1 Assignment using SET 8-13

8.4.2 Conditional execution using IF....8-13
8.4.3 Conditional execution - the CASE statement....8-14
8.4.4 Iteration using LOOP....8-16
8.4.5 Iteration using WHILE....8-16
8.4.6 Iteration using REPEAT....8-16
8.4.7 Invoking procedures - CALL....8-17
8.4.8 Invoking functions - use as a value expression....8-17
8.4.9 Comments in routines....8-18
8.4.10 Restrictions....8-18
8.5 Data manipulation....8-19
8.5.1 Write operations....8-19
8.5.2 Using cursors....8-19
8.5.3 SELECT INTO....8-21
8.5.4 Transactions....8-21
8.6 Result set procedures....8-21
8.7 Managing exception conditions....8-23
8.7.1 Declaring condition names....8-25
8.7.2 Declaring exception handlers....8-25
8.8 Access rights....8-28
8.9 Using DROP and REVOKE....8-29

9 TRIGGERS
9.1 Creating a trigger....9-1
9.2 Trigger time....9-2
9.3 Trigger event....9-2
9.4 Trigger action....9-3
9.4.1 New and old table aliases....9-4
9.4.2 Altered table rows....9-4
9.4.3 Recursion....9-5
9.5 Comments on triggers....9-5
9.6 Using DROP and REVOKE....9-5

10 HANDLING ERRORS AND EXCEPTION CONDITIONS
10.1 Syntax errors....10-1
10.2 Semantic errors....10-1
10.3 Run-time errors....10-2
10.3.1 Testing for run-time errors and exception conditions....10-2

A HOST LANGUAGE DEPENDENT ASPECTS
A.1 Embedded SQL in C/C++ programs....A-2
A.1.1 SQL statement format....A-2
A.1.2 Host variables....A-2
A.1.3 Preprocessor output format....A-5
A.1.4 Scope rules....A-6
A.2 Embedded SQL in COBOL programs....A-7
A.2.1 SQL statement format....A-7
A.2.2 Restrictions....A-8
A.2.3 Host variables....A-8
A.2.4 Preprocessor output format....A-10
A.2.5 Scope rules....A-10
A.3 Embedded SQL in FORTRAN programs....A-11
A.3.1 SQL statement format....A-11
A.3.2 Host variables....A-12
A.3.3 Preprocessor output format....A-14
A.3.4 Scope rules....A-14

B RETURN CODES
B.1 SQLSTATE return codes....B-2

B.2 Internal Mimer SQL return codes B-4
B.2.1 Warnings and messages B-4
B.2.2 ODBC errors B-5
B.2.3 Data-dependent errors B-7
B.2.4 Limits exceeded B-9
B.2.5 SQL statement errors B-10
B.2.6 Program-dependent errors B-19
B.2.7 Databank and table errors B-22
B.2.8 Miscellaneous errors B-24
B.2.9 Internal errors B-27
B.2.10 Communication errors B-31
B.2.11 JDBC errors B-34

C DEPRECATED FEATURES
C.1 INCLUDE SQLCA C-1
C.2 SQLCODE C-1
C.3 SQLDA C-1
C.4 Parameter marker representation C-2
C.5 VARCHAR(size) C-2
C.6 SET TRANSACTION C-2
C.7 DBERM4 ..... C-2

D APPLICATION PROGRAM EXAMPLES
D.1 EXAMPLE program D-2
D.1.1 Building the EXAMPLE program D-2
D.1.2 C source code for the EXAMPLE program D-3
D.1.3 COBOL source code for the EXAMPLE program D-5
D.1.4 FORTRAN source code for the EXAMPLE program D-7
D.2 DSQLSAMP program using dynamic SQL D-9
D.2.1 Building the DSQLSAMP program D-9
D.2.2 Source code for the DSQLSAMP program (in C) D-10
D.3 FREQCALL program using dynamic SQL and a stored procedure D-29
D.3.1 Building the FREQCALL program D-29
D.3.2 C source code for the FREQCALL program D-30
D.3.3 COBOL source code for the FREQCALL program D-32
D.3.4 FORTRAN source code for the FREQCALL program D-35
D.4 WAKECALL program using dynamic SQL and a result-set procedure D-37
D.4.1 Building the WAKECALL program D-37
D.4.2 C source code for the WAKECALL program D-38
D.4.3 COBOL source code for the WAKECALL program D-40
D.4.4 FORTRAN source code for the WAKECALL program D-42
D.5 BLOBSAMP program using Dynamic SQL for handling binary data D-45
D.5.1 Building the BLOBSAMP program D-46
D.5.2 Source code for the BLOBSAMP program (in C) D-47
D.6 OSQLSAMP program using dynamic SQL D-55
D.6.1 Building the OSQLSAMP program D-56
D.6.2 Source code for the OSQLSAMP program (in C) D-58

1 INTRODUCTION

Mimer SQL is an advanced relational database management system developed by Mimer Information Technology AB. The database management language Mimer SQL is compatible in all essential features with the established SQL standards (see the Mimer SQL Reference Manual for details).

The information contained in this handbook generally applies to all the platforms supported by Mimer SQL.

From time to time platform-specific notes appear in the general description, presented as follows:

Denotes information that applies specifically to Unix platforms.

Denotes information that applies specifically to VMS platforms.

Denotes information that applies specifically to Windows platforms.

Mimer SQL is accessed through the following user interfaces:

- Embedded SQL is used through a host programming language (C/C++, COBOL or FORTRAN as available on the host computer). SQL statements are included as part of the source code for an application program, which is compiled and linked with the appropriate language-specific facilities. The SQL statements are executed in the context of the application program.
- BSQL is a line-oriented interface designed for interactive use or running command files.
- ODBC is a database independent interface specified by Microsoft. Through ODBC, Mimer SQL can support many of the tools available on the platforms supporting ODBC (e.g. Windows, Unix).
- JDBC is the Java database API specified by Sun Microsystems Inc.

This manual describes the usage of SQL embedded in application programs, and provides, together with the Mimer SQL Reference Manual, the complete reference material for Mimer SQL. The BSQL facilities are described in the Mimer SQL User's Manual.

For details on how to read the syntax diagrams which appear in this manual see Section 1.1 of the Mimer SQL Reference Manual.

Note: To simplify the description in this manual, pseudo code is used in all examples. Details pertinent to real host programming languages may be found in Appendix A. The structure of the pseudo code is largely self-explanatory. No distinction is made between integers of different lengths. Only statements essential to the illustration at hand are shown in the examples: many statements required in actual application programs are omitted.

2 CREATING MIMER SQL APPLICATIONS

An application that accesses a Mimer SQL database server can be created by:

- Using the ODBC database API with the C/C++ host language
- Using the JDBC database API with the Java language
- Embedding Mimer SQL in one of the supported host programming languages.

2.1 Using the ODBC database API

Unix

Information on the availability and use of Driver Managers for ODBC on Unix platforms can be provided by your Mimer SQL distributor. The library to specify as the ODBC driver when defining a Mimer ODBC data source is **libmimer**. An ODBC SDK (various versions available) is also required in order to develop applications.

VMS

An ODBC driver is supplied with Mimer SQL on VMS and client applications can link with it to use ODBC on VMS platforms. The Driver Manager for ODBC on VMS is not yet available. For further assistance, contact your Mimer SQL representative.

A Mimer SQL database server running on a VMS node can always be accessed remotely by a client application using ODBC from another type of platform via the client/server interface.

Win

When a Mimer SQL client is installed on a Windows platform, the ODBC driver manager and other resources needed to use ODBC are also installed. The ODBC SDK (available from Microsoft) is also required in order to develop applications.

2.2 Using the JDBC database API

The Mimer JDBC Driver is required in order to access a Mimer SQL database over TCP/IP from a Java application.

The JDBC driver can be used on any client computer which has the Java runtime environment is installed. Note that only Mimer SQL servers of version 8.2 or later support JDBC clients.

2.3 Using embedded Mimer SQL

2.3.1 The scope of embedded Mimer SQL

The following groups of SQL statements are common to embedded SQL and interactive SQL:

- Data manipulation statements for reading or changing the content of the database and invoking stored routines. These are basically similar between interactive SQL and embedded SQL, but differ in certain details as a result of the different environments in which the statements are used.
- Procedure control statements for performing operations that occur specifically within stored routines and triggers.
- Transaction control statements for grouping database operations in transactions (indivisible units of work).
- Access control statements for allocating privileges and access rights to users of the system. These are identical between interactive SQL and embedded SQL.
- Data definition statements for creating and altering objects in the database. These are identical between interactive SQL and embedded SQL.
- Connection statements for identifying the current user of the system.
- System administration statements for controlling the availability of the database and its physical components, managing backups and updating database statistics.
- Declarations for variables, conditions, cursors and handlers occurring within stored routines or triggers.

There are a number of commands provided for use with BSQL which are not included in the Mimer SQL interface, these are described in Chapter 9 of the *Mimer SQL User's Manual*.

Note: In the Mimer SQL Reference Manual, Mimer SQL statements are identified as valid for use in embedded SQL, for interactive use or both (this is specified in the "Usage" section of the statement description).

2.3.2 General principles for embedding SQL statements

2.3.2.1 Host languages

Statements in Mimer SQL may be embedded in application programs written in C/C++, COBOL or FORTRAN. The basic principles for writing embedded SQL programs are the same in all languages and all embedded SQL statements are embedded in the same way.

Information given in this manual applies to all languages unless otherwise explicitly stated. Language-specific information is detailed in Appendix A.

The ESQL preprocessor is used to process SQL statements embedded in a host language.

Unix

Mimer SQL supports an ESQL preprocessor for the C/C++ host language on Unix platforms.

VMS

Mimer SQL supports an ESQL preprocessor for the COBOL, FORTRAN and C/C++ host languages on VMS platforms.

Win

Mimer SQL supports an ESQL preprocessor for the C/C++ host language on Windows platforms.

2.3.2.2 Identifying SQL statements

SQL statements are included in the host language source code exactly as though they were ordinary host language statements (i.e. they follow the same rules of conditional execution, etc., which apply to the host language).

SQL statements are identified by the leading keywords EXEC SQL (in all host languages) and are terminated by a language-specific delimiter. Every separate SQL statement must be delimited in this way. Blocks of several statements may not be written together within one set of delimiters. For instance, two consecutive DELETE statements must be written (in COBOL) as:

```
EXEC SQL DELETE FROM HOTEL   END-EXEC.
EXEC SQL DELETE FROM ROOMS   END-EXEC.
```

and not

```
EXEC SQL DELETE FROM HOTEL
         DELETE FROM ROOMS   END-EXEC.
```

Single SQL statements can however be split over several lines, following the host language rules for line continuation. The following embedded statement is thus acceptable in a FORTRAN program (the continuation mark is a "+" in column 6 on the second line):

```
      EXEC SQL DELETE FROM HOTEL
     +            WHERE CITY = 'SAN FRANCISCO'
```

The keywords "EXEC SQL" may not be split over more than one line.

2.3.2.3 Included code

Any code which is included in the program by the host language compiler (as directed by host language INCLUDE statements) is not recognized by the SQL preprocessor. If external source code modules containing SQL statements are to be included in the program, the non-standard SQL INCLUDE statement must be used:

```
EXEC SQL INCLUDE filename
```

Files included in this way are physically integrated into the output from the preprocessor.

Note: The file name must be enclosed in SQL string delimiters if it contains any non-alphanumeric characters.

2.3.2.4 Comments

Comments may be written in the embedded SQL program according to the rules for writing comments in the host language. Thus comments may be written within an SQL statement if the host language accepts comments within host language statements. The following statement is valid in C/C++:

```
exec sql DELETE FROM HOTEL /* Hotel was closed */
                WHERE CITY = 'SAN FRANCISCO';
```

The keywords “EXEC” and “SQL” may not be separated by a comment.

2.3.2.5 Recommendations

The following recommendations are imposed by the use of embedded SQL:

- Variable names beginning with the letters “SQL” should be avoided (except for SQLSTATE and SQLCODE, which should be used when appropriate).
- Subroutine or subprogram names ending with a number should be avoided.

Language-specific restrictions are described in Appendix A.

2.3.3 Processing embedded SQL

2.3.3.1 Preprocessing - the ESQL command

An application program using embedded SQL statements must first be preprocessed using the ESQL command before it can be passed through the host language compiler, since the host language itself does not recognize the embedded SQL syntax. Preprocessors are available for the host languages supported on each platform (see Section 2.3.2.1).

The input to the preprocessor is thus a source code file containing host language statements and embedded SQL statements. The output from the preprocessor is a source code file in the same host language, with the embedded SQL statements converted to source code data assignment statements and subroutine calls which pass the SQL statements to the Mimer SQL database manager. The original embedded SQL statements are retained as comments in the output file, to help in understanding the program if a source code debugger is used.

The output from the preprocessor is human-readable source code, still retaining a large part of the structure and layout of the original program, which is used as input to the appropriate host language compiler to produce object code.

The default file extensions for preprocessor input and output files depend on the host language used and are shown in the table below:

Language	Input file extension	Output file extension
C	.ec	.c
COBOL	.eco	.cob
FORTRAN	.efo	.for

The preprocessor is invoked by the following command:

```
$ esql language [flagger] [options] input-file [output-file]
```

***language* flags**

Unix-style	VMS-style	Function
-c	**/C**	Indicates that the input file is written using the C/C++ host language.
-cob(ol)	**/COBOL**	Indicates that the input file is written using the COBOL host language.
-for(tran)	**/FORTRAN**	Indicates that the input file is written using the FORTRAN host language.

***flagger* flags**

Unix-style	VMS-style	Function
-e(ntry)	**/ENTRY**	Ensures that all SQL statements which are beyond the bounds of the entry level SQL2 standard will generate a preprocessor warning and will be flagged with a warning comment in the output file.
-t(ransitional)	**/TRANSITIONAL**	Ensures that all SQL statements which are beyond the bounds of the transitional SQL2 standard will generate a preprocessor warning and will be flagged with a warning comment in the output file.
-i(ntermediate)	**/INTERMEDIATE**	Ensures that all SQL statements which are beyond the bounds of the intermediate level SQL2 standard will generate a preprocessor warning and will be flagged with a warning comment in the output file.
-f(ull)	**/FULL**	Ensures that all SQL statements which are beyond the bounds of the full level SQL2 standard will generate a preprocessor warning and will be flagged with a warning comment in the output file.

***options* flags**

Unix-style	VMS-style	Function
-s(ilent)	**/SILENT**	Suppresses the display of the copyright message and input file name on the screen (warnings and errors are **always** displayed on the screen).
-l(ine)	**/LINE**	Generates #line preprocessing directives for source written in the C language. These force the C compiler to produce diagnostic messages with line numbers relating to the input C source code rather than the code generated by the preprocessor (and thus compiled by the C compiler).

***input-file* and *output-file* flags**

Unix-style	VMS-style	Function
filename	***filename***	The ***input-file*** containing the source code to be preprocessed. If no file extension is specified, the appropriate file extension for the source language is assumed (previously described in this section).
filename	**/OUTPUT=*filename***	The ***output-file*** which will contain the compiler source code generated by the preprocessor. If not specified, the output file will have the same name as the input file, but with the appropriate default output file extension (previously described in this section).

Note: The application programmer should never attempt to directly modify the output from the preprocessor. Any changes which may be required in a program should be introduced into the original embedded SQL source code. Mimer Information Technology AB cannot accept any responsibility for the consequences of modifications to the preprocessed code.

The preprocessor checks the syntax and to some extent the semantics of the embedded SQL statements (see Chapter 10 for a more detailed discussion of how errors are handled). Syntactically invalid statements cannot be preprocessed and the source code must be corrected.

The preprocessor also places comments in the generated output to indicate the SQL2 standard conformance level for each SQL statement (entry-level, transitional, intermediate or full). If a *flagger* option (see preceding table) has been specified, then an SQL statement that is beyond the bounds of the specified conformance level is flagged with a warning comment in the preprocessor output and a warning on the screen. Mimer SQL extensions are always flagged with a warning comment and generate a preprocessor warning.

2.3.3.2 Processing embedded SQL - the compiler

The output from the ESQL preprocessor is compiled in the usual way using the appropriate host language compiler, and linked with the appropriate routine libraries.

Unix

On Unix platforms, the C compiler defined for the **MIM_CC** symbol in the **makeopt** file found in the installation examples directory is supported.

VMS

The following compilers (all sold by Compaq) are supported on the VMS platform:

- DEC C
- DEC FORTRAN
- DEC COBOL

Note: For COBOL, the source program must be formatted according to the ANSI rules. Use the /ANSI option when compiling the resulting COBOL program.

Win

On Windows platforms, the C compiler identified by the **cc** symbol in the file **.\dev\makefile.mak** below the installation directory is supported.

Other compilers, from other software distributors, may or may not be able to compile the ESQL preprocessor output. Mimer SQL cannot guarantee the result of using a compiler that is not supported.

At run-time, database management requests are passed to the SQL compiler responsible for implementing the SQL functions in the application program.

The SQL compiler performs two functions:

- SQL statements are checked semantically against the data dictionary.
- Operations performed against the database are optimized (i.e. internal routines determine the most efficient way to execute the SQL request, with regard to the existence of secondary indexes and the number of rows in the tables addressed by the statement). The programmer does not need to worry, for instance, about the order in which tables are addressed in a complex selection condition. This optimization process is completely transparent.

Note: Since all SQL statements are compiled at run-time, there can be no conflict between the state of the database at the times of compilation and execution. Moreover, the execution of SQL statements is always optimized with reference to the current state of the database.

2.3.4 Essential program structure

All application programs using embedded Mimer SQL must include certain basic components, summarized below in the order in which they appear in a program.

- Host variable declarations. Any host variables used in SQL statements must be declared inside a so-called SQL DECLARE SECTION. This is described in more detail in Chapter 3.

- The status information variable SQLSTATE, if used, must be declared inside the SQL DECLARE SECTION. This variable provides the application with status information for the most recently executed SQL statement.
- Executable SQL statements. This is the "body" of the program, and performs the required operations on the database. Normally, these begin with connecting to Mimer SQL and performing the required transactions before finally disconnecting from Mimer SQL.

The following table summarizes the functions for data manipulation in interactive SQL and embedded SQL.

Operation	**Interactive SQL**	**Embedded SQL**
Retrieve data	SELECT generates a result table directly	SELECT is used to declare a cursor. The cursor must be opened and positioned. Data is retrieved into host variables one row at a time with FETCH. Alternative: SELECT INTO retrieves a single-row result set directly into host variables
Update data	UPDATE operates on a set of rows or columns	UPDATE operates on a set of rows or columns UPDATE CURRENT operates on a single row through a cursor
Insert data	INSERT inserts one or many rows at a time	INSERT inserts one or many rows at a time
Delete data	DELETE operates on a set of rows	DELETE operates on a set of rows DELETE CURRENT operates on a single row through a cursor

Operation	**Interactive SQL**	**Embedded SQL**
Invoke routine	CALL is used to execute all stored procedures, i.e. both result set and non-result set procedures are handled the same way Functions can be specified where an expression could be used and are invoked when an expression used in the same context would be evaluated	Result set procedures are called by using the CALL clause in a cursor declaration and then using FETCH. The CALL statement is used directly for non-result set procedures Functions can be specified where an expression could be used and are invoked when an expression used in the same context would be evaluated

Many SQL statements (e.g. data definition statements) are simply embedded in their logical place in the application program and are executed without direct reference to other parts of the program. Some features of embedded SQL however require special consideration, and are dealt with in detail in the chapters that follow:

- Access authorization through the use of user and program idents.
- Data manipulation statements which require the use of cursors (FETCH, UPDATE CURRENT, DELETE CURRENT). These together with cursor handling statements are probably the most commonly used statements in embedded SQL.
- Transaction control, which is essential for a consistent database.
- Dynamic SQL, which is a special set of statements allowing an application program to process SQL statements entered by the user at run-time.
- Exception handling, which controls the action taken when, for instance, the end of a result set is reached.

2.4 Linking applications

Unix

The makefile called **ex_makefile**, found in the installation examples directory, provides a verified example of the recommended way to build applications on Unix platforms. Applications built using the procedure contained in this makefile will reference the Mimer SQL shared library called **libmimer**.

VMS

All Mimer SQL applications should be linked with the options file MIMER.OPT, as shown in the following example:

```
$ LINK main,MIMLIB8:MIMER/OPT
```

The MIMER.OPT file includes the following:

- MIMLIB8:LR.OLB (object library)
- MIMLIB8:MDR.OLB (object library)
- MIMLIB8:MIMDB8.EXE (shareable library)

If an image linked in this fashion is activated, it will translate the logical name MIMDB8 to get the name of the Mimer SQL shareable library to be used. The logical name will have been defined by the MIMSETUP8 command procedure.

Win

The example makefile **.\dev\makefile.mak** below the installation directory should be copied and used in the recommended way to build applications on Windows platforms.

If applications are linked as recommended above to reference the Mimer SQL shared library, they will automatically use a new version of Mimer SQL when it is installed, without having to be re-linked.

3 COMMUNICATING WITH THE APPLICATION PROGRAM

Information is transferred between the embedded SQL application program and the Mimer SQL database manager in four ways:

- through host variables used in SQL statements
- through the status variable SQLSTATE
- through the diagnostics area, accessed by the SQL statement GET DIAGNOSTICS
- through an SQL descriptor area (dynamic area).

3.1 Host variable usage

Host variables are used in SQL statements to pass values between the database and the application program.

3.1.1 Declaring host variables

All variables used in SQL statements must be declared for the preprocessor, by enclosing the variable declarations between the SQL statements BEGIN DECLARE SECTION and END DECLARE SECTION. Any variables declared outside the SQL DECLARE SECTION will not be recognized by the preprocessor. Variables are declared within the section using the normal host language syntax; for instance the following example in C declares only the character variables **user** and **passw** for use in SQL statements:

```
int rc,
    pf,
    count;
exec sql BEGIN DECLARE SECTION;
char user[129],
     passw[19];
exec sql END DECLARE SECTION;
```

Variables which are not used in SQL statements may also be declared in the SQL DECLARE SECTION. This will however extend the symbol table established by the preprocessor more than is necessary.

The use of array variables is currently not supported in embedded Mimer SQL (except for character string variables).

3.1.2 Using variables in statements

Host variables may be used:

- to receive information from the database (SELECT INTO, FETCH, CALL and SET statements)
- to assign values to columns in the database (CALL, INSERT and UPDATE statements)
- to manipulate information taken from the database or contained in other variables (in expressions)
- to get descriptor and diagnostics information (GET DESCRIPTOR, SET DESCRIPTOR and GET DIAGNOSTICS)
- in dynamic SQL statements

In all these contexts, the data type of the host variable or database column must be compatible with the data type of the corresponding database value or host variable. General considerations of data type compatibility may be found in Chapter 4 of the Mimer SQL Reference Manual. Host language specific aspects are described in Appendix A of this manual.

If you have an INTEGER column containing values that do not fit into the largest integer variable allowed on your machine (remember that Mimer SQL supports INTEGER values with a precision of up to 45 digits), you can, for example, use a character string or float host variable for that column. In this case, Mimer SQL automatically performs the necessary conversions.

Host variable names are preceded by a colon when used in SQL statements (see the Mimer SQL Reference Manual).

Note: The colon is not part of the host variable name, and should not be used when the variable is referenced in host language statements.

Example:

```
exec sql  SELECT column
          INTO   :var
          FROM   table
          WHERE  condition;
IF var < limit THEN ...
```

3.1.3 Indicator variables

In embedded SQL, *indicator variables* associated with main variables are used to handle NULL values in database tables. Indicator variables should be an exact numeric data type with scale zero and are declared in the same way as main variables in the SQL DECLARE SECTION. See Appendix A, under “Host variables” - “Declaration”, for a description of how main and indicator variables should be declared in the specific host languages.

Indicator variables are used in SQL statements by either specifying the name of the indicator variable, preceded by a colon, after the main variable name or by using the keyword INDICATOR

```
:main_variable :indicator_variable
```

or

```
:main_variable INDICATOR :indicator_variable
```

Transfer from tables to host variables

When a NULL value is retrieved into a host variable by a FETCH, SELECT INTO, EXECUTE or CALL statement, the value of the main variable is undefined and the value of the indicator variable is set to -1. An error occurs if the main variable is not associated with an indicator variable in the SQL statement. It is therefore recommended as a precaution that indicator variables are used for all columns which are not defined as NOT NULL in the database.

An indicator variable should always be used when a host variable is used for a routine parameter with mode OUT or INOUT because a NULL value can always be returned via a routine parameter.

When a non-null value is assigned to a main variable associated with an indicator variable, the indicator variable is set to zero or a positive value. A positive value indicates that the value assigned to a main character variable was truncated, and gives the length of the original value before truncation.

Transfer from host variables to tables

When the host variable associated with an indicator variable is used to assign a value to a column, the value assigned is NULL if the value of the indicator variable is set to -1. In such a case the value of the main variable is irrelevant. If the indicator variable has a value of zero or a positive value, or if the main variable is not associated with an indicator variable, the value of the main variable itself is assigned to the column.

3.2 The SQLSTATE variable

The SQLSTATE variable provides the application, in a standardized way, with return code information about the most recently executed SQL statement. SQLSTATE must be declared between the BEGIN DECLARE SECTION and the END DECLARE SECTION (i.e. in the SQL declare section), as a 5 character long string (excluding any terminating null byte). The return codes provided by SQLSTATE can contain digits and capital letters.

SQLSTATE consists of two fields. The first two characters of SQLSTATE indicates a class, and the following three characters indicates a subclass. Class codes are unique, but subclass codes are not. The meaning of a subclass code depends on the associated class code. To determine the category of the result of an SQL statement, the application can test the class of SQLSTATE according to the following:

SQLSTATE class	Result category
'00'	Success
'01'	Success with warning
'02'	No data
Other	Error

For a list of SQLSTATE values see Appendix B.

3.3 The diagnostics area

The diagnostics area holds status information for the most recently executed SQL statement. There is always one diagnostics area for an application, no matter how many connections the application holds. Information from the diagnostics area is selected and retrieved by the GET DIAGNOSTICS statement. The syntax for GET DIAGNOSTICS (including a description of the diagnostics area) is in the *Mimer SQL Reference Manual*. The GET DIAGNOSTICS statement does not change the contents of the diagnostics area, although it does set SQLSTATE.

3.4 The SQL Descriptor Area

An SQL descriptor area is used to hold descriptive information required for execution of dynamic SQL statements. SQL descriptor areas are allocated and maintained by embedded SQL statements, described in the Mimer SQL Reference Manual.

The SQL descriptor area is discussed in detail in Section 7.3.

4 IDENTS AND DATABASE CONNECTIONS

An ident is an authorization-id used to identify users, program idents and group idents in a Mimer SQL database. User idents are authorized to connect to the database, program idents provide specific privileges required when executing certain operations and a group ident is a collective identity which provides common privileges to its members.

Idents connect to a database through the CONNECT statement and the ENTER statement is used to take up the privileges provided by a program ident (see below).

A *database* in Mimer SQL refers to the complete collection of databanks which may be accessed from one Mimer SQL system. Mimer SQL supports the ability to change between different connections (i.e. access different databases) from within the same application program. An application program may have several database connections open simultaneously, although only one is active at any one time.

4.1 Mimer SQL idents

There are four kinds of ident in Mimer SQL:

USER — idents are authorized to connect to a Mimer SQL database, by using the CONNECT statement in an application program or by entering the correct ident name and password in an interactive environment. Any privileges a user ident holds may be exercised once the ident has logged on. User idents are generally associated with specific physical individuals authorized to connect to the database.

OS_USER — is an ident type which allows the user currently logged in to the operating system to connect to a Mimer SQL database without providing a username or password. If an OS_USER ident is defined with a password in Mimer SQL, the ident may connect to Mimer SQL in the same way as any other user ident (i.e. by providing a username and password). An OS_USER ident is subject to the same access restrictions as any other user ident.

PROGRAM idents may not initiate a connection to a Mimer SQL database, but may be entered from within an application program or interactive environment by using the ENTER statement. A connection should have been established before the ENTER statement is used (see Section 4.3.1). Entering a program ident is analogous to logging on as a user ident, in that the program ident gains access to the system and any privileges the ident holds become applicable. Program idents are generally associated with specific functions within the system, not with physical individuals.

GROUP idents are collective identities for groups of user or program idents. Any privileges granted to or revoked from a group ident automatically apply to all members of the group. Any ident can be a member of as many groups as required, and one group can include any number of members. Group idents provide a facility for organizing the privilege structure in the database system.

USER, OS_USER and PROGRAM idents are authorized users of the system. Every USER and PROGRAM ident has a unique ident name and a private password which must be correctly supplied to the CONNECT or ENTER statement in application programs. An OS_USER may access the database without explicitly providing a username or password on condition that the username for the user currently logged in to the operating system correspond to the definition of an OS_USER in the Mimer SQL database.

4.2 Access to the database

The access of each ident to the database is defined by privileges granted within the system. The privileges are grouped as follows:

System privileges

BACKUP gives the right to perform databank backup and restore operations

DATABANK gives the right to create databanks

IDENT gives the right to create idents

SCHEMA gives the right to create schemas

SHADOW gives the right to create and manage databank shadows

STATISTICS gives the right to execute the UPDATE STATISTICS statement

Object privileges

TABLE gives the right to create tables in a specified databank

EXECUTE gives the right to call a routine or to enter (connect to) a specified program ident

MEMBER grants membership in a specified group ident

USAGE gives the right to specify the named domain where a data type would normally be specified (in contexts where use of domains is allowed) or the right to use a specified sequence

Access privileges

SELECT	gives the right to read the table contents
INSERT	gives the right to add new rows to the table
DELETE	gives the right to remove rows from the table
UPDATE	gives the right to update the contents of the table
REFERENCES	gives the right to use the primary key or unique keys of the table as a foreign key from another table

System privileges are automatically granted to the system administrator at installation, and may be passed on to other idents. Object and access privileges are initially granted only to the creator of an object. The creator may however grant the privileges on to other idents.

All privileges may be granted with or without GRANT OPTION, which controls the right of the receiving ident to grant the privilege on to another ident.

Certain operations are not controlled by explicit privileges, but may only be performed by the creator of the object involved. These operations include ALTER (with the exception of ALTER IDENT, which may be performed by either the ident himself or by the creator of the ident), DROP, and COMMENT. Similarly, privileges may only be revoked by their grantor.

4.3 Connecting to a database

4.3.1 Connecting

Only idents of type USER or OS_USER are allowed to log on to Mimer SQL. Logging on is requested from an application program with the CONNECT statement (see the *Mimer SQL Reference Manual* for the syntax description).

The CONNECT statement establishes a connection between a user and a database. The user may be a USER ident (in which case the password must be provided) or an OS_USER (if such an ident has been created in the Mimer SQL database). To connect as the OS_USER ident with the same name as the current operating system user, provide an empty ident name string.

A connection may be established to any local or remote database, which has been made accessible from the current machine (see the Mimer SQL System Management Handbook for details), by specifying the database by name or by using the keyword DEFAULT.

If the keyword DEFAULT is used, the OS_USER ident is used for the connection attempt. If the database name is given as an empty string, the DEFAULT database is used.

The database may be given an explicit connection name for use in DISCONNECT and SET CONNECTION statements. If no explicit name is given, the database name is used as the connection name.

Normally, CONNECT should be the first executable SQL statement in an application program using embedded SQL. However, if an executable statement is issued before any connection has been established in the current application, an **implicit** connection will be attempted. An implicit connection is made to the DEFAULT database using the current operating system user. In order for the connect attempt to be successful, the current operating system user must be defined as an OS_USER ident in Mimer SQL and the DEFAULT database must be defined as a local database on the machine on which the current operating system user is defined. If an implicit connection has previously been established in the application and there is no current connection, issuing an executable statement will result in a new attempt to make the same implicit connection. However, if an explicit connection has previously been established in the application and there is no current connection, issuing an executable statement will cause an error.

4.3.2 Changing connection

A connection established by a successful CONNECT statement is automatically active. An application program may make multiple connections to the same or different databases using the same or different idents, provided that each connection is identified by a unique connection name. Only the most recent connection is active. Other connections are dormant, and may be made active by the SET CONNECTION statement. Resources such as cursors used by a connection are saved when the connection becomes dormant, and are restored by the appropriate SET CONNECTION statement.

The statement sequence below connects to a user-specific database as a specified ident name and to the DEFAULT database as OS_USER. The user-specific connection is initially active. Then the DEFAULT connection is activated. Finally the user-specific connection is activated again using SET CONNECTION.

```
exec sql CONNECT TO 'db' AS 'common' USER 'ident' USING 'pswd';
...
exec sql CONNECT TO DEFAULT;
...
-- Set activate connection to COMMON
exec sql SET CONNECTION 'common';
```

Note: If different connections are made with different idents, the apparent access rights of the application program may change when the current connection is changed.

4.3.3 Disconnecting

The DISCONNECT statement breaks the connection between a user and a database and frees all resources allocated to that user for the specified connection (all cursors are closed and all compiled statements are dropped). The connection to be broken is specified as the connection name or as one of the keywords ALL, CURRENT or DEFAULT. (If a transaction is active when the DISCONNECT is executed, a ROLLBACK is performed).

A connection does not have to be active in order to be disconnected. If an inactive connection is broken, the application still has uninterrupted access to the database through the current (active) connection, but the broken connection is no longer available for activation with SET CONNECTION.

If the active connection is broken, the application program cannot access the database until a new CONNECT or SET CONNECTION statement is issued.

Note: The distinction between breaking a connection with DISCONNECT and making a connection inactive by issuing a CONNECT or SET CONNECTION for a different connection is, a broken connection has no saved resources and cannot be reactivated by SET CONNECTION.

The table below summarizes the effect on the connection "con1" of CONNECT, DISCONNECT and SET CONNECTION statements depending on the state of the connection.

Statement	**con1 non-existent**	**con1 current**	**con1 inactive**
CONNECT TO db1 AS con1	con1 current	error - connection already exists	error - connection already exists
DISCONNECT con1	error - connection does not exist	con1 disconnected	con1 disconnected
SET CONNECTION con1	error - connection does not exist	ignored	con1 made current
CONNECT TO db2 AS con2	-	con1 made inactive	con1 unaffected
Statement	**con1 non-existent**	**con1 current**	**con1 inactive**
DISCONNECT con2	-	con1 unaffected	con1 unaffected
SET CONNECTION con2	-	con1 made inactive	con1 unaffected

4.3.4 Program idents - ENTER and LEAVE

Program idents may be entered from within an application program by using the ENTER statement (see the *Mimer SQL Reference Manual* for the syntax description). This statement must be issued in a context where a user is already connected as program idents cannot connect directly to the system.

When a program ident is entered, any privileges granted to that ident become current and privileges belonging to the previous ident (i.e. the ident issuing the ENTER statement) are suspended. However, any cursors opened by the previous ident remain open.

Program idents are disconnected with the LEAVE statement. If LEAVE is requested with the optional keyword RETAIN, the full environment of the program ident being left is kept. Cursors left open by the program ident are deactivated but not closed, and retain their positions in the respective result tables. The environment is restored if the program ident is re-entered.

If LEAVE is requested without RETAIN, the environment of the program ident being left is dropped. This means that all cursors and compiled statements are destroyed.

Note: The distinction between leaving a program ident with the option RETAIN and entering a new program ident is, while both operations save the environment of the program ident, cursors left open at ENTER may still be used but those left open at LEAVE RETAIN are inaccessible until the program ident is re-entered.

The statements ENTER and LEAVE may not be issued within transactions (see Chapter 6).

5 ACCESSING DATABASE OBJECTS

5.1 Retrieving data using cursors

Data is retrieved from database tables with the FETCH statement, which fetches the values from an individual row in a result set into host variables. The result set is defined by a SELECT construction or a result set procedure CALL (see Section 8.5) used in a cursor declaration. The cursor may be thought of as a pointer which moves through the rows of the result set as successive FETCH statements are issued. An exception is raised to indicate when the FETCH has reached the end of the result set. Data retrieval involves several steps in the application program code, which are as follows:

- declaration of host variables to hold data
- declaration of a cursor with the appropriate SELECT conditions or result set procedure CALL
- opening the cursor
- performing the FETCH
- closing the cursor

These steps are built into the application program as shown in the general frameworks below (only SQL statements are shown in the frameworks).

For a SELECT:

```
exec sql BEGIN DECLARE SECTION;
... var1, var2, ... varn ...
exec sql END DECLARE SECTION;

exec sql DECLARE cursor-name CURSOR FOR select-statement;

exec sql OPEN cursor-name;

loop as required
   exec sql FETCH cursor-name INTO :var1,:var2,...,:varn;
end loop

exec sql CLOSE cursor-name;
```

For a result set procedure CALL:

```
exec sql BEGIN DECLARE SECTION;
... var1, var2, ... varn ...
exec sql END DECLARE SECTION;

exec sql DECLARE cursor-name CURSOR FOR CALL routine-invocation;

exec sql OPEN cursor-name;

loop as required
   exec sql FETCH cursor-name INTO :var1,:var2,...,:varn;
end loop

exec sql CLOSE cursor-name;
```

5.1.1 Declaring host variables

All host variables used to hold data fetched from the database and used in selection conditions or as result set procedure parameters must be declared within an SQL DECLARE SECTION (see Chapter 3). Indicator variables for columns that may contain NULL values must also be declared. The same indicator variable may be associated with different main variables at different times, but declaration of a dedicated indicator variable for each main variable is recommended for clarity.

5.1.2 Declaring the cursor

A cursor operates as a row pointer associated with a result set. A cursor is defined by the DECLARE CURSOR statement and the set of rows addressed by the cursor is defined by the SELECT statement in the cursor declaration. Cursors are local to the program in which they are declared. A cursor is given an identifying name when it is declared.

DECLARE CURSOR is a declarative statement which does not result in any implicit connection to a database (see Chapter 4 for details on connecting to a database). Preprocessing the statement generates a series of parameters used by the SQL compiler but does not generate any executable code; the query-expression or result set procedure call in the cursor declaration is not executed until the cursor is opened.

If the cursor declaration contains a CALL to a result set procedure, it is FETCH that actually executes the procedure. The RETURN statement is used from within the result set procedure to return a row of the result set. Each FETCH causes statements in the result set procedure to execute until a RETURN statement is executed, which will return the row data defined by it. Execution of the procedure is suspended at that point until the next FETCH. If, during execution, the end of the procedure is encountered instead of a RETURN statement, the FETCH result is end-of-set. See Section 8.6 for a detailed description of result set procedures.

It is advisable always to use an explicit list of items in the SELECT statement of the cursor declaration. The shorthand notations “SELECT *” and “SELECT table.*” are useful in interactive SQL, but can cause conflicts in the variable lists of FETCH statements if the table definition is changed.

The cursor declaration can use host variables in the WHERE or HAVING clause of the SELECT statement. The result set addressed by the cursor is then determined by the values of these host variables at the time when the cursor is opened.

The same cursor declaration can thus address different result sets depending on when the cursor is opened, for example:

```
exec sql DECLARE C1 CURSOR..;        -- cursor with host variables
set variables
exec sql OPEN C1;                    -- open one result set
...
exec sql CLOSE C1;
change variables
exec sql OPEN C1;                    -- open different result set
```

Scrollable cursors can be declared using the SCROLL keyword. When a cursor is declared as scrollable, records can be fetched using an orientation specification. This makes it possible to scroll through the result set with the cursor.

Cursors which are to be used only for retrieving data may be declared with a FOR READ ONLY clause in the SELECT statement. This can improve performance slightly in comparison with cursors which permit update and delete operations.

5.1.3 Opening the cursor

A declared cursor must be opened with the OPEN statement before data can be retrieved from the database. The OPEN statement evaluates the cursor declaration in terms of

- the privileges the current user holds on any tables and views accessed by the cursor
- the values of any host variables used in the SELECT clause
- for a cursor calling a result set procedure, whether the current user has the required EXECUTE privilege on the procedure and also the values of any IN parameters

When the OPEN statement has been executed, the cursor is positioned **before** the first row in the result set.

5.1.4 Retrieving data

Once a cursor has been opened, data may be retrieved from the result set with FETCH statements (see the *Mimer SQL Reference Manual* for the syntax description).

Host variables in the variable list correspond in order to the column names specified in the SELECT clause of the cursor declaration. The number of variables in the FETCH statement may not be more than the number of columns selected. The number of variables may be less than the number of columns selected, but a “success with warning”-code is then returned in SQLSTATE.

A suitably declared record structure may be used in place of a variable list in host languages where this is supported (see Appendix A).

Each FETCH statement moves the cursor to the specified row in the result set before retrieving data. In strict relational algebra, the ordering of tuples in a relation (the formal equivalent of rows in a table) is undefined. The SELECT statement in the cursor declaration may include an ORDER BY clause if the ordering of rows in the result set is important to the application.

Note: A cursor declared with an ORDER BY clause cannot be used for updating table contents.

If no ORDER BY clause is specified, the ordering of rows in the result set is unpredictable.

Note: The variables into which data is fetched are specified in the FETCH statement, not in the cursor declaration. In other words, data from different rows in the result set may be fetched into different variables.

When there are no more rows to fetch, the exception condition NOT FOUND will be raised. The following construction thus fetches rows successively until the result set is exhausted:

```
exec sql DECLARE C1 CURSOR FOR select-statement;
exec sql OPEN C1;

exec sql WHENEVER NOT FOUND GOTO done;
loop
   exec sql FETCH C1 INTO :var1,:var2,...,:varn;
end loop

done:
exec sql CLOSE C1;
```

The access rights for a user are checked when the cursor is opened and they remain unchanged for that cursor until the cursor is closed. For example, if an application program declares and opens a cursor, then SELECT access on the table is revoked from the user running the program, data can still be fetched from the result set as long as the cursor remains open. Any subsequent attempt to open the same cursor will, however, fail.

5.1.5 Closing the cursor

An opened cursor remains open until it is closed with the CLOSE statement, a COMMIT or ROLLBACK is performed or until the current connection is disconnected. Once a cursor is closed, the result set is no longer accessible. The cursor declaration remains valid, however, and a new cursor may be opened with the same declaration.

Note: The result set addressed by the new cursor may not be the same if the contents of the database or the values of variables used in the declaration have changed.

Normally, resources used by the cursor remain allocated when the cursor is closed and will be used again if the cursor is re-opened. The optional form "CLOSE *cursor-name* RELEASE" deallocates cursor resources. Use of CLOSE with the RELEASE option is recommended in application programs which open a large number of cursors, particularly where system resources are limited.

Note: The use of CLOSE with the RELEASE option may slow down performance if there is a following OPEN, since it requires that new resources are allocated at the next OPEN for that cursor. For this reason it should only be used when necessary.

Cursors are local to a connection and remain open but dormant when the connection is made dormant. The state of dormant cursors is fully restored (including result set addressed and position in the result set) when the connection is reactivated. Cursors are, however, closed and cursor resources are deallocated, when a connection is disconnected.

Note: Cursors opened in a program ident context are closed and resources deallocated when LEAVE is executed within the same connection, unless LEAVE RETAIN is specified.

5.2 Retrieving single rows

If the result of a SELECT statement is known to be a single row, the SELECT INTO statement may be used as an alternative to fetching data through a cursor. This is a much simpler programming construction, since cursors are not used and the only requirement is that host variables used in the SELECT INTO statement are declared in the DECLARE SECTION. However, there are two disadvantages associated with SELECT INTO:

- An error occurs if the result set addressed by the search condition contains more than one row. In other words, SELECT INTO can only be reliably used when there is **no possibility** of a multi-row result set (essentially when the search condition includes a UNIQUE or PRIMARY KEY column or returns just the result of a set function, e.g. COUNT(*)).
- Execution of the SELECT INTO statement involves a check that the result set contains a single row, which may incur unnecessary overhead. Even if it is known that the result row is unique, a single FETCH operation through a cursor may be a more efficient implementation.

Use of a SELECT INTO statement is justified when the result set may contain several rows, but it is a condition for continued execution of the application program that the result row is unique. With a cursor, this would require a construction which checked that one and only one FETCH operation could be performed (alternatively, use a separate SELECT COUNT with the same search condition as the cursor). In such a case, a SELECT INTO statement with a check on the return code (see Chapter 10) is probably the preferred solution.

A CALL statement can be used to return information to the one or more host variables associated with the output parameter(s) of the procedure.

A SET statement can be used with a function invocation to return information to one host variable.

5.3 Retrieving data from multiple tables

Data can be retrieved from multiple tables in embedded SQL by addressing several tables in the SELECT statement of the cursor declaration, in the same way as in interactive SQL. The preprocessor generates a SELECT statement addressing multiple tables, which is optimized by the SQL compiler when the cursor is opened. For example:

```
exec sql DECLARE C1 CURSOR FOR SELECT ... FROM A, B
                                    WHERE A.X=B.Y;
exec sql OPEN C1;
...
```

An alternative way to link information between tables could be to define the search condition for one cursor in terms of a variable fetched through another cursor:

```
exec sql DECLARE C1 CURSOR FOR SELECT X FROM A;
exec sql DECLARE C2 CURSOR FOR SELECT ... FROM B WHERE Y=:hostx;

exec sql OPEN C1;
exec sql FETCH C1 INTO :hostx;
exec sql CLOSE C1;

exec sql OPEN C2;
exec sql FETCH C2;
...
```

When considering the two alternatives, the first one is preferred. The reason for this is:

- The SQL optimizer gets the full information about the query that it is supposed to return a result set for. In this way the optimizer can make more use of statistical information and it can thereby optimize the query to execute in a more efficient way.
- The application will require less resources in form of open cursors.
- If the application is run in a client/server environment, the second alternative will cause more communication over the network, since it will send data over the net which is only used to determine which data from the second cursor that will be selected and is of no real interest to the application.
- The application will be more compact as well as easier to understand and maintain.

5.3.1 The 'Parts explosion' problem

A special case of data retrieval from multiple tables is the use of stacked cursors to fetch data from logical copies of the same table, in a manner that provides a solution to the so called “Parts explosion” problem. A cursor can be defined as REOPENABLE and the same cursor may be opened several times in succession in the same application program, each previous instance of the cursor being saved on a stack and restored when the following instance is closed. A FETCH statement refers to the most recently opened instance of a cursor. Each instance of the cursor addresses an independent result set and the position of each cursor in its own result set is saved on the stack.

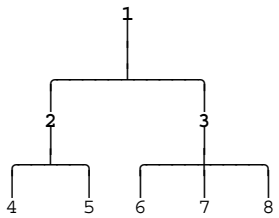
Note: Result sets addressed by different instances of a cursor may differ according to the conditions prevailing when the cursor instance was opened.

The state of the cursor stack needs to be controlled by the application. A counter can be used to indicate if there are more instances of the cursor remaining on the stack. See the example that follows.

Stacked cursors are typically used in application programs which traverse a tree structure stored in the database.

For example (this is a simplified variant of the “parts explosion” problem), traverse a tree structure and print out the leaf nodes:

Tree



Relational table

Parent	Child
1	2
1	3
2	4
2	5
3	6
3	7
3	8

```
procedure TRAVERSE;

     integer CSTACK,LASTC;
exec sql BEGIN DECLARE SECTION;
     integer PARENT,CHILD;
     string  SQLSTATE(5);
exec sql END DECLARE SECTION;
begin
    exec sql DECLARE CTREE REOPENABLE CURSOR FOR
             SELECT  PARENT,CHILD
             FROM    TREE
             WHERE   PARENT = :PARENT;
    CSTACK := 1;
    LASTC := 1;
    PARENT := 1;                      -- start at root node
    exec sql OPEN CTREE;
    loop
        exec sql FETCH CTREE INTO :PARENT,:CHILD;
        if SQLSTATE = "02000" then    -- no more children
            exec sql CLOSE CTREE;     -- pop the parent
            CSTACK := CSTACK-1;
            exit when CSTACK = 0;
            if CSTACK >= LASTC then
               print(PARENT);         -- write leaf node
            end if;
            LASTC := CSTACK;
        else                          -- step to next level
            PARENT := CHILD;
            exec sql OPEN CTREE;      -- stack the current parent
                                         and open new level
            CSTACK := CSTACK+1;
        end if;
    end loop;
end TRAVERSE;
```

The counters CSTACK and LASTC keep track of the number of stacked cursor levels and the latest level in the tree hierarchy respectively.

5.4 Entering data into tables

5.4.1 Cursor-independent operations

The SQL statements CALL, INSERT, DELETE and UPDATE, as well user-defined function invocations, embedded in application programs operate on a set of rows in a table or view in exactly the same way as in interactive SQL. Host variables may be used in the statements to supply values or set search conditions. Host variables may be used as routine parameters.

Examples:

```
exec sql UPDATE BOOKADM.FREEROOMS
         SET    FREECOUNT = FREECOUNT-1
         WHERE  HOTELCODE =  :HOTCOD
         AND    ROOMTYPE  =  :BOOK_ROOMT
         AND    ON_DATE   >= :BOOK_ARRIVE
         AND    ON_DATE   <  :BOOK_DEPART;
```

```
exec sql CALL ASK_FREEROOMS(:HOTCOD,
                            :BOOK_ROOMT,
                             CURRENT_DATE,
                             CURRENT_DATE + INTERVAL '7' DAY,
                            :FREEROOM_COUNT);
```

From the standpoint of the application program, each statement is a single indivisible operation, regardless of how many columns and rows are affected.

5.4.2 Updating and deleting through cursors

The UPDATE CURRENT and DELETE CURRENT statements (see the *Mimer SQL Reference Manual* for the syntax description) allow update and delete operations respectively, to be controlled on a row-by-row basis from an application program. These statements operate through cursors, which are declared and opened as described above for FETCH.

These statements operate on the current row of the cursor referenced in the statement. If there is no current row (e.g. the cursor has been opened but not yet positioned with a FETCH statement), an error is raised.

UPDATE CURRENT changes the content of the current row according to the SET clause in the statement, but does not change the position of the cursor. Two consecutive UPDATE CURRENT statements will therefore update the same row twice.

DELETE CURRENT deletes the current row and does not move the cursor; after a DELETE CURRENT statement, the cursor is positioned “between rows” and there is no current row. The cursor must be moved to the next row with a FETCH statement before any other operation can be performed through the cursor.

For both UPDATE CURRENT and DELETE CURRENT statements, the table name as used in the statement must be exactly the same as the table name addressed in the cursor declaration. The cursor must also address an updatable result set.

If a FOR UPDATE OF clause is used to specify which fetched columns may be updated, only the columns specified may appear in the corresponding UPDATE statement.

Cursors are ***not*** updatable if the data retrieval statement in the cursor declaration contains any of the following features at the top level (i.e. not in a subselect) of the statement:

- reference to more than one table in the FROM clause
- reference to a read-only view in the FROM clause
- the keyword DISTINCT
- set-functions in the SELECT list (AVG, COUNT, MAX, MIN, SUM)
- arithmetic or string concatenation expressions in the SELECT list
- a GROUP BY clause
- an ORDER BY clause
- the UNION keyword

- the result set of an explicit inner or outer JOIN
- a CALL to a result set procedure

When to use UPDATE CURRENT, DELETE CURRENT

UPDATE CURRENT and DELETE CURRENT statements are useful for manipulating single rows in interactive applications where rows are displayed, and the user decides which rows to delete or update. The example below illustrates the program framework for such an operation (the construction is similar for a DELETE CURRENT operation):

```
exec sql BEGIN DECLARE SECTION;
...
exec sql END DECLARE SECTION;
...
exec sql DECLARE C1 CURSOR FOR ... ;
...
exec sql OPEN C1;
exec sql WHENEVER NOT FOUND GOTO done;
loop
    exec sql FETCH C1 INTO :var1,:var2,...,:varn;
    display var1,var2,...,varn;
    prompt "Update this row?";
    if answer = "yes" then
       prompt "Give new values";
       exec sql UPDATE tab SET col1 = :newval1, col2 = :newval2, ...
                WHERE CURRENT OF C1;
       display "row updated";
    end if;
    prompt "Display next row?";
    if answer = "no" then
       exit;
    end if;
end loop;

done:
exec sql CLOSE C1;
```

In situations where there is no requirement to interactively choose rows and where all the rows to be updated or deleted can be specified completely in terms of a WHERE clause, it is more efficient to do so rather than use a cursor. An operation completely specified as a WHERE clause is executed as a single statement, rather than a series of statements (i.e. one for each FETCH etc.).

6 TRANSACTION HANDLING AND DATABASE SECURITY

6.1 Transaction principles

A transaction is an "atomic" operation which may not be divided into smaller operations. Three transaction phases exist: *build-up*, during which the database operations are requested; *prepare,* during which the transaction is validated; *commitment*, during which the operations performed in the transaction are written to disk.

Read-only transaction have only two phases: *build-up* and *prepare*.

Transaction *build-up* may be started explicitly or implicitly (see Section 6.2.1); *prepare* and *commitment* are both initiated explicitly through a request to commit the transaction (using COMMIT). In interactive application programs, build-up takes place typically over a time period determined by the user, while *prepare* and *commitment* are part of the internal process of committing a transaction, which occurs on a time-scale determined by machine operations.

6.1.1 Concurrency control

Since Mimer SQL uses optimistic concurrency control, deadlocks never occur (see Section 6.1.2 for a further discussion of deadlocks). How optimistic concurrency control works in Mimer SQL is described below.

The transaction begins by taking a snapshot of the database in a consistent state. During build-up, changes requested to the contents of the database are kept in a *write-set* and are not visible to other users of the system. This allows the database to remain fully accessible to all users. The application program in which build-up occurs will see the database as though the changes had already been applied. Changes requested during transaction build-up become visible to other users when the transaction is successfully committed.

During build-up, a *read-set* records the state of the database as seen at the time of each operation (including intended changes). If the state of the database at commitment is inconsistent with the read-set, a conflict is reported and the transaction is rolled back (i.e. the write-set is erased and no changes are made to the database). This can happen if, for instance, a transaction updates a row which gets deleted by another user after build-up has started but before the transaction is committed. The application program is responsible for taking appropriate action if a transaction conflict occurs.

Concurrency control guidelines

Because of the nature of this concurrency control protocol, it is important that some of the implications are understood.

A transaction that exists for a long elapsed time has a greater chance of conflicting with changes made by other users than a transaction with a short elapsed time.

At the other extreme, an application that immediately commits every executed SQL statement will seldom meet any conflicts, but will incur unnecessary overhead.

In general:

- keep transactions as short as is reasonably possible
- keep interactive user dialogs outside of transactions

A common situation that can generate unnecessarily large read-sets is the following: An application program reads through the rows in a table in a loop construct, with a conditional exit to update a row on user intervention. It is tempting to simply place a COMMIT after the update statement:

```
loop
   exec sql FETCH C1 INTO :var1,:var2,...,:varn;
   display var1,var2,...,varn;
   prompt "Update row?";
   exit when answer = "yes";
end loop

exec sql UPDATE table SET ... WHERE CURRENT OF C1;
exec sql COMMIT;
```

However, the FETCH loop can create a large read-set while waiting for the user update request, risking transaction conflict at the UPDATE. A tempting solution for this problem might be:

```
loop
   exec sql FETCH C1 INTO :var1,:var2,...,:varn;
   display var1,var2,...,varn;
   prompt "Update row?";
   exit when answer = "yes";
   exec sql ROLLBACK;
end loop;

exec sql UPDATE table SET ... WHERE CURRENT OF C1;
exec sql COMMIT;
```

But since ROLLBACK closes all cursors, this will not work. Instead, something like the following is a better approach:

```
exec sql SET TRANSACTION START EXPLICIT

loop
   exec sql FETCH C1 INTO :var1,:var2,...,:varn;
   display var1,var2,...,varn;
   prompt "Update row?";
   exit when answer = "yes";
end loop

exec sql START;
exec sql UPDATE table SET ...
         WHERE col1 = :var1,
               col2 = :var2, ...
exec sql COMMIT;
```

By using this mechanism, only the update statement is included in the transaction, thus minimizing the risk of a transaction conflict.

6.1.2 Locking

Deadlock situations, which can be relatively common in some database management systems where records are locked during transaction build-up, can not occur in Mimer SQL. In Mimer SQL it is impossible for two processes to be waiting for a record locked by the other process. In some other database management systems this situation may require operator intervention to resolve the problem.

In any database system, at some stage in a transaction, the data records must be locked to prevent access by other processes and to ensure that the transaction is not interrupted. In the Mimer SQL system, no change is made to the database contents during the transaction build-up and no records are locked. This means that the database can be freely accessed (and updated) by any other process; the data accessed by the transaction is only locked during the commit phase. In this way, locks are held only for a very short period of time. The problems associated with locking are further reduced since only those records that are actually to be updated are locked. Other data in the same table continues to be accessible to other transactions.

6.2 Transaction control statements

6.2.1 Starting transactions

Transaction start may be set to EXPLICIT or IMPLICIT.

The default transaction start setting is IMPLICIT, which means a transaction will be started automatically whenever one is needed.

To set the transaction start mode, use the statements:

```
SET TRANSACTION START EXPLICIT;
SET TRANSACTION START IMPLICIT;
```

Different database connections can use different transaction start options.

The START statement can always be used to explicitly start a transaction. This is useful if a number of related updates are to be performed and it is desirable that all the updates succeed or fail together to maintain consistency.

You cannot start a transaction while a transaction is already active.

Explicit transaction start

With this setting, transactions are never automatically started. All transactions **must** be explicitly started by executing the START statement.

Any update operation (insert, update, delete) involving a table in a databank with the TRANS or LOG option **must** occur within a transaction. An error will be raised if such an update is attempted without first starting a transaction.

All the statements issued after the START statement and before the transaction is concluded are grouped together within that single transaction.

A transaction is concluded by executing a COMMIT or ROLLBACK statement.

Implicit transaction start

With this setting, a transaction is started **automatically** (if one is not already active) by an update involving a reference to an object stored in a databank with the TRANS or LOG option (i.e. if none of the objects referenced in the update are stored in a databank with the TRANS or LOG option, no transaction is required and therefore one is not started).

The START statement may be used to explicitly start a transaction if required, typically to allow several updates to be grouped together within a single transaction for consistency, as already described.

An automatically started transaction is concluded by executing a COMMIT or ROLLBACK statement.

All the statements issued after the initiating update and before the concluding COMMIT or ROLLBACK statement are grouped together within that single transaction.

6.2.2 Ending transactions

Transactions must be ended with the COMMIT or ROLLBACK statement.

COMMIT — This statement requests that the operations in the write-set are executed on the database, making the changes permanent and visible to other users. The SQLSTATE value returned when a COMMIT statement is executed indicates either that the transaction commitment was successful (SQLSTATE = '00000') or that a transaction conflict occurred (SQLSTATE <> '00000').

ROLLBACK — This statement abandons the transaction. The read-set and write-set are dropped and no changes are made to the database. ROLLBACK is always successful.

Note: The keyword ROLLBACK is used in Mimer SQL for compatibility with SQL standards. A transaction in Mimer SQL is never physically “rolled back” in the sense of undoing changes made to the database, since changes are not actually effected until a successful COMMIT is performed.

Transactions which are not successfully committed due to a transaction conflict do not have to be explicitly rolled back. The ROLLBACK statement is most commonly used in exception routines for handling error situations which are detected by the application during transaction build-up.

If a connection or program is terminated without requesting a COMMIT or ROLLBACK for the current transaction, the system will abort the transaction. None of the changes requested during the transaction build-up will be made to the database.

Transaction handling in BSQL differs slightly from that described here - see Section 6.3.1 of the Mimer SQL User's Manual for details.

6.2.3 Optimizing transactions

The following SET TRANSACTION options are used to optimize transaction performance:

READ ONLY — this setting should always be used for transactions that do not require update access to the database. Significant performance gains can be achieved, especially for queries retrieving large numbers of rows, when this setting is used in queries when there is no need for update access to the database.

READ WRITE — this setting should only be used for transactions that require update access to the database. This is the default setting for a transaction.

The default option is READ WRITE, or the option defined to be the default for the current session by using the SET SESSION statement (see Section 6.2.6).

The SET TRANSACTION READ command only affects the **single next** transaction started after it is used.

6.2.4 Consistency within transactions

The SET TRANSACTION ISOLATION LEVEL options can be used to control the degree to which the changes occurring within one transaction are affected by the changes occurring within other concurrently executing transactions.

The default option is REPEATABLE READ, or the option defined to be the default for the current session by using the SET SESSION statement (see Section 6.2.6).

The SET TRANSACTION ISOLATION LEVEL command only affects the **single next** transaction started after it is used.

The following options are available:

SERIALIZABLE — this setting guarantees that the end result of the operations performed by two or more concurrent transactions will be the same **as if** the transactions had been executed in a serial fashion, where one executes to completion before the other starts.

REPEATABLE READ — this setting offers the same consistency guarantee as serializable, except that the concurrency effect known as "phantoms" may be encountered (see below for a reference to the definition of this concurrency effect).

READ COMMITTED — this setting offers the same consistency guarantee as repeatable read, except that the concurrency effect known as "non-repeatable read" may also be encountered (see below for a reference to the definition of this concurrency effect).

READ UNCOMMITTED — this setting offers the same consistency guarantee as read committed, except that the concurrency effect known as "dirty read" may also be encountered (see below for a reference to the definition of this concurrency effect).

For a definition of the concurrency effects mentioned above ("phantoms", "non-repeatable read" and "dirty read") refer to the description of SET TRANSACTION in Chapter 6 of the Mimer SQL Reference Manual.

All of the isolation level settings guarantee that each transaction will be executed completely or not at all and that no updates will be lost.

6.2.5 Exception diagnostics within transactions

The SET TRANSACTION DIAGNOSTICS SIZE option allows the size of the diagnostics area to be defined. An unsigned integer value specifies how many exceptions can be stacked in the diagnostics area, and examined by GET DIAGNOSTICS (see the *Mimer SQL Reference Manual*), in situations where repeated RESIGNAL operations have effectively been performed.

The SET TRANSACTION DIAGNOSTICS SIZE setting only affects the **single next** transaction to be started.

The default SET TRANSACTION DIAGNOSTICS SIZE setting (5 or whatever has been defined to be the default by using SET SESSION) applies unless an alternative is explicitly set before **each** transaction.

6.2.6 Setting default transaction options

The SET SESSION statement can be used to define the default settings for the transaction options set by SET TRANSACTION READ, SET TRANSACTION ISOLATION LEVEL and SET TRANSACTION DIAGNOSTICS SIZE.

As these SET TRANSACTION commands only affect the **single next** transaction started after they are used, it is often convenient to define the desired default options for each of them.

A detailed description of the SET SESSION statement can be found in Chapter 6 of the Mimer SQL Reference Manual.

6.2.7 Statements in transactions

The table below summarizes whether statements may or may not be used inside transactions.

Statement	Allowed in transactions	Comments
Access control statements		
GRANT, REVOKE	Yes	
Connection statements		
CONNECT, SET CONNECTION	Yes	
DISCONNECT	Yes	A ROLLBACK is performed on any active transaction.
ENTER, LEAVE (program ident)	No	
Data definition statements		
ALTER, COMMENT	Yes	
CREATE, DROP	Yes	

Statement	Allowed in transactions	Comments
Data manipulation statements		
SELECT EXPRESSION, SELECT INTO, FETCH, INSERT, DELETE, DELETE CURRENT, UPDATE, UPDATE CURRENT	Yes	
OPEN, CLOSE	Yes	COMMIT and ROLLBACK close any open cursors
Declarative statements		
DECLARE CONDITION, DECLARE CURSOR, DECLARE HANDLER, DECLARE VARIABLE	Not applicable	Declarative statement
Diagnostic statements		
GET DIAGNOSTICS, RESIGNAL, SIGNAL	Yes	
Dynamic SQL statements		

PREPARE, DESCRIBE, EXECUTE, EXECUTE IMMEDIATE, ALLOCATE CURSOR, ALLOCATE DESCRIPTOR, DEALLOCATE DESCRIPTOR, DEALLOCATE PREPARE, GET DESCRIPTOR, SET DESCRIPTOR	Yes	See Chapter 7.
Embedded SQL control statements		
DECLARE SECTION, WHENEVER	Not applicable	Declarative statement
Procedure control statements		
CALL, CASE, COMPOUND STATEMENT, IF, LEAVE, LOOP, REPEAT, RETURN, SET, WHILE	Yes	
System administration statements		
ALTER DATABANK RESTORE, CREATE BACKUP, CREATE INCREMENTAL BACKUP, SET DATABANK, SET DATABASE, SET SHADOW, UPDATE STATISTICS	No	These statements create internal transactions to ensure data dictionary consistency

Statement	**Allowed in transactions**	**Comments**
Transaction control statements		
SET SESSION, SET TRANSACTION, START	No	These statements control transaction behavior
COMMIT, ROLLBACK	Yes	

6.2.8 Cursors in transactions

The transaction terminating statements COMMIT and ROLLBACK automatically close any cursors opened by the current connection, regardless of whether they were opened before or after the transaction was started. If a cursor is stacked, all instances of the cursor are closed. Any cursors which may be retained in dormant connections are not affected. Cursors are also closed automatically by LEAVE and DISCONNECT.

In SET TRANSACTION START EXPLICIT mode, cursors may be opened and used outside transactions. Such cursors remain open when an ENTER or LEAVE statement is issued. This is illustrated in the following statement sequence:

```
...
exec sql SET TRANSACTION START EXPLICIT;
exec sql DECLARE C1 CURSOR FOR SELECT col1 FROM tab1;
exec sql DECLARE C2 CURSOR FOR SELECT col2 FROM tab2
                               WHERE checkcol = :var1;
exec sql OPEN C1;
loop
   exec sql FETCH C1 INTO :var1;      -- fetch value from tab1
   exec sql ENTER ... ;               -- change current ident
      exec sql OPEN C2;
      exec sql FETCH C2 INTO ...;     -- fetch row for C2
      exec sql CLOSE C2;
   exec sql LEAVE;
end loop;
...
```

In the above example, the value fetched for the cursor C1 is used to determine the set of rows addressed by cursor C2. Cursor C1 remains open and positioned during the ENTER...LEAVE sequence. Each time the loop is executed, a new value is fetched by C1 and a new set of is rows addressed by C2. The same behavior applies when LEAVE RETAIN is used to leave a program ident but keep the environment for the ident.

A cursor opened and used outside a transaction may however not be used within a transaction. If the same cursor is required outside and inside a transaction, separate instances must be opened. Remember that separate instances of a cursor address separate result sets:

```
...
exec sql SET TRANSACTION START EXPLICIT;
exec sql DECLARE C1 REOPENABLE CURSOR FOR SELECT col1 FROM tab1;
exec sql OPEN C1;
exec sql FETCH C1 INTO ...;      -- first row (outside transaction)
...
exec sql START;
exec sql OPEN C1;                -- new instance of cursor
exec sql FETCH C1 INTO ...;      -- first row again
...
```

6.2.9 Error handling in transactions

In general, errors and exception conditions are reported in SQLSTATE after each executable SQL statement. The value of SQLSTATE indicates the outcome of the preceding statement (see Appendix B for a list of SQLSTATE values). GET DIAGNOSTICS can be used to get detailed status information after an SQL statement.

The value of SQLSTATE after a COMMIT statement indicates the success or failure of the request to commit the transaction, **not** the outcome of any data manipulations performed within the transaction.

Use of the general error handling statement WHENEVER (see Chapter 6 of the Mimer SQL Reference Manual for a description of WHENEVER) in transactions requires some care:

- Program control can be transferred to an exception routine in the event of an error. Make sure that the exception routine is designed to take care of uncompleted transactions. Most commonly, the first SQL statement in the exception routine should be GET DIAGNOSTICS. The exception routine should normally also execute a ROLLBACK statement. Remember that if the exception routine is used from a statement outside a transaction, any open cursors belonging to the current ident will be closed by the ROLLBACK statement. GET DIAGNOSTICS can be used to determine whether or not a transaction is active.

- For transaction conflict, the SQLSTATE value returned from the COMMIT statement falls into the SQLERROR class. If the transaction is to be retried in the event of conflict, make sure that no "WHENEVER SQLERROR GOTO exception" statement is operative. If WHENEVER error handling is used in an application program, a suitable program structure for COMMIT statements is:

```
exec sql WHENEVER SQLERROR GOTO exception;
...
exec sql WHENEVER SQLERROR GOTO retry;
exec sql COMMIT;
exec sql WHENEVER SQLERROR GOTO exception;
...
```

6.3 Transactions and logging

Changes made to a database may be logged, to provide back-up protection in the event of hardware failure, provided that the changes occur within a transaction and that the databanks involved have the LOG option. Transaction handling is, therefore, important even in standalone environments where concurrency control issues do not arise.

Transaction control and logging is determined at the databank level by options set when the databank is defined. The options are:

LOG	All operations on the databank are performed under transaction control. All transactions are logged.
TRANS	All operations on the databank are performed under transaction control. No transactions are logged.
NULL	All operations on the databank are performed without transaction control (even if they are requested within a transaction), and are not logged. Sets of operations (DELETE, UPDATE or INSERT on several rows) which are interrupted will not be rolled back.

Note: All important databanks should be defined with LOG option, so that valuable data is not lost by any system failure.

6.4 Protection against data loss

6.4.1 System interruptions

If a transaction build-up is interrupted by a system failure or a program termination (deliberate or otherwise) the transaction is aborted and none of the requested changes are made to the database.

Transactions which are interrupted after the request to commit, but before all operations in the transaction have been executed on the database, are completed by the automatic recovery functionality when the databank involved is next accessed. There is no possibility of transaction conflict in such an automatic completion, since no other process can access the affected data as long as an incomplete transaction is pending.

In the event of a system failure which interrupts one or more application programs, it may be necessary to examine the database contents “by hand” to determine which transactions failed to commit before the interruption.

6.4.2 Hardware failure

A databank which is damaged by hardware failure (e.g. a disk crash) may be recovered using back-up copies and the transaction log (LOGDB), provided that all write operations on the databank have been logged (and that the back-up copies and the LOGDB databank are intact). The backup and restore facilities are described in the *Mimer SQL System Management Handbook*.

7 DYNAMIC SQL

7.1 Principles of dynamic SQL

Dynamic SQL allows you to execute SQL statements placed in a string variable instead of explicitly writing the statements inside a program. This allows SQL statements to be constructed within an application program. These facilities are typically used in interactive environments, where SQL statements are submitted to the application program from the terminal.

An example of when dynamic SQL is needed would be a program for interactive SQL, where any correct SQL statement may be entered at the terminal and processed by the application. Limited dynamic facilities may however be provided by relatively simple application programs.

The following classes of SQL statements may be submitted to programs using dynamic SQL. Statements excluded from dynamic applications are declarations, diagnostic statements and dynamic SQL statements themselves.

Access control statements:

ENTER
LEAVE

Data definition statements:

CREATE
ALTER
COMMENT
DROP

Security control statements:

GRANT
REVOKE

Transaction control statements:

SET SESSION
SET TRANSACTION
START
COMMIT
ROLLBACK

Data manipulation statements:

CALL
SELECT
SELECT INTO
INSERT
UPDATE
UPDATE CURRENT
DELETE
DELETE CURRENT

System administration statements:

CREATE BACKUP
CREATE INCREMENTAL BACKUP
ALTER DATABANK RESTORE
SET DATABASE
SET DATABANK
SET SHADOW
UPDATE STATISTICS

Statements may be submitted to dynamic SQL applications in two forms:

- Fully defined statements, written exactly as they would be submitted to interactive SQL. For example:

```
GRANT ALL ON HOTEL TO CHARLIE

UPDATE ROOM_STATUS SET STATUS = 'KEY OUT'
WHERE ROOMNO = 'SKY112'

SELECT * FROM HOTEL

SELECT RESERVATION,SUM(AMOUNT)
FROM   BILL
GROUP BY RESERVATION
```

- Statements with “parameter markers”, which identify positions where the value of a host variable will be inserted when the statement is executed or the cursor is opened. A parameter marker is represented by a question mark. For example:

```
UPDATE ROOM_STATUS SET STATUS = ? WHERE ROOMNO = ?

DELETE FROM BOOK_GUEST WHERE RESERVATION = ?

SELECT HOTELCODE,ON_DATE,FREECOUNT*?
FROM   FREEROOMS
WHERE  ROOMTYPE  = ?
  AND  HOTELCODE IN (SELECT HOTELCODE
                     FROM HOTEL
                     WHERE CITY = ?)
```

  Statements submitted with parameter markers are equivalent to normal embedded statements using host variables, except that the statements are defined at run-time.

7.2 General summary of dynamic SQL processing

The following statements are used when SQL statements are dynamically submitted:

Statement	Description
ALLOCATE CURSOR	Allocate extended cursor.
ALLOCATE DESCRIPTOR	Allocate SQL descriptor area.
CLOSE	Close an open cursor.
DEALLOCATE DESCRIPTOR	Deallocate SQL descriptor area.
DEALLOCATE PREPARE	Deallocate prepared SQL statement.
DECLARE CURSOR	Declare a cursor for a statement which will be dynamically submitted.
DESCRIBE	Examine the object form of the statement and assign values to the appropriate parameters in the SQL descriptor area.
EXECUTE	Execute a prepared statement (except result set generating statements).
EXECUTE IMMEDIATE	Shorthand form for PREPARE followed by EXECUTE. This form can only be used for fully-defined non-result set statements with no parameter markers.
FETCH	Fetch rows for a dynamic cursor.
GET DESCRIPTOR	Get values from the SQL descriptor area.
OPEN	Open a prepared cursor.
PREPARE	Compile an SQL source statement into an internal object form.
SET DESCRIPTOR	Set values in the SQL descriptor area.

All statements submitted to dynamic SQL programs must be prepared.

All prepared statements except SELECT statements and result set procedure calls are executed with the EXECUTE statement. SELECT statements and calls to result set procedures are “executed” by the use of OPEN and FETCH for a cursor declared with the prepared statement.

The declaration of a cursor for a statement (DECLARE CURSOR) must always precede the PREPARE operation for the same statement in an application using dynamic SQL.

7.3 SQL descriptor area

The SQL descriptor area is used for managing input and output data in dynamically submitted SQL statements where the number and/or data type of the host variables required is not known at the time the program is written. An SQL descriptor area is allocated with the embedded SQL statement ALLOCATE DESCRIPTOR and deallocated with DEALLOCATE DESCRIPTOR. See Chapter 6 of the Mimer SQL Reference Manual for a description of these SQL statements. A program may allocate several separate descriptor areas, identified by different descriptor names. SQL descriptor areas are used for both result set list items, routine parameters and input host variables.

The following statement types can use information from SQL descriptor areas:

- all SELECT statements and calls to result set procedures
- INSERT, DELETE, UPDATE and CALL statements using parameter markers
- ENTER statements

The following statement types do not use SQL descriptor areas:

- all data definition statements, security control statements, access control statements (except ENTER) and transaction control statements
- INSERT, DELETE, UPDATE and CALL statements using only constant expressions

In practice, programs using dynamically submitted SQL statements are usually written as though all submitted statements use SQL descriptor areas (since the nature of the submitted statement is not known until run-time). SQL descriptor areas can be left out of a program only if it is known in advance that they will not be needed (for instance in an application program which will handle only submitted data definition statements).

7.3.1 The structure of the SQL descriptor area

The SQL descriptor area is a storage area holding information about the described statement. It is allocated and maintained with embedded SQL statements. It consists of one COUNT field and one or more item descriptor areas, each one describing either a selected column or an input/output value:

COUNT
item descriptor area 1
item descriptor area 2
...
item descriptor area n

The COUNT field specifies how many of the item descriptor areas contain data.

The individual fields of the item descriptor area can be accessed with the GET DESCRIPTOR and SET DESCRIPTOR statements. See the Mimer SQL Reference Manual for a description of these statements.

7.4 Preparing statements

All statements submitted to dynamic SQL programs must be prepared. The simplest form of the operation uses a PREPARE statement (see the *Mimer SQL Reference Manual* for the syntax description). The operation may also be combined with EXECUTE as a simple statement in the shorthand form EXECUTE IMMEDIATE.

The source form of the statement must be contained in a host variable, containing the statement string. The statement string itself is not preceded by EXEC SQL nor terminated by the language-specific embedded delimiter.

The prepared form of the statement is named by an SQL-identifier or a host variable (for extended statements, see Section 7.5).

In the following example the source form of the statement is given as a string constant for illustrative purposes, however, the statement would usually be read from some input source, e.g. the terminal, at run-time:

```
...
exec sql BEGIN DECLARE SECTION;
SQL_TXT  CHARACTER(255);
...
exec sql END DECLARE SECTION;
...
SQL_TXT := "CREATE INDEX ROOMHOT ON ROOMS(HOTELCODE)";
exec sql PREPARE OBJECT FROM :SQL_TXT;
...
```

7.5 Extended dynamic cursors

A “normal” cursor is identified by an SQL identifier. An extended cursor makes it possible to represent a dynamic cursor by a host variable or a literal. An extended cursor is allocated by the application with the ALLOCATE CURSOR statement (see the *Mimer SQL Reference Manual* for the syntax description).

When the application is finished with the processing of the SQL statement, the prepared statement may be destroyed by executing the DEALLOCATE PREPARE statement (see the *Mimer SQL Reference Manual* for the syntax description). DEALLOCATE PREPARE also destroys any extended cursor that was associated with the statement.

Example of how extended cursors are used:

```
...
exec sql  BEGIN DECLARE SECTION;
SQL_TXT   CHARACTER(255);
C1        CHARACTER(128);
STM1      CHARACTER(128);
HOSTVAR1  INTEGER;
HOSTVAR2  CHARACTER(10);
...
exec sql  END DECLARE SECTION;
...
SQL_TXT := "SELECT COL1,COL2 FROM TAB1";
STM1 := "STATE_1";
exec sql PREPARE :STM1 FROM :SQL_TXT;
C1 := "CUR1";
exec sql ALLOCATE :C1 CURSOR FOR :STM1;
...
exec sql ALLOCATE DESCRIPTOR 'SQLA' WITH MAX 50;
exec sql DESCRIBE OUTPUT :STM1 USING SQL DESCRIPTOR 'SQLA';
...
exec sql OPEN :C1;
exec sql WHENEVER NOT FOUND GOTO done;
loop
    exec sql FETCH :C1 INTO SQL DESCRIPTOR 'SQLA';
    exec sql GET DESCRIPTOR 'SQLA' VALUE 1 :HOSTVAR1 = DATA;
    exec sql GET DESCRIPTOR 'SQLA' VALUE 2 :HOSTVAR2 = DATA;
    ...
    display HOSTVAR1,HOSTVAR2,...;
end loop;
done:
exec sql CLOSE :C1;
exec sql DEALLOCATE DESCRIPTOR 'SQLA';
exec sql DEALLOCATE PREPARE :STM1;
...
```

7.6 Describing prepared statements

Statements returning a result set and statements containing parameter markers can be described to obtain information about the number and data types of the parameters. There are two forms of DESCRIBE:

- DESCRIBE OUTPUT for result set values
- DESCRIBE INPUT for input and output parameters.

Both forms of DESCRIBE use the object (prepared) form of the statement as an argument. The same statement may be described in both senses if necessary.

For example:

```
exec sql BEGIN DECLARE SECTION;
SQLA1    CHARACTER(128);
MAXOCC   INTEGER;
SOURCE   CHARACTER(255);
exec sql END DECLARE SECTION;
...
MAXOCC := 15;
SQLA1 := "SQL_AREA_1";
exec sql ALLOCATE DESCRIPTOR :SQLA1 WITH MAX 20;
exec sql ALLOCATE DESCRIPTOR 'SQLA2' WITH MAX :MAXOCC;
...
exec sql PREPARE OBJECT FROM :SOURCE;
exec sql DESCRIBE OUTPUT OBJECT USING SQL DESCRIPTOR :SQLA1;
exec sql DESCRIBE INPUT OBJECT USING SQL DESCRIPTOR 'SQLA2';
...
```

DESCRIBE places information about the prepared statement in the SQL descriptor areas. See Section 7.3 for a description of the SQL descriptor area. The contents of the SQL descriptor area is read with the GET DESCRIPTOR statement and updated with the SET DESCRIPTOR statement.

7.6.1 Describing output variables

The items in the result set for a statement are described with the DESCRIBE OUTPUT statement. The keyword OUTPUT may be omitted.

The DESCRIBE OUTPUT statement shows:

- whether the statement returns a result set or not. This is indicated by the value of the COUNT field of the SQL descriptor area which is set to zero for statements that do not return a result set. Statements that return a result set are calls to result set procedures (see Section 8.6) and query-expressions (refer to the Mimer SQL Reference Manual). Dynamic SQL programs must test for this after each DESCRIBE operation because the treatment of statements that return result sets differs from the treatment of those that do not (see Section 7.7). If the statement returns a result set, the DESCRIBE statement will place information about the items in the result set in the fields of the descriptor area
- whether the current descriptor area allocation is sufficient or not. Insufficient area is indicated by the SQLSTATE variable set to a warning state and a value of COUNT (required number of items) greater than that specified in the "WITH MAX ..."-clause of the ALLOCATE DESCRIPTOR statement, or greater than 100 if no "WITH MAX..."-clause was specified. If the area is insufficient, no items are described.

7.6.2 Describing input variables

The DESCRIBE INPUT statement is used to describe parameter markers.

The value of the COUNT field of the SQL descriptor area indicates the number of parameter markers in the statement (a value of zero indicates no input parameters). A value greater than that specified in "WITH MAX..." indicates that the allocated SQL descriptor area is too small and the describe operation will not be performed. This situation is handled as described above for DESCRIBE OUTPUT.

Note: If the prepared statement is a call to a stored procedure that uses parameter markers, these will be described by the DESCRIBE INPUT statement. This is regardless of how the formal parameter is specified in the procedure definition. Whether the parameter is IN, INOUT or OUT can be seen from the PARAMETER_MODE field in the descriptor area.

7.7 Handling prepared statements

After PREPARE and DESCRIBE, the way in which submitted statements are handled differs according to whether the statement is executable or whether it returns a result set.

Executable — statements are executed using the EXECUTE statement, with the object (prepared) form of the submitted statement as the argument.

Result set — a cursor is used for these statements, associated with the object form of the prepared statement and are "executed" with OPEN and FETCH.

7.7.1 Executable statements

Executable statements are identified by a value of zero in the COUNT field of the SQL descriptor area after a DESCRIBE OUTPUT statement. If the statement does not contain any parameter markers, it may be executed directly. If on the other hand, the statement contains parameter markers, the statement must be executed with SQL descriptor areas for input and output values. Input values are specified in a USING clause while output values are specified in an INTO clause.

A statement has output values if it is a call to a stored procedure with parameters that are specified as INOUT or OUT.

Note: All parameter markers used in a call statement are described with the DESCRIBE INPUT statement, regardless of the mode of the formal parameter. A parameter marker with PARAMETER_MODE_INOUT must be present in the descriptors specified in the INTO and the USING clause of the EXECUTE statement but not necessarily with the same physical location in the host program.

Parameter markers must be used for all INOUT or OUT parameters when a call statement is prepared dynamically.

The pseudo code that follows uses the variables RESULT and MARKERS to flag the type of statement and the presence of parameter markers respectively.

```
exec sql ALLOCATE DESCRIPTOR 'SQLRESULT' with max 30;
exec sql ALLOCATE DESCRIPTOR 'INVAL' with max 30;
exec sql ALLOCATE DESCRIPTOR 'OUTVAL' with max 30;

exec sql DESCRIBE OUTPUT OBJECT USING SQL DESCRIPTOR RESULT;
exec sql GET DESCRIPTOR 'SQLRESULT' :NO_OUT = COUNT;

if NO_OUT = 0 then
  RESULT := FALSE;
else
  RESULT := TRUE;
end if;
```

```
exec sql DESCRIBE INPUT OBJECT USING SQL DESCRIPTOR 'INVAL';
exec sql GET DESCRIPTOR 'INVAL' :NO_IN = COUNT;

if NO_IN > 0 then
  MARKERS := TRUE;
else
  MARKERS := FALSE;
end if;

if RESULT then
  ...
else
if MARKERS then
  --
  -- loop over inval
  -- if PARAMETER_MODE_INOUT then copy to OUTVAL
  -- if PARAMETER_MODE_OUT then move to OUTVAL, compact INVAL
  -- count number of input values and output values
  --
else
  NO_INVAL := 0;
  NO_OUTVAL := 0;
end if;

if NO_INVAL > 0 and  NO_OUTVAL > 0 then
  exec sql EXECUTE OBJECT into OUTVAL using INVAL;
elsif NO_INVAL > 0 then
  exec sql EXECUTE OBJECT using INVAL;
elsif NO_OUTVAL > 0 then
  exec sql EXECUTE OBJECT into OUTVAL;
else
  exec sql EXECUTE OBJECT;
end if;
```

The descriptor areas referenced in the EXECUTE statement may be replaced by an explicit list of host variables, provided that the number and data types of the user variables in the source statement are known when the program is written (so that variables can be declared and the appropriate variable list written into the EXECUTE statement). This facility is of limited use, since the occasions when the user constructs freely chosen SQL statements with a predetermined number of user variables are rare.

The shorthand form EXECUTE IMMEDIATE combines the functions of PREPARE and EXECUTE. This form may only be used for executable statements with no parameter markers and is therefore of value only in contexts where the user is restricted to this type of statement. (Data definition and security control statements fall into this category, since user variables are not permitted in the syntax of these statements. EXECUTE IMMEDIATE can therefore be useful for application programs designed specifically to handle database definition statements).

7.7.2 Result set statements

Statements returning a result set are identified by a non-zero value in the COUNT field of the SQL descriptor area after DESCRIBE OUTPUT.

All dynamically submitted SELECT statements and calls to result set procedures must be handled through cursors. Cursors are declared or allocated for the object (prepared) form of submitted result set returning statements.

Note: A DECLARE CURSOR statement must precede the PREPARE statement in the program code. If ALLOCATE CURSOR is used instead of DECLARE CURSOR, the statement must have been prepared before the cursor can be allocated. The SQL statement must also be prepared before the cursor is opened.

If the source form of the result set returning statement contains parameter markers, these must be described before the cursor is opened and the OPEN statement must reference the relevant descriptor area. In the rare case where the number and data type of the user variables are known when the program is first written, the OPEN statement may reference an explicit variable list instead of a descriptor area.

The descriptor area used for the submitted result set returning statement is referenced when data is retrieved with the FETCH statement.

Example:

```
...
exec sql ALLOCATE DESCRIPTOR 'SQLA1' WITH MAX 30;
exec sql ALLOCATE DESCRIPTOR 'SQLA2' WITH MAX 30;
...
exec sql PREPARE 'OBJECT' FROM :SOURCE;
...
exec sql DESCRIBE OUTPUT 'OBJECT' USING SQL DESCRIPTOR 'SQLA1';
exec sql GET DESCRIPTOR 'SQLA1' :NO_OUT = COUNT;
if NO_OUT = 0 then
   RESULT_SET := FALSE;
else
   exec sql ALLOCATE 'C1' CURSOR FOR 'OBJECT';
   RESULT_SET := TRUE;
end if;
...
exec sql DESCRIBE INPUT 'OBJECT' USING SQL DESCRIPTOR 'SQLA2';
exec sql GET DESCRIPTOR 'SQLA2' :NO_IN = COUNT;
if NO_IN = 0 then
   USERVAR := FALSE;
else
   USERVAR := TRUE;
end if;
...
if RESULT_SET then
   if USERVAR then
      exec sql OPEN 'C1' USING SQL DESCRIPTOR 'SQLA2';
   else
      exec sql OPEN 'C1';
   end if;
   ...
   exec sql FETCH 'C1' INTO SQL DESCRIPTOR 'SQLA1';
   ...
else
   ...
end if;
```

7.8 Example framework for dynamic SQL programs

This section gives a general framework (in pseudo code) for dynamic SQL programs designed to handle any valid SQL statement as input. The framework is largely a synthesis of the example fragments given earlier in this chapter.

The framework is written as a single sequential module to emphasize the order of operations.

Host variable declarations are omitted. Handling of values returned by FETCH is also omitted.

See Appendix D for a complete example of how to use dynamic SQL in the C language.

Example framework (features that are specific to real host languages are described in Appendix A):

```
--
-- Allocate two descriptor areas
--
exec sql ALLOCATE DESCRIPTOR 'SQLA1' WITH MAX 50;
exec sql ALLOCATE DESCRIPTOR 'SQLA2' WITH MAX 50;
--
-- read statement from terminal
--
read INPUT into SOURCE;
--
-- prepare statement
--
exec sql PREPARE 'OBJECT' FROM :SOURCE;
--
-- describe statement and set type/parameter usage flags
--
exec sql DESCRIBE OUTPUT 'OBJECT' USING SQL DESCRIPTOR 'SQLA1';
exec sql GET DESCRIPTOR 'SQLA1' :NO_OUT = COUNT;
if NO_OUT = 0 then
   RESULT_SET := FALSE;
else
   RESULT_SET:= TRUE;
   --
   -- allocate cursor for SELECT statements
   --
   exec sql ALLOCATE 'C1' CURSOR FOR 'OBJECT';
end if;
--
exec sql DESCRIBE INPUT 'OBJECT' USING SQL DESCRIPTOR 'SQLA2';
exec sql GET DESCRIPTOR 'SQLA2' :NO_IN = COUNT;
if NO_IN = 0 then
   USERVAR := FALSE;
else
   USERVAR := TRUE;
end if;
-- execute statement or open cursor and fetch after assigning
-- values to input variables
--
if RESULT_SET then
   if USERVAR then
      exec sql OPEN 'C1' USING SQL DESCRIPTOR 'SQLA2';
   else
      exec sql OPEN 'C1';
   end if;
   loop
      exec sql FETCH 'C1' INTO SQL DESCRIPTOR 'SQLA1';
      exit when NO_MORE_REQUIRED or SQLSTATE = "02000";
      ...  -- process results of FETCH
   end loop;
   exec sql CLOSE 'C1';
else
   if USERVAR then
      exec sql EXECUTE 'OBJECT' USING SQL DESCRIPTOR 'SQLA2';
   else
      exec sql EXECUTE 'OBJECT';
   end if;
end if;
exec sql DEALLOCATE PREPARE 'OBJECT';
...
```

8 PERSISTENT STORED MODULES

It is possible to define and store routines in Mimer SQL. A number of routines may be collected together into a module. This chapter describes stored routines and modules, together referred to as Persistent Stored Modules (SQL/PSM).

SQL/PSM consists of syntax and semantics for variable and cursor declarations, assignment of the results of expressions to variables and parameters, conditional statements, control statements for looping and branching, condition handling, get diagnostics for status information and routine invocations.

8.1 Routines

In Mimer SQL, the term routine is used to collectively refer to functions and procedures. Essentially the same constructs may be used in each.

A routine can be created by declaring it in a module definition (see Section 8.3). or be created on its own by executing the CREATE FUNCTION or CREATE PROCEDURE statement. A routine created on its own cannot be subsequently added to a module.

A routine belongs to the schema in which it was created and the routine name may be qualified in the normal way with the name of the schema. Only the ident with the same name as the schema to which a routine belongs may refer to it by its unqualified name, all other idents must use the fully qualified routine name.

A given schema cannot contain more than one function or more than one procedure with the same name, i.e. a function cannot have the same qualified name (i.e. *schema_name.function_name*) as another function and a procedure cannot have the same qualified name as another procedure.

It is possible for a function to have the same qualified name as a procedure, because the invocation of a function is distinct from that of a procedure.

In order to invoke a routine, the ident invoking it must have been granted EXECUTE privilege on the routine. Routines may be recursively invoked.

Note: When routines and modules are created using BSQL, the create statement must be delimited by the "@" character (see Section 7.6 of the Mimer SQL User's Manual for details and examples).

The following points should be noted for procedures:

- they are invoked by using the CALL statement.
- any result from a procedure must be passed back via one of the output parameters, except in the special case of a result set procedure, which can return rows of a result set to a cursor (see Section 8.6).

The following points should be noted for functions:

- they are invoked from an SQL statement where a value is required (certain restrictions apply, see Section 8.4.8). Example:
  ```
  SET :name = get_author('1-55860-461.8').
  ```
- the parameters of a function provide input only and the function result is returned as the value of the function invocation.

A routine essentially consists of static SQL source which is stored in the data dictionary and which may be invoked by name whenever it is to be executed.

The SQL source for a routine comprises a definition of various routine components (see Section 8.2 for details) followed by the routine body.

The routine body consists of a single executable SQL statement, which may be a compound SQL statement (see Section 8.2.5) containing local declarations and a number of SQL statements, delimited by a BEGIN and END.

Note: It is recommended that a compound SQL statement always be used for the body of a routine, as this offers the greatest flexibility and results in a consistent structure for all routines.

It is possible to declare exception handlers within a compound SQL statement to handle specific exceptions or classes of conditions (see Section 8.7.2).

8.1.1 Functions

A function is invoked by specifying the function invocation where a value expression would normally be used. The parameters of a function are used to provide input only, values cannot be passed back to the calling environment through the parameters of a function.

A function always returns a single value and the data type of the return value is defined in the returns clause, which is specified after the parameter definition part of the function definition.

The function returns its value when a RETURN statement is executed within the body of the function. The data type of the value expression in the RETURN statement must be assignment-compatible with the data type specified in the returns clause of the function.

A function can be created with the same name as a predefined function (e.g. ABS, SIGN). If such a function also has the same number of parameters as the predefined function and each parameter has a data type that is assignment-compatible with the corresponding parameter of the predefined function, the invocations of the two functions cannot be distinguished. In this situation, any unqualified reference to the function name will be taken to be a reference to the **predefined** function.

The SQL statements that apply to a function are:

CREATE FUNCTION	create a function that exists on its own
DROP FUNCTION	drop a function that exists on its own
GRANT EXECUTE	grant the privilege to invoke a function
REVOKE EXECUTE	revoke the privilege to invoke a function
COMMENT ON FUNCTION	define a comment on a function

Refer to the Mimer SQL Reference Manual for a description of the SQL statements mentioned above.

Examples:

```
CREATE FUNCTION SQUARE_ROOT(ROOT INTEGER)
RETURNS INTEGER
CONTAINS SQL
BEGIN
  RETURN ROOT*ROOT;
END

CREATE FUNCTION COUNT_BILL(RESERVATION_NUMBER INTEGER)
RETURNS INTEGER
READS SQL DATA
BEGIN
  DECLARE X INTEGER;
  SELECT COUNT(*) INTO X FROM BILL WHERE RESERVATION =
RESERVATION_NUMBER;
  RETURN X;
END

CREATE FUNCTION TWIST(IN_CURRENCY CHAR(3))
RETURNS DECIMAL(6,3)
READS SQL DATA
BEGIN
   DECLARE CNT INTEGER;
   DECLARE L_RATE DECIMAL(6,3);
   SELECT RATE INTO L_RATE FROM EXCHANGE_RATE
                WHERE CURRENCY = IN_CURRENCY;
   GET DIAGNOSTICS CNT = ROW_COUNT;
   IF CNT = 0 THEN
       SET L_RATE = 1.0;
   END IF;
   RETURN L_RATE;
END

CREATE FUNCTION TRANSLATE_DATE(OLD_DATE CHAR(8))
RETURNS DATE
BEGIN
   RETURN CAST(SUBSTRING(OLD_DATE FROM 1 FOR 4) || '-' ||
                  SUBSTRING(OLD_DATE FROM 5 FOR 2) || '-' ||
                  SUBSTRING(OLD_DATE FROM 7 FOR 2) AS DATE);
END
```

8.1.2 Procedures

A procedure is normally invoked explicitly by executing the CALL statement and does not return a value. The parameters of a procedure can be used to provide input and may be used to pass values back to the calling environment.

There is a special type of procedure, called a result set procedure, which returns rows of a result set to a cursor when it is invoked by executing the FETCH statement in that context.

A result set procedure is distinguished from a normal procedure by having a values clause specified after the parameter definition part of the procedure definition (see Section 8.6 for a detailed description of result set procedures).

The SQL statements that apply to a procedure are:

CREATE PROCEDURE	create a procedure that exists on its own
DROP PROCEDURE	drop a procedure that exists on its own
GRANT EXECUTE	grant the privilege to invoke a procedure
REVOKE EXECUTE	revoke the privilege to invoke a procedure
CALL	invoke a procedure
COMMENT ON PROCEDURE	define a comment on a procedure

Refer to the Mimer SQL Reference Manual for a description of the SQL statements mentioned above.

Examples:

```
CREATE PROCEDURE MY_PROCEDURE ( IN TEST CHAR(8) )
NOT DETERMINISTIC
MODIFIES SQL DATA
BEGIN
  DECLARE SQLSTATUS CHAR(5) DEFAULT '?????';
  DECLARE ERRCNT INTEGER DEFAULT 0;
  DECLARE CASE_EXCEPTION CONDITION FOR SQLSTATE VALUE '20000';
  DECLARE TEST_SUCCESS   CONDITION FOR SQLSTATE VALUE 'Z0000';
  DECLARE TEST_FAILURE   CONDITION FOR SQLSTATE VALUE 'Z9999';

  CASE TEST
    WHEN '0414' THEN

      L0414:
      BEGIN
        DECLARE X CURSOR FOR SELECT   EMPNUM, HOURS
                             FROM     WORKS
                             WHERE    PNUM = 'P2'
                             ORDER BY EMPNUM DESC;
        SET CNT = 0;
        SET CNT2 = 0;
        BEGIN
          DECLARE LION CURSOR FOR SELECT   EMPNUM
                                  FROM     STAFF
                                  WHERE    EMPNUM = 'E20';
          P200:
          LOOP
             SET CNT = CNT + 1;
             BEGIN
               DECLARE EXIT HANDLER FOR NOT FOUND
               BEGIN
               END;

               OPEN X;
               SET I = 0;
               WHILE I < 20 DO
                  FETCH X INTO EMPNO1, HOURS1;
                  SET I = I + 1;
               END WHILE;
             END;
             CLOSE X;
             IF CNT = 5 THEN LEAVE P200; END IF;
             ...
          END LOOP P200;
      ...
```

```
    END L0414;
  WHEN '0415' THEN
     ...
  END CASE;
...
END

CALL MY_PROCEDURE('0415')

COMMENT ON PROCEDURE MY_PROCEDURE IS 'This is my procedure'

DROP PROCEDURE MY_PROCEDURE
```

8.2 Syntactic components of a routine definition

8.2.1 Routine parameters

A routine may have zero or more parameters and each parameter must have a name and a data type specified.

Each parameter of a **procedure** can have an optional mode specification (IN, OUT or INOUT - see CREATE PROCEDURE in the *Mimer SQL Reference Manual* for details). When the mode is not explicitly specified, IN is assumed by default.

It is not possible to specify the mode for the parameters of a **function** (they always have the default mode, IN).

A parameter name must be unique within the routine. A parameter can have the same name as a routine name, however this is not generally recommended. The parameter name can be up to 128 characters in length, see Section 4.2 of the Mimer SQL Reference Manual for further details about naming SQL objects.

The parameter data type can be any data type supported by Mimer SQL (see Section 4.3 in the Mimer SQL Reference Manual).

Note: A domain name cannot be specified for the data type of a routine parameter.

A parameter name may be referenced in an unqualified manner throughout a routine, at all scope levels - see Section 8.2.5 for a discussion of scope in routines.

Examples:

```
CREATE FUNCTION MY_FUNCTION(A INTEGER, B DECIMAL(5,2))
RETURNS DECIMAL(5,2)
BEGIN
   ...
   ...
END

CREATE PROCEDURE LOOKUP(IN I INTEGER, OUT R_VAL VARCHAR(20))
BEGIN
   ...
   ...
END
```

8.2.2 Routine language indicator

The language indicator specifies the language of the routine. Currently, the only language name supported is SQL.

If no language indicator is specified, LANGUAGE SQL is assumed by default.

8.2.3 Routine deterministic clause

The deterministic clause for a routine can specify NOT DETERMINISTIC or DETERMINISTIC. If a deterministic clause is not specified, NOT DETERMINISTIC is assumed by default.

A DETERMINISTIC routine is one that is guaranteed to produce the same result every time it is invoked with the same set of input values.

Therefore, a DETERMINISTIC routine must not contain a reference to:
CURRENT_DATE, LOCALTIME or LOCALTIMESTAMP.

Specifying a routine to be DETERMINISTIC allows repeated invocations of it to be optimized.

8.2.4 Routine access clause

The access clause for a routine specifies which SQL statements are permitted within the routine.

The three different options for the routine access clause (CONTAINS SQL, READS SQL DATA and MODIFIES SQL DATA) are described under CREATE PROCEDURE in the *Mimer SQL Reference Manual*.

If no routine access clause is specified, then CONTAINS SQL is implied.

8.2.5 Scope in routines - the compound SQL statement

A compound SQL statement allows a sequence of procedural SQL statements to be considered as a single SQL statement (see COMPOUND STATEMENT in the *Mimer SQL Reference Manual* for a description of the syntax).

A routine body may contain only **one** executable SQL statement and the compound SQL statement allows a routine to be defined which can actually contain any number of SQL statements.

A compound SQL statement also defines a local scope in which variables, exception handlers, and cursors can be declared. Compound SQL statements may be nested, one within the other, and thus local scopes may be nested.

A compound SQL statement may be labeled, which effectively names the local scope defined by it. The label name can be used whenever the scope environment needs to be referred to explicitly, e.g. when qualifying the names of objects which have been declared in the compound SQL statement. The label name must not be the same as a routine name.

It is important to understand the effect of scoping on declared items, particularly with respect to: out-of-scope references to variables (see Section 8.2.6), the scope within which an exception handler remains in effect and the flow of control effects following the use of different types of exception handler (see Section 8.7.2).

The SQL statement LEAVE is specifically provided to give the programmer the ability to force the flow of control to exit from a labeled scope.

Example:

```
CREATE PROCEDURE MY_PROCEDURE(INOUT Y INTEGER)
CONTAINS SQL
S0:
BEGIN
  ...
  S1:
  BEGIN
    IF Y < 0 THEN
      SET Y = 0;
      LEAVE S0;
    END IF;
    ...
  END S1;
  ...
END S0;
```

In the example above, the effect of the LEAVE statement is to pass flow of control to the statement END S0, i.e. flow of control exits from the scope labeled S0.

All open cursors declared in a compound SQL statement are closed whenever flow of control leaves the compound SQL statement for any reason.

Note: A compound SQL statement may be preceded by a label which names the scope delimited by the BEGIN and END (this is called the beginning label). Specifying the label next to the END is optional. However, if a label is specified next to the END, the beginning label must be specified.

8.2.5.1 The ATOMIC compound SQL statement

The execution of any SQL statement, other than a procedure-control-statement, is atomic. (See the beginning of Chapter 6 of the Mimer SQL Reference Manual for a definition of a procedure-control-statement.)

The execution of a compound SQL statement defined as ATOMIC is also atomic.

When the execution of an SQL statement is atomic, an atomic execution context becomes active while the statement, or any contained sub-query, is executing. While an atomic execution context is active, it is possible for another atomic execution context to become active within it.

While an atomic execution context is active the following is true:

- It is not possible to explicitly terminate a transaction, thus all changes made within the atomic execution context occur within the same transaction.
- If an SQL statement within the atomic execution context fails and no handler has been declared within it to handle the error, all the changes made within the atomic execution context will be cancelled and the associated exception will be propagated to the calling environment.
- If an SQL statement within the atomic execution context fails and an UNDO handler has been declared within it to handle the error, all the changes made within the atomic execution context will be cancelled and the handler will be executed.

An atomic compound SQL statement is defined by specifying the keyword ATOMIC next to the BEGIN delimiter. The COMMIT and ROLLBACK statements **cannot** be used within an atomic compound SQL statement.

A compound SQL statement is explicitly defined as not being atomic by specifying NOT ATOMIC next to the BEGIN delimiter. If nothing is specified next to the BEGIN delimiter, NOT ATOMIC is assumed by default.

If the compound SQL statement contains a declaration for an UNDO exception handler (see Section 8.7.2), the compound SQL statement **must** be ATOMIC.

Examples:

```
CREATE FUNCTION MY_ATOMIC_FUNCTION(I INTEGER)
RETURNS INTEGER
BEGIN ATOMIC
  ...
  -- All statements executed between this BEGIN
  -- and END execute within the same active atomic
  -- execution context.
  -- UNDO exception handlers are permitted.
  -- No COMMIT or ROLLBACK allowed!
  ...
END

CREATE PROCEDURE MY_NON_ATOMIC_PROCEDURE(I INTEGER)
BEGIN NOT ATOMIC
  ...
  -- This compound SQL statement is not atomic.
  -- COMMIT and ROLLBACK statements are permitted.
  -- No UNDO exception handlers allowed!
  ...
END

CREATE FUNCTION MY_DEFAULT_FUNCTION(I INTEGER)
RETURNS INTEGER
BEGIN
  ...
  -- This compound SQL statement is not atomic, by default.
  -- COMMIT and ROLLBACK statements are permitted.
  -- No UNDO exception handlers allowed!
  ...
END
```

8.2.6 Declaring variables

It is possible to declare variables, cursors, condition names and exception handlers at the beginning of a compound SQL statement. These items can, therefore, be declared in a routine when a compound SQL statement is used for the routine body.

This section discusses the declaration of variables. Discussions about declaring the other items mentioned above can be found in the following sections:

cursors	see Section 8.5.2.
condition names	see Section 8.7.1.
exception handlers	see Section 8.7.2.

Variables of any data type supported by Mimer SQL may be declared. The name of a variable must be unique within the scope of its declaration and must not conflict with the name of any of the routine parameters. Variable names can be a maximum of 128 characters in length and are case insensitive. See Section 4.2 of the Mimer SQL Reference Manual for further details on naming SQL objects.

Note: The data type for a variable must be specified explicitly, it is not possible to specify a domain.

More than one variable of the same type can be declared in a single variable declaration (see the examples below).

It is possible to specify an optional expression, which may be NULL, that defines the default expression for a variable declaration. The variable(s) created by the variable declaration are given the initial value derived from the default expression. If a default expression is not specified, the value NULL is assumed.

Examples:

```
DECLARE Z INTEGER;
DECLARE X, Y INTEGER DEFAULT 9;
DECLARE ABX VARCHAR(50);
DECLARE A INTEGER DEFAULT NULL;
```

Note: It is possible to declare a variable which has the same name as a column name in a table. All ambiguous references will be interpreted as a reference to a column name rather than a variable. It is therefore recommended that a suitable naming convention be adhered to that clearly distinguishes between the names of table columns and variables.

The name of a variable may be qualified in the normal way with the beginning label of the scope in which it has been declared.

Example:

```
CREATE PROCEDURE MY_PROCEDURE(IN X INTEGER)
S0:
BEGIN
  DECLARE A, B INTEGER;
  S1:
  BEGIN
    DECLARE B, C INTEGER;
    ...
  END S1;
  S2:
  BEGIN
    DECLARE Y INTEGER;
    ...
  END S2;
END S0;
```

The qualified names for the variables in the preceding example are as follows:

```
S0.A  S0.B  S1.B  S1.C and S2.Y.
```

8.2.7 The ROW data type

Mimer SQL supports a pseudo data type called the ROW data type. It can be used in a compound SQL statement in place of the data type specified when a variable is declared.

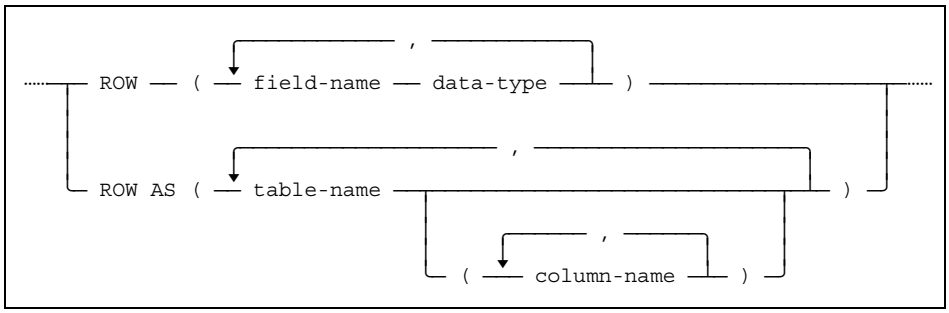
A variable which is declared as having the ROW data type implicitly defines a **row value**, which is a single construct that has a value which effectively represents a table row.

A row value is composed of a number of named values, each of which has its own data type and represents a column value in the overall row value.

A ROW data type can be defined either by explicitly specifying a number of field-name/data-type pairs or by specifying a number of table columns from which the unqualified names and data types are inherited.

8.2.7.1 ROW data type syntax

The syntax for defining a ROW data type is:



The following points apply to the specification of a ROW data type:

- A domain **cannot** be specified for *data-type*.
- The value specified for *data-type* can be a ROW data type specification.

- Two fields in the same ROW data type specification must **not** have the same name (this restriction applies equally to fields named by specifying a *field-name* value and those named by inheriting the unqualified name of a table column).
- If *table-name* is specified without a list of column names, **all** the columns in the table are used to define fields in the ROW data type.

8.2.7.2 Using the ROW data type

A ROW variable field is referenced like this: ***variable-name.field-name***.

A value may be assigned to one of the fields in a ROW variable in the same way as a value would be assigned to a variable declared with the same data type as the field. The data type of the field must be assignment compatible with the value being assigned to it.

If the declaration of a ROW variable does not include a DEFAULT clause, **each field** in the ROW variable is set to NULL initially.

The value of a field in a ROW variable may be used in the same way as any value of that type.

When a ROW data type is defined by specifying table columns, the names and data types of its fields are inherited from the columns in the table(s). Subsequently assigning values to the ROW variable will **not** affect the table(s) used to define the ROW data type.

A row value, which may be the value of a ROW variable, may be assigned to a ROW variable. The row value and the ROW variable are assignment-compatible if, and only if, both contain the same number of values and each value in the row value is assignment-compatible with the corresponding field in the ROW variable.

Two row values, one or both of which may be the value of a ROW variable, may be compared. The row values are comparison-compatible if, and only if, both contain the same number of values and each value in one is comparison-compatible with the corresponding value in the other.

A ROW variable may be used within a compound SQL statement in the following contexts:

- As the only expression specified in a RETURN statement used in a result set procedure. The ROW variable must be assignment-compatible with the row value defined by the procedure VALUES clause.
- As the only target variable specified in the INTO clause of a SELECT INTO statement. The row value selected must be assignment-compatible with the ROW variable and will be assigned to it.
- As the only target variable specified in the INTO clause of a FETCH statement. The row value fetched must be assignment-compatible with the ROW variable and will be assigned to it.

- As the procedure-variable or expression in a SET assignment statement (see the description above of assignment-compatibility involving ROW variables).
- As an argument in a comparison (see the description above of comparison-compatibility involving ROW variables).

8.2.8 Row value expression

A row value expression is an expression that specifies a row value. The values that represent the column values of the row value expression are specified as value expressions in a comma-separated list which is delimited by parentheses.

A row value expression can be used in the following contexts:

- As the only expression in a RETURN statement used in a result set procedure.
- As the expression following the DEFAULT keyword in a DECLARE VARIABLE statement for a variable declared to have the ROW data type.
- As a row value in a comparison or assignment operation.

Examples:

```
RETURN (24, 16, 'xyz', 11.3, X+4/9);

DECLARE RC ROW (A INTEGER, B INTEGER, S VARCHAR(10))
  DEFAULT (14, 27, 'hello');

IF RC = (14, 27, 'hello') THEN
  SET RC.S = 'bye';
END IF;

SET RC = (99, 105, 'new value');
```

8.3 Modules

A module is simply a collection of routines. All the routines in a module are created by declaring them when the module is created. Routines cannot be added to or removed from a module after the module has been created.

A module belongs to the schema in which it is created and all the routines contained in a module must belong to the same schema as the module.

The name of a routine in a module may be qualified in the normal way by using the name of the schema to which the routine belongs. The module name is never used to qualify the name of a routine.

Note: It is not possible to grant EXECUTE privilege on a module. In order to allow an ident to invoke a routine, whether it exists on its own or in a module, EXECUTE privilege on the routine must be granted to the ident.

When a module is dropped, all the routines in the module will be dropped as well. See Section 8.9 for a discussion of CASCADE effects on modules and routines.

The operations that may be performed on a module are:

CREATE using CREATE MODULE

DROP using DROP MODULE

COMMENT using COMMENT ON MODULE

Refer to the Mimer SQL Reference Manual for a description of the SQL statements mentioned above, brief examples follow.

Examples:

```
CREATE MODULE MY_MODULE
    DECLARE PROCEDURE P1 ... ;
    DECLARE PROCEDURE P2 ... ;
    DECLARE FUNCTION F1 ... ;
    ...
END MODULE

COMMENT ON MODULE MY_MODULE IS 'This is my example module'

DROP MODULE MY_MODULE CASCADE
```

8.4 SQL constructs in routines

The following SQL constructs are specifically provided for use in the body of a routine.

8.4.1 Assignment using SET

The SET statement is used to assign a value to a variable declared in a routine or an output parameter of a procedure (i.e. a parameter with mode OUT or INOUT).

Examples:

```
SET A = 5;
SET X = NULL;
SET Y = 11 + A;
SET D = CURRENT_DATE;
SET Z = NEXT_VALUE OF Z_SEQUENCE;
SET X = CASE Y WHEN 1 THEN Y ELSE 0 END;
```

8.4.2 Conditional execution using IF

The IF statement provides a mechanism for conditional execution of SQL statements based on the truth value of a conditional expression.

Note: If the conditional expression includes (or equals) NULL, the conditional expression evaluates to *false*. Testing for the NULL value must be done by using IS NULL (see Section 5.9.6 of the Mimer SQL Reference Manual).

A basic IF statement consists of a conditional expression followed by a list of one or more SQL statements in a THEN clause, which are executed if the conditional expression evaluates to *true* and, optionally, a list of one or more SQL statements in an ELSE clause which are executed if the conditional expression evaluates to *false*.

All of the predicates supported by Mimer SQL are allowed in the conditional expression of an IF statement - see Section 5.9 of the Mimer SQL Reference Manual.

One or more IF statements can be nested, one within the other, by using an ELSEIF clause in place of the ELSE clause in the IF statement containing another.

The IF statement does not in any sense define a local scope, it is simply a mechanism for conditionally executing a sequence of SQL statements.

Once the SQL statements to be executed have been selected, they execute in the same way as any ordinary sequence of SQL statements. This point is particularly important when considering exception condition handling behavior (see Section 8.7).

Examples:

```
IF X > 50 THEN
  SET X = 50;
  SET Y = 1;
ELSE
  SET Y = 0;
END IF;

IF Y IN (2,3,4) THEN
  ...
ELSE
  ...
END IF;

IF X > 50 THEN
  SET X = 50;
  SET Y = 2;
ELSEIF X > 25 THEN
  SET Y = 1;
ELSE
  SET Y = 0;
END IF;

IF NOT EXISTS (SELECT * FROM TABLE_1) THEN
  ...
ELSE
  ...
END IF;

IF X > (SELECT C1 FROM T2 WHERE ... ) THEN
  ...
ELSE
  ...
END IF;
```

8.4.3 Conditional execution - the CASE statement

The CASE statement provides another mechanism for conditional execution of SQL statements. The CASE statement comes in two forms, a “simple case” and a “searched case”.

A “simple case” works by evaluating equality between one value expression and one or more alternatives of a second value expression.

Example:

```
DECLARE Y INTEGER;

CASE Y
  WHEN 1 THEN ...
  WHEN 2 THEN ...
  WHEN 3 THEN ...
  ELSE  ...
END CASE;
```

A “searched case” works by evaluating, for truth, a number of alternative search conditions.

Example:

```
CASE
  WHEN EXISTS (SELECT * FROM BILL) THEN ...
  WHEN X > 0 OR Y = 1 THEN ...
ELSE ...
END CASE;
```

For both forms of the CASE statement the following is true:

- A sequence of one or more SQL statements can follow the THEN clause for each of the conditional alternatives, in the same way as for an IF statement, even though only a single implied SQL statement is shown in the examples above.
- Each alternative sequence of SQL statements in a CASE statement is treated in the same way, with respect to the behavior of exception handlers etc., as has already been described for sequences of SQL statements in an IF statement (see Section 8.4.2).
- Like the IF statement, the CASE statement simply provides a mechanism for selecting a sequence of SQL statements to execute. The CASE statement as a whole is not considered, in any sense, to be a single statement.
- The conditional part of each WHEN clause is evaluated, working from the top of the CASE statement down. The SQL statements that are actually executed are those following the THEN clause of the first WHEN condition to evaluate to true. If none of the WHEN conditions evaluate to true, the SQL statements following the CASE statement ELSE clause are executed.

The presence of an ELSE clause in the CASE statement is optional and if it is not present (and none of the WHEN conditions evaluate to true) an exception condition is raised to indicate that a case was not found for the CASE statement.

Note: If it is desired that there be no operation performed and no exception condition raised if none of the WHEN conditions evaluate to true, then an ELSE clause should be specified as an empty compound SQL statement.

Only the single selected sequence of SQL statements that follow a THEN or the ELSE is executed before the CASE statement terminates. There is no potential “fall through” to subsequent THEN sequences as is found in case statements in some other programming environments.

Note: The CASE statement is distinct from the CASE expression - see Section 5.6 of the Mimer SQL Reference Manual.

8.4.4 Iteration using LOOP

The LOOP statement may be preceded by a label that can be used as an argument to LEAVE in order to terminate the loop. The LOOP statement can contain a sequence of one or more SQL statements which are executed, in order, repeatedly.

The iteration is terminated by executing the LEAVE statement, or if an exception condition is raised.

Example:

```
L1:
LOOP
  IF ... LEAVE L1;
  ...
END LOOP L1;
```

8.4.5 Iteration using WHILE

The WHILE statement may be preceded by a label that can be used as an argument to LEAVE in order to terminate the while loop. The WHILE statement can contain a sequence of one or more SQL statements which are executed, in order, repeatedly.

The WHILE statement includes a conditional expression and iteration continues as long as this expression evaluates to true. Iteration may also be terminated by executing the LEAVE statement, or if an exception condition is raised.

Example:

```
SET I = 0;
L1:
WHILE I <= 10 DO
  ...
  SET I = I + 1;
END WHILE L1;
```

8.4.6 Iteration using REPEAT

The REPEAT statement may be preceded by a label that can be used as an argument to LEAVE in order to terminate the repeat loop. The REPEAT statement can contain a sequence of one or more SQL statements which are executed, in order, repeatedly.

The REPEAT statement includes an UNTIL clause, which specifies a conditional expression, and iteration continues until this expression evaluates to true. Iteration may also be terminated by executing the LEAVE statement, or if an exception condition is raised.

Example:

```
SET I = 0;
L1:
REPEAT
  ...
  SET I = I + 1;
UNTIL I > 10
END REPEAT L1;
```

8.4.7 Invoking procedures - CALL

The CALL statement is used to invoke a procedure. The name of the procedure may be qualified with the name of the schema to which it belongs. A value expression or target variable must be specified for each of the procedure's parameters (see Section 4.2.6 of the Mimer SQL Reference Manual for the definition of a target variable).

If the procedure parameter has mode OUT or INOUT, a target variable must be specified. For procedure parameters with mode IN, a value expression may be specified.

SQL/PSM is not strongly typed, so the expression specified for each procedure parameter need not have exactly the same data type as the parameter, however the expression must be assignment-compatible with the procedure parameter for which it is supplied (see Section 4.5 of the Mimer SQL Reference Manual for a discussion of assignment and implicit data type conversions).

Examples:

```
CALL PROC1();
CALL PROC2(X,Y);
CALL IDENT1.PROC7(CURRENT_DATE, X+3, Z);
```

8.4.8 Invoking functions - use as a value expression

Functions are not invoked by calling them explicitly. A function is invoked, and it returns its value, when it is used in a procedure-control-statement or in an assignment where a value-expression would normally be used.

The name of the function may be qualified with the name of the schema to which it belongs.

For definition of procedure-control-statement, see the beginning of Chapter 6 of the Mimer SQL Reference Manual.

The following restrictions apply:

- If the function is defined as NOT DETERMINISTIC, it must **not** be used in the expression that follows the CASE keyword or the WHEN keyword in the simple case form of a CASE statement (see Section 8.4.3).
- If MODIFIES SQL DATA has been specified for the *access-clause* of the function, it must **not** be used in the expression following the DEFAULT keyword in a DECLARE VARIABLE statement.

Examples:

```
IF MY_FN(X) > 70 THEN
  ...
ELSE
  ...
END IF;

SET X = NEXT_AVAIL(TOP, LAST_ALLOC);

SELECT TRANSGRESS(ARRIVE,DEPART) FROM BOOK_GUEST WHERE ... ;

INSERT INTO ROOM_PRICES VALUES(CODE_MAP('LAPONIA'), ... ;

SELECT * FROM ROOM_STATUS WHERE BOOKADM.STATES(STATUS) = ... ;

CREATE TABLE OFFICES ( OFFICE_NAME CHAR(20), CHECK
(CHECK_NAMES(OFFICE_NAME) = 0), ... ;
```

8.4.9 Comments in routines

Any text that occurs after "--" and before end-of-line in a routine is taken to be a comment.

Example:

```
CREATE PROCEDURE MY_PROCEDURE(Y INTEGER)
-- This is a comment: Note that Y has mode IN (default)
READS SQL DATA
BEGIN
DECLARE B INTEGER;
-- Here is another comment
SET B = Y + 22; -- Y is input to the procedure
...
END
```

8.4.10 Restrictions

The following groups of SQL statement may not be used in a routine:

- Access Control statements
- Data Definition statements
- Embedded SQL Control statements
- Security Control statements
- Dynamic SQL statements
- System Administration statements.

Refer to beginning of Chapter 6 in the Mimer SQL Reference Manual for a definition of the statement groups mentioned above.

Note: Any SQL statements used in a routine must be executable, so the usual restriction on the use of SELECT versus SELECT INTO applies (only the latter being considered executable - the former may, however, be used in a conditional expression, e.g. in an IF statement or a cursor declaration).

The following restrictions apply to result set procedures:

- A COMMIT or ROLLBACK statement must **not** be executed in a result set procedure because it will interfere with the open cursor that will exist in the context from where the result set procedure is called.

- A function or procedure which executes a COMMIT or ROLLBACK statement must **not** be invoked from within a result set procedure.
- A function or procedure which has MODIFIES SQL DATA specified for its access clause must **not** be invoked from within a result set procedure.

8.5 Data manipulation

8.5.1 Write operations

INSERT, UPDATE and DELETE statements may be used in a routine provided MODIFIES SQL DATA has been specified for the access clause (see Section 8.2.4).

Routine parameters and variables may be used in these statements wherever an expression can normally be used, as shown in the examples below.

Example:

```
CREATE PROCEDURE INSERT_HOTEL(IN I_NAME CHAR(15),
                                 I_CITY CHAR(20),
                                 I_OVERBOOK DEC(3,2))
MODIFIES SQL DATA
BEGIN
   INSERT INTO HOTEL VALUES(SUBSTRING(I_NAME FROM 1 FOR 4),
                            I_NAME,I_CITY,I_OVERBOOK);
...
```

The ROW_COUNT option of the GET DIAGNOSTICS statement may be used immediately after an INSERT, UPDATE, DELETE, SELECT INTO or FETCH statement to determine the number of rows affected by the preceding statement.

Example:

```
DECLARE ROWS INTEGER;
...
INSERT INTO ROOMS SELECT ...;
GET DIAGNOSTICS ROWS = ROW_COUNT;
IF ROWS > 0 THEN
```

Note: All SQL statements except GET DIAGNOSTICS will overwrite the information in the diagnostics area.

8.5.2 Using cursors

Cursors may be declared and used in a compound SQL statement to receive a result set from a query-expression or from a result set procedure. A cursor may not have the same name as another cursor declared in the same scope.

Cursors in a procedural usage context are used in much the same way, in terms of the SQL statements used, as cursors declared outside routines. It is possible to open cursors, fetch data into variables and use the statements UPDATE and DELETE where current of cursor.

Examples:

```
BEGIN
  DECLARE X CURSOR FOR SELECT CHARGE_CODE,AMOUNT
                         FROM BILL FOR UPDATE;
  DECLARE I_CHARGE_CODE CHAR(3);
  DECLARE I_AMOUNT DEC(8,2);
  ...
  OPEN X;
L1:
  BEGIN
    DECLARE EXIT HANDLER FOR NOT FOUND
    BEGIN
    END;
    LOOP
       FETCH X INTO I_CHARGE_CODE,I_AMOUNT;
       IF I_CHARGE_CODE = '270' AND ... THEN
          UPDATE BILL SET AMOUNT = AMOUNT * 1.10 WHERE CURRENT OF X;
       END IF;
    END LOOP;
  END;
  CLOSE X;
END

DECLARE D DATE DEFAULT CURRENT_DATE;
DECLARE C1,C2 CHAR(5);
DECLARE Z SCROLL CURSOR FOR CALL MY_PROC(1,D);
DECLARE I INTEGER;
...
OPEN Z;
...
FETCH FIRST FROM Z INTO C1;
...
FETCH ABSOLUTE I FROM Z INTO C2;
```

The first example above demonstrates detection of the NOT FOUND exception as a method of checking that a fetch statement does not return any data. If a NOT FOUND exception occurs in the example, an exit handler is invoked. After the exit handler has finished, the flow of control leaves the compound SQL statement labeled L1.

Alternatively, the GET DIAGNOSTICS statement can be used to retrieve the number of rows affected by the FETCH statement, as shown below.

Example:

```
DECLARE ROWS INTEGER;
L1:
LOOP
   FETCH X INTO I_CHARGE_CODE,I_AMOUNT;
   GET DIAGNOSTICS ROWS = ROW_COUNT;
   IF ROWS = 0 THEN
      LEAVE L1;
   END IF;
END LOOP;
CLOSE X;
```

The following specific restrictions apply to cursors used in routines:

- no dynamic functions can be used (i.e. extended cursor names and the use of SQL descriptors)
- REOPENABLE cursors are not allowed
- the use of the keyword RELEASE with the CLOSE statement is not permitted.

Using FETCH to get result set data from a result set procedure may cause parts of the result set procedure to execute (see Section 8.6). The result set procedure will be “in use” until the associated cursor is closed.

8.5.3 SELECT INTO

Another way of fetching data is by using a SELECT INTO statement. This can only be used when a single record is fetched from the database. If more than one record fulfills the search criteria, an exception condition is raised.

Example:

```
DECLARE TOTAL INTEGER;
SELECT SUM(AMOUNT) INTO TOTAL
  FROM BILL WHERE RESERVATION = IN_RESERVATION;
```

8.5.4 Transactions

It is possible to start and end transactions within a routine. A transaction is implicitly started when a routine that accesses the database is invoked. It is also possible to explicitly start a transaction by using the START statement. When a transaction is ended, either by a COMMIT or ROLLBACK statement, all open cursors in the routine are closed.

It is possible to affect the behavior of transactions by using the SET TRANSACTION and SET SESSION statements.

Note: If a compound SQL statement is defined as ATOMIC, a transaction **cannot** be terminated within it because execution of the COMMIT or ROLLBACK statements is not permitted.

8.6 Result set procedures

A result set procedure is a special type of procedure that allows a result set to be returned. A result set procedure is called by specifying it in a cursor declaration and then using FETCH to get the result set data. In interactive SQL a result set procedure is called by using the CALL statement and the result set data is dealt with in the same way as a select.

Example (embedded SQL):

```
EXEC SQL DECLARE X CURSOR FOR CALL MY_RESULT_PROC(1, 5);
```

A result set procedure is distinguished when it is created or declared by a VALUES clause which follows the parameter part of the procedure definition. The VALUES clause defines the data types of the columns in the result set and may contain an AS clause which names the columns.

Example:

```
CREATE PROCEDURE MY_RESULT_PROC(IN A INTEGER, IN B INTEGER)
                 VALUES ( VARCHAR(32), INTEGER(10) )
                     AS ( CLIENT_NAME, CLIENT_ID )
READS SQL DATA
BEGIN
...
END;
```

All result set procedure parameters have mode IN, therefore any data returned from a result set procedure is returned via the procedure's result set.

The option MODIFIES SQL DATA must not be specified for the access clause of a result set procedure (see Section 8.2.4).

Note: A function or procedure which has MODIFIES SQL DATA specified for its access clause must **not** be invoked from within a result set procedure.

A result set procedure must **not** execute a COMMIT or ROLLBACK statement. There is always at least one open cursor during the execution of a result set procedure and these statements cause all open cursors to be closed.

Note: A function or procedure which executes a COMMIT or ROLLBACK statement must **not** be invoked from within a result set procedure.

A row in the result set of a result set procedure is returned by executing the RETURN statement. The arguments to a RETURN statement can be NULL, an expression or a variable which has the ROW data type.

When a FETCH is executed, the SQL statements in the body of the result set procedure are executed until a RETURN statement is executed.

The execution of the result set procedure is then suspended until the next FETCH statement is executed for the calling cursor, then flow of control within the result set procedure continues until the next RETURN statement is encountered, or until the end of the procedure is reached.

After flow of control has exited from the scope of a result set procedure the next attempt to FETCH more data into the calling cursor will flag end-of-set.

Thus, a result set procedure call can be used in place of the usual SELECT when declaring a cursor.

The following example (using embedded SQL) is intended to demonstrate how execution within the result set procedure proceeds, and is suspended, in response to FETCH statements being executed for the calling cursor:

```
EXEC SQL CREATE PROCEDURE MY_RESULT_PROC(X INTEGER)
                          VALUES (VARCHAR(10), INTEGER)
                              AS (TX, XP)
CONTAINS SQL
BEGIN
  DECLARE XP INTEGER DEFAULT X;

  RETURN ('FIRST ROW', XP);
  SET XP = X*2;
  RETURN ('SECOND ROW', XP);
  SET XP = X*3;
  RETURN ('THIRD ROW', XP);
END;
```

```
EXEC SQL DECLARE Z CURSOR FOR CALL MY_RESULT_PROC(3);

EXEC SQL WHENEVER NOT FOUND GOTO done;

EXEC SQL FETCH Z INTO :T, :X;
(This will fetch 'FIRST ROW', 3)
Result set procedure flow of control suspended at XP=X*2

EXEC SQL FETCH Z INTO :T, :X;
(This will fetch 'SECOND ROW', 6)
Result set procedure flow of control suspended at XP=X*3

EXEC SQL FETCH Z INTO :T, :X;
(This will fetch 'THIRD ROW', 9)
Result set procedure flow of control suspended at END;

EXEC SQL FETCH Z INTO :T, :X;
Flow of control exits from procedure scope
 and the NOT FOUND exception is raised.

done:
EXEC SQL CLOSE Z;
```

More typically, a loop construct would be used in the result set procedure to deal with RETURN statements. It is also permissible to use a cursor within the result set procedure to get data to be returned via a SELECT.

Closing the cursor for a result set procedure will close any open cursors declared within it and no further execution of the procedure will occur. Reopening the cursor will start execution of the result set procedure afresh from the beginning (i.e. no state information is saved between a close and reopen).

8.7 Managing exception conditions

An exception is raised if an error occurs when executing an SQL statement. Every exception is identified by an exception condition, expressed in terms of its SQLSTATE value.

An SQLSTATE value is represented by the keyword SQLSTATE followed by a 5-character string containing only uppercase alphanumeric characters. The first two characters of the string identify the exception class and the last three the exception sub-class.

In Mimer SQL, the range of possible SQLSTATE values is divided into standard values and implementation-defined values. The implementation-defined values are those beginning with the characters “J-R”, “T-Z”, “5-6” and “8-9”.

Whenever an exception is raised, the exception condition is placed in the diagnostics area and the SQLSTATE value can be retrieved by using the RETURNED_SQLSTATE option of GET DIAGNOSTICS.

In addition to expressing an exception condition in terms of its SQLSTATE value, it is possible (within a compound SQL statement) to declare a condition name to represent it (see Section 8.7.1). Whenever a condition name is used, it is immediately translated into the SQLSTATE value it represents.

It is possible to raise an exception without an error occurring by using the SIGNAL statement. When the SIGNAL statement is used, the specified exception condition is placed in the cleared diagnostics area (expressed as its SQLSTATE value) and control proceeds as if an error had just occurred.

Example:

```
SIGNAL SQLSTATE 'UE456';
```

It is possible to declare exception handlers in a compound SQL statement that perform some action when exceptions are raised. The action defined by the exception handler is associated with one or more specific exception conditions, or one or more exception class groups, specified when the exception handler is declared (see Section 8.7.2).

If there is an exception handler action defined for an exception condition that is raised, the exception handler action is performed and execution continues in the manner defined by the type of the exception handler (see Section 8.7.2).

If no exception handler action has been defined for an exception condition that is raised, the default error handling mechanism is invoked (which usually makes the exception condition visible to the calling environment).

If the exception NOT FOUND or an SQLWARNING is raised in an unhandled situation, execution will continue and the exception will be cleared by execution of the next statement in the procedure. The GET DIAGNOSTICS statement can be used to test for the NOT FOUND exception and an SQLWARNING.

It may be necessary for an exception handler action to re-raise the current exception condition or to raise an alternative exception condition. The RESIGNAL statement is provided for this purpose and it may only be executed from within an exception handler.

If RESIGNAL is executed without specifying an exception condition, the current exception condition remains in the diagnostics area and the error handling mechanism proceeds to deal with the error as if the current exception handler action had not been found. If an exception condition is specified (in the same way as for SIGNAL), this is pushed onto the top of the stack of exceptions in the diagnostics area, becoming the current SQLSTATE value, and the error handling mechanism proceeds as just described. The size of the exceptions stack in the diagnostics area is set by using the SET TRANSACTION DIAGNOSTICS SIZE statement (see Section 6.2.5).

Use of RESIGNAL is useful in situations where there are nested exception handler actions defined and it is required that an enclosing exception handler action be invoked from an inner one, or where the default error handling mechanism is to be allowed to proceed from some point within a defined exception handler action.

Examples:

```
RESIGNAL;

RESIGNAL SQLSTATE 'UE456';
```

8.7.1 Declaring condition names

As discussed above, exception conditions are identified by an SQLSTATE value. Whenever an exception is raised, the exception condition that identifies it is stored in the diagnostics area in the form of its SQLSTATE value.

It is always possible to specify an exception condition by using its SQLSTATE value, e.g. SQLSTATE VALUE 'S0700', however it is often desirable to declare a condition name that represents the SQLSTATE value in a way that more meaningfully describes the exception.

Condition names may be declared in a compound SQL statement, see the Mimer SQL Reference Manual for a detailed description of the compound SQL statement.

Example:

```
DECLARE INVALID_PARAMETER CONDITION FOR SQLSTATE 'UE456';
...
SIGNAL INVALID_PARAMETER;
```

Following this declaration, the condition name “INVALID_PARAMETER” can be used instead of the SQLSTATE value “SQLSTATE VALUE 'UE456'” whenever there is a need to refer to this exception condition.

Whenever a condition name is used (in an exception handler declaration or when signaling an exception condition) it is immediately translated into the SQLSTATE value it represents, i.e. the condition name itself is never placed in the diagnostics area or registered against an exception handler action.

All SQLSTATE values in Mimer SQL which lie outside the range of standard values are treated as implementation-defined, so all SQLSTATE values are handled in the same way and may be specified explicitly in all situations.

8.7.2 Declaring exception handlers

Exception handlers may be declared in a compound SQL statement in order to define an action which will be executed if specified exceptions are raised within the scope of the exception handler.

The structure of the handler action is the same as the body of a routine, i.e. a single executable procedural SQL statement. The exceptions to which the handler action will respond may be specified as a list of exception conditions or by specifying one or more exception class groups.

The exception class groups are:

- SQLWARNING covers SQLSTATE values beginning with “01”.
- NOT FOUND covers SQLSTATE values beginning with “02”.
- SQLEXCEPTION covers all other SQLSTATE values (including those in the implementation defined range), excluding those beginning with “00”.

An exception handler which is declared to respond to one or more exception class groups is referred to as a **general exception handler**.

An exception condition may be specified by its SQLSTATE value or a condition name declared to represent it. An exception handler which is declared to respond to one or more specific exception conditions is referred to as a **specific exception handler**.

The same exception condition must not be specified more than once in the same exception handler declaration.

An exception handler can either be a general exception handler or a specific exception handler, i.e. an exception handler declaration cannot contain both exception class groups and specific exception conditions.

Exception handlers are declared in the local handler declaration list of a compound SQL statement and the scope of an exception handler is that compound SQL statement plus all the SQL statements contained within it, including all other compound SQL statements nested within it.

The exception handler will be executed if one of the exceptions it is declared to respond to is raised within the scope of the handler.

A local handler declaration list can only contain one exception handler declared to respond to a particular exception condition or exception class group.

It is possible to declare a general and a specific exception handler, both of which cover the same scope, where an exception condition specified for the specific handler is in one of the exception class groups specified for the general handler. If the exception condition is raised in this situation, the specific handler is executed in preference to the general handler.

It is possible for the scope of two specific exception handlers, which respond to the same exception condition, to overlap. This will be the case if there are two nested compound SQL statements and each declares a specific exception handler for the same exception condition (this is permitted, provided the two exception handlers are not declared in the same local handler declaration list). In this situation the innermost exception handler action will be executed.

The same is true for two general exception handlers in this situation.

The RESIGNAL statement can be used in situations like this, in the inner exception handler action, to get the outer exception handler action to execute by propagating the exception out from the exception handler action which is currently executing.

Exception handlers fall into the following types:

Exit Handler: This type of exception handler will execute when the exception condition(s) that apply to it are raised. After the handler has executed, flow of control exits the scope of the compound SQL statement containing the exception handler declaration, by effectively performing a LEAVE (see Section 8.2.5).

Continue Handler: This type of exception handler will execute when the exception condition(s) that apply to it are raised. After the handler has executed, flow of control continues by executing the SQL statement immediately following the SQL statement that raised the exception.

Undo Handler: The execution of this type of handler will be initiated when the exception condition(s) that apply to it are raised. Before the handler action executes, all changes made by the executed SQL statements in the compound SQL statement, or by any SQL statements triggered by them, are canceled. The handler action is then executed and flow of control exits the scope of the compound SQL statement containing the exception handler declaration, by effectively performing a LEAVE (see Section 8.2.5).

Note: An UNDO exception handler can only be declared in a compound SQL statement which has been defined as ATOMIC (see Section 8.2.5.1).

Examples:

```
S1:
BEGIN
  DECLARE EXIT HANDLER FOR SQLEXCEPTION
  BEGIN
    ...
    ...
  END;
  ...
  ...
END S1;

S2:
BEGIN
  DECLARE CONTINUE HANDLER FOR SQLSTATE 'S0700'
  BEGIN
    ...
    ...
  END;
  ...
  ...
END S2;

S3:
BEGIN
  DECLARE EXIT HANDLER FOR SQLSTATE 'S0700'
  BEGIN
    ...
  END;
  ...
  S4:
  BEGIN ATOMIC
    DECLARE UNDO HANDLER FOR SQLSTATE 'S0700'
    BEGIN
      ...
    END;
    ...
  END S4;
  ...
END S3;
```

The GET DIAGNOSTICS statement can be used in a general exception handler to get the specific SQLSTATE value that provoked execution of the exception handler.

Example:

```
DECLARE STATE CHAR(5) DEFAULT '?????';
DECLARE CONTINUE HANDLER FOR SQLEXCEPTION
BEGIN
  GET DIAGNOSTICS EXCEPTION 1 STATE = RETURNED_SQLSTATE;
  CASE STATE
  WHEN '22003' THEN ...
  WHEN '20000' THEN ...
  ELSE ...
  END CASE;
END;
```

When the GET DIAGNOSTICS statement is used inside a routine it is important to keep in mind how exceptions are handled within routines, because this affects how and where the statement can be used.

When an exception is raised inside a routine there are two possibilities:

1. An exception handler has been declared to respond to the exception, it is invoked and the diagnostics area is cleared when the exception handler finishes. In this situation the GET DIAGNOSTICS statement can be used as the **first** statement in the exception handler to get the SQLSTATE value of the exception (it must be the first statement, because executing any other SQL statement will clear the diagnostics area).

2. If there is no exception handler declared to handle the exception, the default error handling mechanism will immediately resignal the exception to the calling environment (typically an enclosing compound SQL statement), therefore there is no opportunity to execute GET DIAGNOSTICS directly after the statement causing the error.

8.8 Access rights

The ident creating a routine must, as is usual, have the appropriate access rights on the tables and other database objects referenced from the SQL statements in the routine. The creating ident must also have the right to create objects in the schema to which the routine is to belong (i.e. the ident must be the creator of the schema).

The right of the creator to access referenced database objects is verified when the CREATE FUNCTION, CREATE MODULE or the CREATE PROCEDURE statement is executed.

If an ident wishes to invoke a routine, that ident must have EXECUTE privilege on the routine.

Note: In order for the creator of a routine to grant EXECUTE privilege on the routine to another ident, the creator must have the WITH GRANT option in affect for **all** the access rights held on all the database objects referenced within the routine.

The above note is an important security point, because granting EXECUTE privilege on a routine is effectively granting appropriate access rights to the given ident on all the database objects referenced in the routine, therefore all those access rights must be held by the grantor with the WITH GRANT option.

An ident may be granted EXECUTE privilege on a routine with the WITH GRANT option and if this option is in affect, the ident may grant EXECUTE privilege on that routine to other idents.

Routines can be used as a security layer in the database. By having EXECUTE privilege on a routine granted, an ident only gets the right to perform the specific operations specified in the routine and not general access to the referenced database objects.

Note: It is not possible to grant EXECUTE privilege on a module, only on routines.

8.9 Using DROP and REVOKE

Care must be taken if database objects are dropped when the CASCADE option is used and when REVOKE is used, particularly with respect to routines and modules.

It is important to bear in mind the following points in connection with modules and routines:

- Dropping an object referenced by an SQL statement in a routine will cause the routine to be dropped.
- If the access rights on a database object are revoked from the creator of a routine that contains an SQL statement referencing the object, the routine will be dropped.
- If a routine belonging to a module is dropped because of the effects of a cascade, the routine is effectively removed from the module (i.e. the module is not dropped).

If an ident attempts to drop a routine for which there is a compiled version currently being held by another ident, the DROP operation will fail because the routine is “in use”.

When a routine is invoked, it is compiled and the compiled version of the routine is held by the invoking ident. Any other idents invoking a routine while a compiled version of it exists will use the existing compiled version and this will be held by them as well.

A compiled version of a routine will generally be held by an ident until the ident disconnects. If the routine invocation is contained in a dynamic SQL statement, deallocating the statement will release the compiled version of the routine immediately without the need for a disconnect.

9 TRIGGERS

A trigger defines an SQL statement that is to be executed before, after or instead of a specified data manipulation operation on a named table or view.

The execution of the SQL statement can be made conditional on the evaluation of a search condition.

The SQL statement is typically a compound SQL statement, thus allowing a number of SQL statements to be executed by the trigger. The compound SQL statement must be defined as ATOMIC.

Thus, the body of a trigger is similar to the body of a routine and the same constructs may be used within it.

9.1 Creating a trigger

A trigger is created by using the CREATE TRIGGER statement (see Chapter 6 of the Mimer SQL Reference Manual).

Examples:

```
CREATE TRIGGER LOG_DATA BEFORE DELETE ON BOOK_GUEST
BEGIN ATOMIC
  ...
END;

CREATE TRIGGER LOG_DATA AFTER INSERT ON BOOK_GUEST
REFERENCING NEW TABLE AS GUESTS
BEGIN ATOMIC
  DECLARE GUEST_CURSOR CURSOR FOR SELECT * FROM GUESTS;
  ...
END;

CREATE TRIGGER JOIN_UPDATE INSTEAD OF UPDATE ON INVOICE
REFERENCING OLD TABLE AS OLD_INVOICE
            NEW TABLE AS NEW_INVOICE
 BEGIN ATOMIC
   DECLARE OLD_INVOICE CURSOR FOR SELECT * FROM OLD_INVOICE;
   DECLARE NEW_INVOICE CURSOR FOR SELECT * FROM NEW_INVOICE;
   ...
END;
```

A trigger is created on a named table or view and the trigger must be created in the schema to which the table or view belongs. If the trigger is created on a base table, the table **must** be stored in a databank with the TRANS or LOG option.

The trigger name must follow the rules for naming private database objects (see Section 4.2.2 of the Mimer SQL Reference Manual) and the name must be unique within the schema in which the trigger is created.

It is possible to create any number of triggers on a named table, each of which may have the same trigger time (see Section 9.2) and trigger event (see Section 9.3) specified.

If two or more triggers exist on the same table with the same trigger time and trigger event, they will be executed in an order determined by their internal creation timestamp.

Note: When triggers are created using BSQL, the create statement must be delimited by the "@" character.

9.2 Trigger time

The trigger time specifies when, in relation to the execution of the triggering data manipulation statement, the trigger is executed.

The possible values for the trigger time are:

AFTER — this specifies that the trigger will be executed following the execution of the triggering data manipulation statement.

BEFORE — this specifies that the trigger will be executed prior to the execution of the triggering data manipulation statement.

INSTEAD OF — this specifies that the trigger will execute when the triggering data manipulation statement would normally be executed. In this case the triggering data manipulation statement itself has no direct effect, it only causes the trigger to execute.

If the trigger time specified is AFTER or BEFORE, the table name must specify a **base table** which is located in a databank with the TRANS or LOG option.

If the trigger time specified is INSTEAD OF, the table name must specify a **view**.

9.3 Trigger event

The trigger event specifies the data manipulation statement that will cause the trigger to execute. The possible values for the trigger event are: INSERT, UPDATE and DELETE.

The trigger will be executed once each time the specified data manipulation statement is executed on the table on which the trigger was created.

Note: If the trigger time is INSTEAD OF, the trigger event itself has no effect on the table, it just causes the trigger to execute. The environment executing the trigger event behaves **as if** the data manipulation statement is actually being executed, even though no changes actually occur in the table(s) that would normally be affected. The only data manipulations possible in this case are those performed by the trigger action.

9.4 Trigger action

The trigger action, like the body of a routine, consists of a single procedural SQL statement. In addition, the execution of the SQL statement can be made conditional on the evaluation of a search condition.

The search condition is specified in the optional WHEN clause of the CREATE TRIGGER statement.

As for routines, it is recommended that a compound SQL statement always be used for the trigger action.

Note: The entire trigger action must be executed in a single atomic execution context, therefore if a compound SQL statement is used, it must be defined as ATOMIC (see Section 8.2.5.1).

The SQL statement(s) of the trigger action are always executed within the transaction started for the trigger event. The normal restrictions on the use of certain procedural SQL statements within a transaction apply.

In addition, because the trigger action must be atomic, a COMMIT or ROLLBACK statement cannot be executed within it.

The creator of the trigger must hold the appropriate access rights, **with grant option**, for all the operations performed within the trigger action. This is checked when the CREATE TRIGGER statement is executed.

If the trigger time specified for the trigger is BEFORE, the following restrictions apply to the trigger action:

- the trigger action must not contain any SQL statement that performs an update (i.e. DELETE, INSERT and UPDATE statements are not permitted)
- a routine whose access clause is MODIFIES SQL DATA must not be invoked from within the trigger action.

If an exception is raised from the trigger action, it can be handled within the trigger by declaring a handler in the normal way for a compound SQL statement (see Section 8.7.2).

If there is no handler declared in the trigger action to handle the exception, it will propagate to the environment executing the trigger event and will be dealt with appropriately there. The default behavior at that level will be to undo the effect of the trigger event and all the operations performed in the trigger action.

It is possible to explicitly raise an exception from within the trigger action, or from within an exception handler declared in it, by executing the SIGNAL statement.

9.4.1 New and old table aliases

It is possible, by using the optional REFERENCING clause in the CREATE TRIGGER statement, to specify the existence of two special tables called the "old table alias" and the "new table alias".

The name of each table is specified separately in the REFERENCING clause and they must not have the same name. Both tables are accessible from within the trigger action.

Each specified table is created just prior to the execution of the trigger and each is local to the entire scope of the trigger action, including the optional WHEN clause.

Note: A REFERENCING clause is not permitted in a CREATE TRIGGER statement which has BEFORE specified for the trigger time.

The old table alias and new table alias contain copies of those database table rows affected by the data manipulation operation that caused the execution of the trigger (i.e. the trigger event).

In the case of a trigger executing AFTER the trigger event, the alias tables show the database table rows **actually** affected by the trigger event.

In the case of a trigger executing INSTEAD OF the trigger event, the alias tables show the database table rows that **would have been** affected by the trigger event had its data manipulations actually been performed.

The old table alias shows each affected database table row in the state it was in **before** normal execution of the trigger event.

The new table alias shows how each affected table row appears (or would appear, in the case of INSTEAD OF triggers) **after** normal execution of the trigger event.

If the trigger event was an INSERT, the **old** table alias cannot be specified. If the trigger event was a DELETE, the **new** table alias cannot be specified.

The rows of the alias tables are created before the trigger executes and do not change during execution of the trigger.

If the trigger action contains data manipulation statements that attempt to alter the alias tables, an error will be raised when the CREATE TRIGGER statement is executed.

9.4.2 Altered table rows

When the rows of the database table on which the trigger was created are examined from within the trigger action, they will always reflect the **actual** data manipulations performed by the trigger event and the trigger action.

In the case of an AFTER trigger, all rows inserted by the trigger event will be visible, all rows deleted by the trigger event will not be found and all rows updated by the trigger event will appear in their altered state.

In the case of an INSTEAD OF trigger, **none** of the data manipulations specified by the trigger event will seen when the table is examined because the trigger event does not actually perform any of its data change operations.

The rows of the old table alias and the new table alias will **always** show the changes that were specified by the trigger event, even if these changes were not actually performed on the database table (as is the case for INSTEAD OF triggers).

9.4.3 Recursion

Any data manipulation statements occurring in a trigger action will be executed in the normal way. It is, therefore, possible that the execution of a data manipulation statement in the trigger action may lead to the execution of another trigger or the recursive execution of the current trigger.

In either case, the execution context of the current trigger action is preserved and newly-invoked trigger executes in the normal way, in its own execution context, with appropriate versions of any old table alias and new table alias (see Section 9.4.1).

9.5 Comments on triggers

The COMMENT ON TRIGGER statement can be used to create a comment on a trigger.

Only the creator of the schema to which the trigger belongs may create a comment on the trigger.

9.6 Using DROP and REVOKE

The following points apply to triggers when using DROP and REVOKE with triggers:

- A trigger can be dropped by using the DROP TRIGGER statement.
- Only the creator of the trigger can drop it using the DROP TRIGGER statement.
- When a trigger is dropped, the comments created on it are also dropped.
- Dropping an object referenced from an SQL statement in a trigger action will cause the trigger to be dropped.
- If the required privileges held on a database object are revoked from the creator of a trigger whose trigger action contains an SQL statement referencing the object, the trigger will be dropped.

10 HANDLING ERRORS AND EXCEPTION CONDITIONS

Errors may arise at three general levels in an embedded SQL program (not counting errors in the SQL-independent host language code). These are syntax, semantic and run-time errors.

See Section 8.7 for information about managing exception conditions in routines and triggers.

10.1 Syntax errors

Syntax errors are constructions which break the rules for formulating SQL statements. For example:

- spelling errors in keywords

```
SLEECT                    for    SELECT
```

- incorrect or missing delimiters

```
DELETEFROM                for    DELETE FROM
SELECT column1;column2    for    SELECT column1,column2
```

- incorrect clause ordering

```
UPDATE..WHERE..SET        for    UPDATE..SET..WHERE
```

Syntactically incorrect statements are not accepted by the preprocessor. The error must be corrected before the program can be successfully preprocessed.

10.2 Semantic errors

Semantic errors arise when SQL statements are formulated in full accordance with the syntax rules, but do not reflect the programmer's intentions correctly. Some semantic errors, e.g. incorrect references to database objects, are detected and reported by the ESQL preprocessor but other semantic errors will not become apparent until run-time.

10.3 Run-time errors

Run-time errors and exception conditions (for example warnings) arising during execution of embedded SQL statements are signaled by the contents of the SQLSTATE status variable described in Section 3.2. A list of possible SQLSTATE values is provided in Appendix B.

The GET DIAGNOSTICS statement can be used to retrieve detailed information about an exception (see the *Mimer SQL Reference Manual* for the syntax description).

The NATIVE_ERROR and MESSAGE_TEXT fields of the diagnostics area retrieved by using GET DIAGNOSTICS are used to get the internal Mimer SQL return code and the descriptive text, respectively, relating to the exception (these are listed in Appendix B).

10.3.1 Testing for run-time errors and exception conditions

The application program may test the outcome of a statement in one of two ways:

- by explicitly testing the content of the SQLSTATE variable
- by using the SQL statement WHENEVER (see the *Mimer SQL Reference Manual* for the syntax description), which tests the class of the SQLSTATE variable.

An application program may contain any number of WHENEVER statements, and the statements may be placed anywhere in the program. A separate WHENEVER statement must be issued for each situation (NOT FOUND, SQLERROR or SQLWARNING) which is to be tested. When an exception condition arises, action will be taken as specified in the WHENEVER statement most recently encountered in the code, for the respective condition.

WHENEVER statements are expanded by the preprocessor into explicit tests. These tests are placed after **every subsequent SQL statement in that program** until a new WHENEVER statement is issued for the same condition.

Two important consequences follow:

- WHENEVER statements are preprocessed strictly in the order in which they appear in the source code, regardless of execution order or conditional execution that the source code might imply. For instance, the WHENEVER statement in the following FORTRAN construction is expanded by the preprocessor, even though its “execution” is never actually requested:

```
      ...
       GOTO 1025
       EXEC SQL WHENEVER SQLERROR GOTO 1600
1025   CONTINUE
       EXEC SQL DELETE FROM MYTABLE
      ...
```

- Mixing explicit tests and WHENEVER statements requires care. As a general rule, it is advisable to use *either* hand-written tests *or* WHENEVER statements in a program module, and to avoid mixing them.

  The condition handling defined by a WHENEVER statement applies to the SQL statements that follow it in the source code. If a "GOTO" action is defined, the pre-processor inserts an exception test and action directly after each SQL statement affected by it and thus *before* any hand-written tests in the source code. The hand-written test in this situation would never be executed.

  If "CONTINUE" is specified in a WHENEVER statement, the pre-processor does not insert an exception test and action, thus no exception handling is defined by the WHENEVER statement. Any hand-written tests present in the source code *will* then take effect.

  The interchange between hand-written exception handling and the implicit exception handling inserted by the pre-processor (or not) can be confusing. It is therefore advisable to make a clear coding decision to use one method or the other.

A HOST LANGUAGE DEPENDENT ASPECTS

SQL statements may be embedded in any of the following host languages:

- C/C++
- COBOL
- FORTRAN

See Section 2.3.2.1 for list of the ESQL preprocessors available for a specific platform.

This appendix describes features of embedded SQL which differ between the respective host languages.

Note: It is not a complete description of the rules for writing embedded SQL programs. The programmer should use the main body of this manual as a guide to writing programs, and refer to this appendix for language-specific details.

The following topics are discussed for each language:

- SQL statement format: delimiters, margins, line continuation, comments, special characters.
- Restrictions.
- Host variables - declarations, SQL data type correspondence, value assignment rules.
- Preprocessor output format.
- Scope rules.

A.1 Embedded SQL in C/C++ programs

Mimer SQL supports embedded SQL for C/C++ following the ANSI standard.

A.1.1 SQL statement format

Statement delimiters

SQL statements are identified by the leading delimiter “exec sql” and terminated by a semicolon (;).

Example:

```
exec sql DELETE FROM HOTEL;
```

Line continuation

Line continuation rules for SQL statements are the same as those for ordinary C statements.

For a string constant, a white-space character (ASCII HEX-values 09 - 0D, or 20, i.e. <TAB>, <LF>, <VT>, <FF>, <CR> or <SP>), can be used to join two or more sub-strings. Each substring must be separately enclosed in delimiters.

Examples:

```
exec sql SELECT HOTELCODE, ROOMNO
            FROM   BOOK_GUEST
            WHERE  RESERVATION = :CUSTNO;

exec sql COMMENT ON TABLE ROOM_PRICES IS
            'Prices apply from date given'<LF>
            ' in column FROM_DATE';
```

Comments

Comments, from // to end-of-line or enclosed between the markers /* and */, may be written anywhere within SQL statements where a white-space is permitted, except between the keywords “exec” and “sql” and within string constants. The comment may replace the white-space, for example

```
exec sql DELETE/* all rows */FROM HOTEL;
```

Special characters

The delimiters in SQL are apostrophes (') for string constants and quotation marks (") for delimited identifiers. This is contrary to the C string delimiter usage.

Keywords are separated by a white-space character.

A.1.2 Host variables

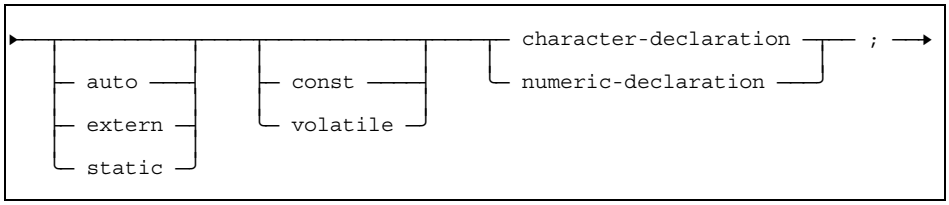
Declarations

Host variables used in SQL statements must be declared within the SQL DECLARE SECTION, delimited by the statements BEGIN DECLARE SECTION and END DECLARE SECTION.

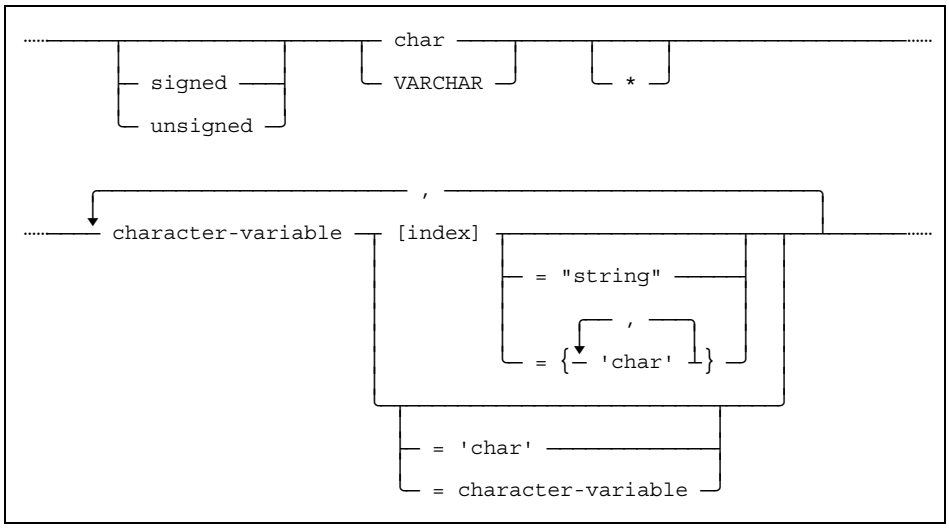
Variables declared within the SQL DECLARE SECTION must conform to the following rules in order to be recognized by the SQL preprocessor:

- host variables may be of AUTO, EXTERN or STATIC class, or parameters
- array variables are not permitted with the exception of character arrays
- character arrays are interpreted as null terminated strings
- the VARCHAR host variable data type is recognized by the ESQL/C preprocessor and should be used when variable-length character data is to be returned from SQL as a null terminated string without any blank padding (the VARCHAR host variable should be declared with a length one greater than the length of the SQL VARCHAR)
- where binary data is stored in a character array, the size of the array must match the length of the binary data exactly because binary data is not terminated and therefore all array elements are significant
- variable names are case significant
- indicator variables should be declared as short or int
- SQLSTATE should be declared as char[6].

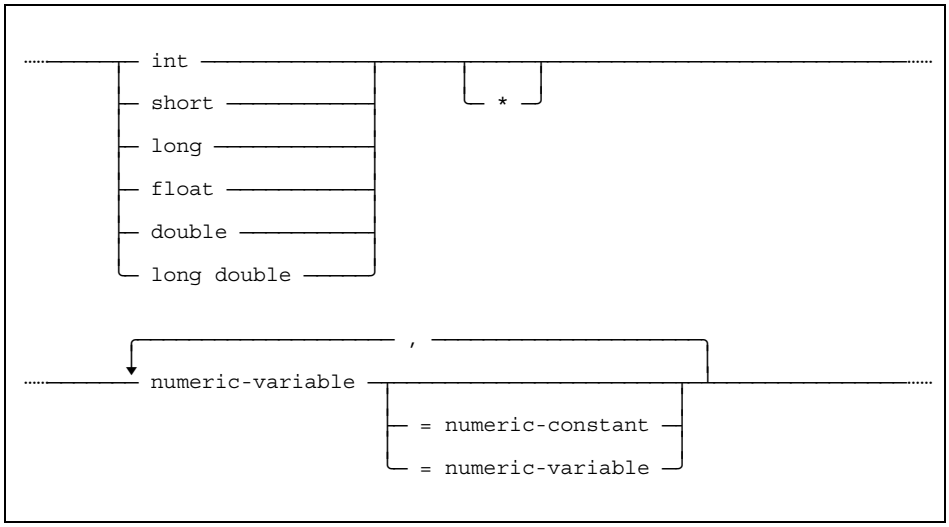
A syntax diagram showing the variable declarations recognized by the ESQL/C preprocessor is given below:



where *character-declaration* is



and *numeric-declaration* is



The following points should be noted:

- In accordance with the syntax rules of C, keywords are case-sensitive and are given in the required case in the syntax diagram. This deviates from the general practice in Mimer SQL documentation of using upper-case to denote keywords
- *Index* must be a number which is 1 or greater
- Only data types char and VARCHAR can be indexed.

SQL data type correspondence

Valid host data types are listed below for each of the data types used in SQL statements.

SQL data type	C variable declaration
SMALLINT INTEGER BIGINT	short int long
DECIMAL NUMERIC	float double long double
FLOAT REAL DOUBLE PRECISION	float double long double
CHARACTER VARCHAR DATETIME INTERVAL BINARY BINARY VARYING	char VARCHAR [1]

[1] The VARCHAR host variable type is recognized by the ESQL/C preprocessor and converted to the char data type in C.

Note: Your C compiler may not support all of these possible declarations.

Value assignments

The general rules for conversion of values between compatible but different data types (see Chapter 4 of the Mimer SQL Reference Manual) apply to the transfer of data between the database and host variables, with the data type correspondence as given in the table above.

When reading any character array host variable, declared as char or VARCHAR, the contents of the variable must be terminated by a null byte. When a host variable declared as char is read, its value is blank padded to the same length as the host variable. When a host variable declared as VARCHAR is read, no blank padding is performed.

When retrieving a value shorter than the character array host variable (declared as char), the host variable will be padded with blanks (and terminated with a null byte). When retrieving a value to a VARCHAR host variable, the variable will not be blank padded, just terminated by a null byte.

When retrieving binary data into a character array there is no padding or termination of the binary string, so all the character array elements have significance. The character array must, therefore, be declared with exactly the same length as the binary data.

When any type conversion is done when retrieving a value to a character host variable, the data will be right justified. When type conversion is done when retrieving a value to a VARCHAR host variable, the data will be left justified.

Example:

```
char cstr[9];
VARCHAR vstr[9];
retrieving the value 'abc ' will give the following
result:
cstr = 'abc     ' /* blankpadded to eight characters */
vstr = 'abc '     /* the same length as the value */

retrieving the value 123, the values will be as:
cstr = '     123' /* right justified */
vstr = '123'      /* left justified */
```

See Chapter 4 of the Mimer SQL Reference Manual for a further discussion of different character string assignments.

A.1.3 Preprocessor output format

Output from the ESQL/C preprocessor retains SQL statements from the original source code as comments. Comments on the same line as SQL statements are retained as “comments within comments”, marked by the delimiters /+ and +/.

The preprocessed code is structured to reflect the structuring of the original source code.

The use of the #line directive will ensure that any information from the C compiler will correctly reference line numbers in the original source code. It will also help a debugger correctly coordinate display of source lines in the original source file and the generated C file. Refer to information on running ESQL for the platform you are using for details on how to get #line directives.

A.1.4 Scope rules

Host variables follow the same scope rules as ordinary variables in C. SQL descriptor names, cursor names and statement names must be unique within the compilation unit. A compilation unit for C is the same as a file (including included files).

A.2 Embedded SQL in COBOL programs

Mimer SQL supports embedded SQL for COBOL following the COBOL-85 ANSI standard.

A.2.1 SQL statement format

Statement delimiters

SQL statements are identified by the leading delimiter EXEC SQL and terminated by END-EXEC.

SQL statements are treated exactly as ordinary COBOL statements with regard to the use of an ending period to mark the end of a COBOL sentence. Any valid COBOL punctuation may be placed after the END-EXEC terminator.

Examples:

```
EXEC SQL DELETE FROM HOTEL END-EXEC.
IF SQLSTATE NOT = "02000" THEN
            EXEC SQL COMMIT END-EXEC
ELSE
            EXEC SQL ROLLBACK END-EXEC.
```

Margins

Statements (including delimiters) may be written anywhere between positions 8 and 72 inclusive.

Line continuation

Line continuation rules for SQL statements are the same as those for ordinary COBOL statements.

If a string constant within an SQL statement is divided over several lines, the first non-blank character on the continuation line must be a string delimiter. There is no terminating string delimiter at the end of the line preceding the continuation line.

Example:

```
EXEC SQL SELECT HOTELCODE, ROOMNO
            FROM   BOOK_GUEST
            WHERE  RESERVATION = :CUST-NUMBER END-EXEC.
EXEC SQL COMMENT ON TABLE ROOM_PRICES IS
                      'Prices apply from date given
-                     ' in column FROM_DATE' END-EXEC.
```

An alternative way to break a character string constant over several lines, is to use a white-space character (ASCII HEX-values 09 - 0D, or 20, i.e. <TAB>, <LF>, <VT>, <FF>, <CR> or <SP>), to join two or more substrings. Each substring must be separately enclosed in delimiters.

```
EXEC SQL COMMENT ON TABLE ROOM_PRICES IS
                      'Prices apply from date given'<LF>
                      ' in column FROM_DATE' END-EXEC.
```

Comments

Comment lines, marked by an asterisk (*) in position 7, may be written within SQL statements. The whole line following a comment mark is treated as a comment.

Debugging lines and page eject lines (marked by D and / respectively in position 7) are treated as comments by the preprocessor.

Special characters

The delimiters in SQL are apostrophes (') for string constants and quotation marks (") for delimited identifiers. This is contrary to the default COBOL string delimiter usage.

Observe that the minus sign (-) is valid in variable names in COBOL. All arithmetic expressions using this operator should have at least one space separating the operands from the operator. Thus

:A - B means “variable called A minus column B”
:A-B means “variable called A-B”

A.2.2 Restrictions

The following restrictions apply specifically to COBOL:

- END-EXEC is a keyword reserved to SQL.
- COBOL figurative constants (such as ZERO and SPACE) may not be used as constants in SQL statements.

A.2.3 Host variables

Declarations

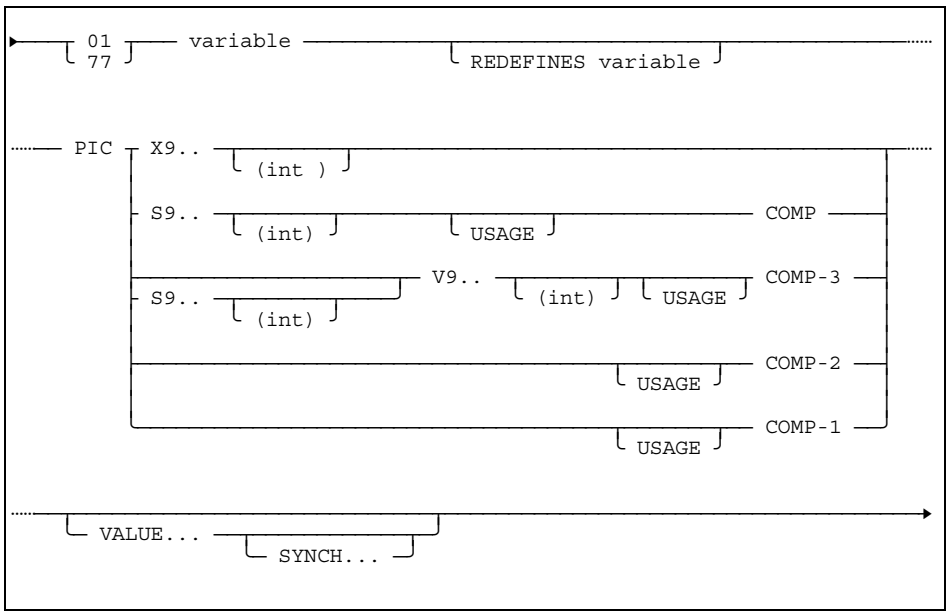
Host variables used in SQL statements must be declared within the SQL DECLARE SECTION, delimited by the statements BEGIN DECLARE SECTION and END DECLARE SECTION.

Variables declared within the SQL DECLARE SECTION must conform to the following rules in order to be recognized by the SQL preprocessor:

- variable names must begin with a letter. Within this restriction, any valid COBOL variable name may be used
- host variable structures may not be used
- the specifications JUSTIFIED, BLANK WITH ZERO, and OCCURS may not be used
- the data type must be consistent with SQL data types as specified below
- level number 01 or 77 should be used for all variable names which are used in SQL statements. Other levels may be used for program host variables, but they are not recognized by the preprocessor

- FILLER entries are ignored for variables used in SQL statements
- Indicator variables should be declared as PIC S9(4) COMP or PIC S9(9) COMP

A syntax diagram for COBOL variable declarations recognized by the ESQL/COBOL preprocessor is given below. Other declarations are ignored by the preprocessor:



Commas and semicolons may be used in accordance with standard COBOL practice.

The following abbreviations are accepted:

PIC	for	PICTURE or PICTURE IS
USAGE	for	USAGE or USAGE IS
COMP	for	COMPUTATIONAL
SYNC	for	SYNCHRONIZED

Note: The PIC S9(n)9(m) formulation is not accepted.

SQL data type correspondence

Valid host data types are listed below for each of the data types used in SQL statements.

Varying-length character string structures may be used in embedded SQL statements in COBOL programs. In assigning the value of such variables to columns, the current length of the string is used. The variable name used in SQL statements is the name of the structure (level 01 declaration), not of the character string element (level 49).

SQL data type	COBOL data declaration	Comments
SMALLINT INTEGER	01 name PIC S9(n) COMP.	1 ≤ n ≤ 9
DECIMAL NUMERIC	01 name PIC S9(n)V9(m) COMP-3.	1 ≤ n+m ≤ 15
FLOAT DOUBLE	01 name COMP-2.	
REAL	01 name COMP-1.	
CHARACTER VARCHAR DATETIME INTERVAL	01 name PIC X(n).	1 ≤ n

Value assignments

The general rules for conversion of values between compatible but different data types (see Chapter 4 of the Mimer SQL Reference Manual) apply to the transfer of data between the database and host variables, with the data type correspondence as given in the table above.

The first element in a varying-length character string structure is used to store the current length of the character string. When writing to the variable, the first element is updated with the current length of the variable. If the column value is longer than the variable, the value is truncated.

A.2.4 Preprocessor output format

Output from the ESQL/COBOL preprocessor retains SQL statements from the original source code as comments. Comments within SQL statements are retained exactly as written. The output follows the ANSI standard for record format, and should be compiled with a COBOL compiler set to accept ANSI standard.

Debugging lines and page eject lines (using D and / respectively in position 7) remain unchanged after preprocessing.

The preprocessed code is structured to reflect the structuring of the original source code.

A.2.5 Scope rules

Host variables follows the same scope rules as ordinary variables in COBOL. SQL descriptor names, cursor names and statement names must be unique within the compilation unit. A compilation unit for COBOL is the same as a routine.

A.3 Embedded SQL in FORTRAN programs

Mimer SQL supports embedded SQL for ANSI FORTRAN-90 fixed format.

Source statements must be provided as fixed format, 80 byte records.

A.3.1 SQL statement format

Statement delimiters

SQL statements are identified by the leading delimiter EXEC SQL. The end of an SQL statement is marked by the end of the line when the following line does not begin with a continuation character. The FORTRAN-90 statement delimiter ";" can also be used.

Example:

```
    EXEC SQL DELETE FROM HOTEL
```

Margins

Statements (including delimiters) may be written anywhere between positions 7 and 72 inclusive.

Line continuation

Line continuation rules for SQL statements are the same as those for ordinary FORTRAN statements. The continuation character is any character except space and 0 (zero) in position 6. The FORTRAN limitation of a maximum of 19 continuation lines per statement does not apply within SQL statements.

For a string constant, a white-space character (ASCII HEX-values 09 - 0D, or 20, i.e. <TAB>, <LF>, <VT>, <FF>, <CR> or <SP>), can be used to join two or more substrings. Each substring must be separately enclosed in delimiters.

Examples:

```
    EXEC SQL SELECT HOTELCODE, ROOMNO
+            FROM   BOOK_GUEST
+            WHERE  RESERVATION = :CUSTNO

    EXEC SQL COMMENT ON TABLE ROOM_PRICES IS
+                     'Prices apply from date given'<LF>
+                    ' in column FROM_DATE'
```

Statement numbers

Any labeled SQL statement in the source code will generate a CONTINUE statement during preprocessing. Declarative SQL statements used before the first executable SQL statement should not be labeled.

Comments

Comment lines, marked by * or C in position 1, may be written within SQL statements. The whole line following a comment mark is treated as a comment, and the following line must either be another comment or follow the continuation rules given above.

Note: Lines which are completely blank are not treated as comments by the ESQL/FORTRAN preprocessor. The absence of a continuation character indicates the end of the previous statement, and a completely blank line may be used to structure comments in the output from the preprocessor. See Section A.3.3 for details.

FORTRAN-90 style comments may also be used, marked by the "!" character (the text between the "!" and the end-of-line is treated as a comment).

Debugging lines (marked with a D in position 1) are treated as comments by the preprocessor.

A.3.2 Host variables

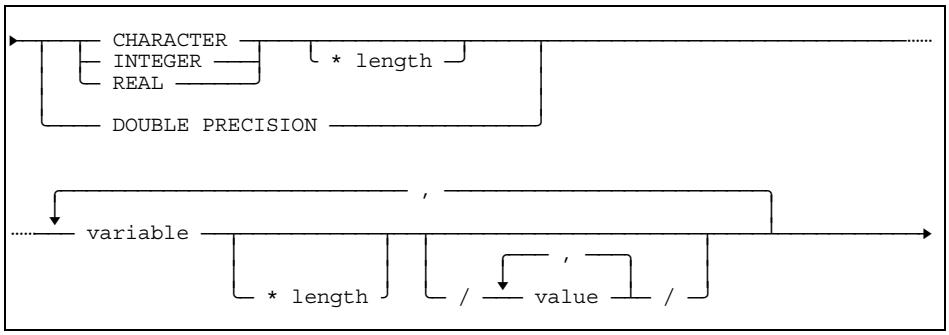
Declarations

Host variables used in SQL statements must be declared within the SQL DECLARE SECTION, delimited by the statements BEGIN DECLARE SECTION and END DECLARE SECTION.

Variables declared within the SQL DECLARE SECTION must conform to the following rules in order to be recognized by the SQL preprocessor:

- any valid FORTRAN variable name may be used.
- variables must be scalar variables (i.e. they may not be elements of vectors or arrays).
- implicit declaration by means of the IMPLICIT statement or default typing may not be used.
- FORTRAN COMPLEX variables may not be used.
- character variables must be declared with a fixed constant length. Expressions and variable length declarations (such as CHARACTER*(*)) may not be used.
- indicator variables should be declared as INTEGER*2 or INTEGER*4.

A syntax diagram showing the variable declarations recognized by the ESQL/FORTRAN preprocessor is given below:



The data type declaration must be separated from the variable name by at least one space (which is not required in FORTRAN declarations outside the SQL DECLARE SECTION).

Thus the declaration:

```
INTEGER*2A
```

is not recognized. The required formulation is:

```
INTEGER*2 A
```

Lists of variables following a single default data type declaration are accepted. Any declarations in a list which are not valid in SQL contexts are ignored by the preprocessor. Thus the following statement declares variables A and D as INTEGER*4 and B as INTEGER*2 for use in SQL statements, while the array C is ignored:

```
INTEGER*4    A, B*2, C(10), D
```

SQL data type correspondence

Valid host data types are listed below for each of the data types used in SQL statements.

SQL data type	FORTRAN data declaration
SMALLINT INTEGER	INTEGER*2 INTEGER*4
DECIMAL NUMERIC	REAL*4
FLOAT DOUBLE REAL	REAL*8 DOUBLE PRECISION
CHARACTER VARCHAR DATETIME INTERVAL	CHARACTER*n

The following additional points should be noted:

- FORTRAN does not support DECIMAL data types. A string of digits including a decimal point is interpreted as a REAL constant in FORTRAN. Exponential notation should always be used to specify floating point values in SQL statements.
- DOUBLE PRECISION constants may be written with a D as the exponent marker in FORTRAN (e.g. 1.23D+02). The only permissible exponent marker within SQL statements is E (e.g. 1.23E+02).

Value assignments

The general rules for conversion of values between compatible but different data types (see Chapter 4 of the Mimer SQL Reference Manual) apply to the transfer of data between the database and host variables, with the data type correspondence as given in the table above.

A.3.3 Preprocessor output format

Output from the ESQL/FORTRAN preprocessor retains SQL statements from the original source code as comments. The output follows the ANSI standard for record format, and should be compiled with a FORTRAN compiler set to accept ANSI standard. Comments within SQL statements are retained exactly as written.

Completely blank lines between SQL statements and following comments cause the preprocessor to write the comments after the generated SQL call. Otherwise comments immediately following SQL statements are output before the generated call. Debugging lines (using D in position 1) remain unchanged after preprocessing.

The preprocessed code is structured to reflect the structuring of the original source code.

A.3.4 Scope rules

Host variables follows the same scope rules as ordinary variables in FORTRAN. SQL descriptor names, cursor names and statement names must be unique within the compilation unit. A compilation unit for FORTRAN is the same as a routine.

B RETURN CODES

Mimer SQL returns two kinds of return codes to an application. The SQLSTATE variable returns a standardized, general error code, which gives a rough description of the status for the most recently executed SQL statement.

GET DIAGNOSTICS can be called to access the exception information stored in the diagnostics area that applies to the most recently executed SQL statement (see Section 10.3).

The symbol <%> in the text of error messages listed in this chapter indicates the location of an identifier inserted at run-time

B.1 SQLSTATE return codes

SQLSTATE contains a 5-character long return code string that indicates the status of an SQL statement. These return codes are standardized following the established standards. Observe that not all standardized SQLSTATE return codes are used by Mimer SQL.

The SQLSTATE values consists of two fields. The class, which is the first two characters of the string, and the subclass, which is the terminating three characters of the string.

List of SQLSTATE values:

Class	Subclass	Meaning
00	000	**Success**
01	000	**Success with warning**
01	002	- Disconnect error
01	003	- Null value eliminated in set function
01	004	- String data, right truncation
01	005	- Insufficient item descriptor areas
01	006	- Privilege not revoked
01	007	- Privilege not granted
02	000	**No data**
07	000	**Dynamic SQL error**
07	001	- *using-clause* does not match dynamic variables
07	002	- *using-clause* does not match target specification
07	003	- Cursor specification cannot be executed
07	004	- *using-clause* required for dynamic parameters
07	006	- Restricted data type attribute violation
07	007	- *using-clause* required for result fields
07	008	- Invalid descriptor count
07	009	- Invalid descriptor index
08	000	**Connection exception**
08	001	- Client unable to establish connection
08	002	- Connection name in use
08	003	- Connection does not exist
08	004	- Server rejected the connection
08	006	- Connection failure
0A	000	**Feature not supported**
20	000	**Case not found for a case statement**
21	000	**Cardinality violation**

Class	Subclass	Meaning
22	000	**Data exception**
22	001	- String data, right truncation
22	002	- Null value, no indicator variable
22	003	- Numeric value out of range
22	005	- Error in assignment
22	006	- Invalid interval format
22	007	- Invalid datetime format
22	008	- Datetime field overflow
22	011	- Substring error
22	012	- Division by zero
22	015	- Interval field overflow
22	018	- Invalid character value for CAST
22	019	- Invalid escape character
22	023	- Invalid parameter value
22	024	- Unterminated C string
22	025	- Invalid escape sequence
22	027	- Trim error
23	000	**Integrity constraint violation**
24	000	**Invalid cursor state**
25	000	**Invalid transaction state**
26	000	**Invalid SQL statement identifier**
28	000	**Invalid authorization specification**
2E	000	**Invalid connection name**
33	000	**Invalid SQL descriptor name**
34	000	**Invalid cursor name**
35	000	**Invalid exception number**
37	000	**Syntax error or access violation** (in PREPARE or EXECUTE IMMEDIATE)
3C	000	**Ambiguous cursor name**
40	000	**Transaction rollback**
40	001	- Serialization failure
40	003	- Statement completion unknown
42	000	**Syntax error or access violation**
44	000	**WITH CHECK OPTION violation**
S1	000	**General error**
S1	001	- Memory allocation failure

B.2 Internal Mimer SQL return codes

Here the internal Mimer SQL return code values are listed together with the associated text message. See Section 10.3 for details on how to retrieve this information after an exception has been raised. The codes are grouped according to function as follows:

Code numbers	Functional group
> 0	Warnings, messages
= 0	Success
100	No data
-100 to -999	ODBC error codes
-10000 to -10999	Data-dependent errors
-11000 to -11999	Limits exceeded
-12000 to -12999	SQL statement errors
-13000 to -13999	Not currently used
-14000 to -14999	Program-dependent errors
-15000 to -15999	Not currently used
-16000 to -16999	Databank and table errors
-17000 to -17999	Not currently used
-18000 to -18999	Miscellaneous errors
-19000 to -19999	Internal errors
-20000 to -20999	Not currently used
-21000 to -21999	Communication errors
-22000 to -22999	JDBC error codes

Corrective action is given in general terms for each group of codes. When reporting errors to Mimer support, make sure you include the internal Mimer SQL return code.

B.2.1 Warnings and messages

No corrective action is normally required for internal Mimer SQL return code values greater than zero.

54 Character string was truncated
55 Insufficient item descriptor areas
56 Privilege not revoked
57 Privilege not granted
58 Zero bits were added to the binary string
90 Login failure
91 Soft enter performed
92 No cursor state was saved on stack
94 Message text not found
100 Row not found for FETCH, UPDATE or DELETE, or the result of a query is an empty table
100 No data - Item number is greater than the value of count

B.2.2 ODBC errors

These errors arise when ODBC calls to Mimer SQL fails for some reason.

-100 Illegal sequence
-101 Out of memory
-102 Option out of range
-103 Function not supported
-104 Connection not open
-105 Connection in use
-106 Invalid argument value
-107 Invalid transaction operation code
-108 Internal network buffer overflow
-109 Invalid C data type
-110 Invalid SQL data type
-111 Bad address
-112 Function already active
-113 Operation canceled
-114 Wrong number of parameters
-115 Use ODBC function SQLTransact to commit or rollback transaction
-116 Statement is not in a prepared state
-117 Invalid transaction state
-118 Unknown statement type
-119 Unknown internal data type
-120 Extype corrupt
-121 Invalid buffer length
-122 String data truncated
-123 Numeric data truncated
-124 Numeric value out of range
-125 Invalid numeric value
-126 Bad parameter passed to numeric package
-127 Invalid column number
-128 Database name mandatory
-129 Connect dialog failed
-130 Data truncated
-131 Invalid connection string attribute
-132 Invalid cursor state
-133 Invalid parameter number
-134 Descriptor type out of range
-135 Invalid type passed to DICOA3
-136 Function type out of range
-137 Invalid cursor name
-138 Duplicate cursor name
-139 Cursor hash table corrupt
-140 ODBC database control block chain corrupt
-141 Option type out of range
-142 Option value not supported

-143	Option not supported
-144	Invalid row or keyset size
-145	Invalid concurrency option
-146	Invalid fetch type
-147	Not a scrollable cursor
-148	Row position out of range
-149	Only one SQLPutData for fixed length parameter
-150	SQLPutData does not support block cursors
-151	Driver not capable
-152	Table type out of range
-153	Invalid string length
-154	Data type out of range
-155	Syntax error found in escape clause
-156	DDO buffer overflow
-157	Uniqueness option type out of range
-158	Accuracy option type out of range
-159	Column type out of range
-160	Scope type out of range
-161	Nullable type out of range
-162	Internal type mismatch
-163	Conversion between data types not supported
-164	Invalid date, time, or timestamp
-165	Restricted data type attribute violation
-166	Date, time, or timestamp data truncated
-167	Database has not been configured. Run Configure MIMER 7.1
-168	Translated native SQL string was truncated
-169	ODBC extension DATE, TIME or TIMESTAMP is not supported
-170	ODBC extension OUTER JOIN is not supported
-171	ODBC extension for procedure invocation is not supported
-172	Unrecognized first word in escape clause, expected 'CALL','FN','OJ','D','T' or 'TS'
-173	This server version does not support the used scalar function
-174	Unrecognized scalar function found in escape clause
-175	Argument missing in scalar function
-176	Too many arguments in scalar function
-177	Syntax error, incomplete escape clause
-178	Syntax error, unmatched apostrophe in string literal
-179	Syntax error, unmatched quote in delimited identifier
-180	Invalid data type specified in scalar function CONVERT
-181	Information type out of range
-182	Parameter type may only be used with procedures
-183	Parameter type out of range
-184	Update and delete where current fully supported (not simulated)
-185	Option value changed
-186	Static scrollable cursor used instead of keyset or dynamic cursor
-187	Error in row, please check next error code

-188 Cancel treated as FreeStatement/CLOSE
-189 Attempt to fetch before the result set returned the first rowset
-190 Invalid cursor position
-191 Unknown first parameter in scalar function TIMESTAMPADD
-192 Unknown first parameter in scalar function TIMESTAMPDIFF
-193 Bad parameter passed to datetime package
-203 Interval second fraction truncated
-204 Interval truncation error
-205 Interval convert error
-206 Year to month interval cannot be converted to a numeric value because it is not a single field
-207 Interval cannot be converted to a numeric value because it is not a single field
-208 Invalid interval literal
-209 Interval leading field truncation error
-210 Interval trailing field truncated
-211 Binary data truncated
-212 Binary truncation error
-213 Binary length invalid
-214 Binary data invalid
-215 Binary not supported
-216 Inconsistent descriptor information
-217 Cannot modify IRD
-218 Invalid use of null pointer
-219 Character data not hexadecimal
-220 Internal error
-221 Significant parts of datetime/interval string truncated
-224 Binary data truncated
-225 String data truncated
-226 Binary data truncated

B.2.3 Data-dependent errors

These errors arise when an SQL statement cannot be executed correctly because of the data content of variables, expressions, and so on in the statement. The appropriate corrective action is determined by the nature of the error and the specific context in the application program.

-10001 Transaction aborted due to conflict with other transaction
-10100 PRIMARY KEY constraint violation, attempt to insert NULL value
-10101 PRIMARY KEY constraint violation, attempt to insert duplicate value
-10102 Domain constraint violation <%>
-10103 Table constraint violation <%>
-10104 View constraint violation <%>
-10105 INSERT or UPDATE operation invalid because the referencing table <%> does not satisfy a referential constraint

-10106	UPDATE or DELETE operation invalid because the referenced table <%> does not satisfy a referential constraint
-10107	The result of a subquery in a basic predicate is more than one value
-10108	Result of SELECT INTO or EXECUTE INTO statement is a table of more than one row
-10109	Type constraint violation
-10110	UNIQUE constraint violation
-10199	Host variable type packed decimal is not supported
-10200	Reserved numeric operand found during data type conversion
-10201	Length error or incorrect value found during data type conversion
-10202	Division by zero attempted
-10203	Negative overflow occurred during data type conversion
-10204	Positive overflow occurred during data type conversion
-10205	Loss of significance occurred during data type conversion
-10207	Undefined value found during data type conversion
-10208	Restricted data type attribute violation
-10210	Error in assignment
-10211	Undefined value found during data type conversion
-10212	Overflow occurred during data type conversion
-10221	The NULL value cannot be assigned to a host variable because no indicator variable is specified
-10222	NULL not allowed for item descriptor area
-10250	Data type not supported
-10301	Loss of significance occurred in arithmetic operation <%>
-10302	Positive overflow occurred in arithmetic operation <%>
-10303	Negative overflow occurred in arithmetic operation <%>
-10304	Division by zero attempted
-10305	Bad parameter encountered in arithmetic operation <%>
-10306	Invalid input for numeric function
-10307	The binary strings are of unequal length
-10310	Invalid character value for CAST
-10311	String data truncated
-10312	Numeric value out of range
-10313	Illegal (negative) substring length
-10314	Like pattern escape character not followed by underscore or percent character
-10315	Length of like pattern escape character is not equal to 1
-10321	Datetime loss of significance
-10322	Datetime positive overflow
-10323	Datetime negative overflow
-10325	Bad parameter encountered in datetime operation
-10326	Datetime illegal operand
-10327	Invalid datetime value
-10328	Datetime subtype mismatch
-10329	Invalid datetime format
-10331	Interval loss of significance

-10332 Interval positive overflow
-10333 Interval negative overflow
-10335 Bad parameter encountered in interval operation
-10336 Interval illegal operand
-10337 Invalid interval value
-10338 Interval subtype mismatch
-10339 Invalid interval format
-10601 Invalid value for field of item descriptor area
-10602 Invalid datetime or interval code

B.2.4 Limits exceeded

These errors arise when internal limits in the Mimer SQL system are exceeded. Some of the limitations are determined by installation-specific parameters, while others are fixed by Mimer SQL. In general, errors of this nature require either re-installation of the system with extended limitations or modification of the application program to reduce the system demands. Contact your Mimer representative if you have difficulty avoiding errors of this nature.

-11001 Dynamic storage area exhausted in host level interface (DYNDE3)
-11002 Internal DB dynamic storage area (SQLPOOL) exhausted
-11011 Internal storage (SQLPOOL) for like pattern exhausted
-11012 Transaction list exhausted
-11013 Too many databanks referenced in statement (max 30)
-11014 Too many databanks active in transaction
-11047 The maximum number of recursive invocations has been exceeded
-11100 Internal limit exceeded : query stack
-11101 Internal limit exceeded : scan stack
-11102 Internal limit exceeded : generation stack
-11103 Internal limit exceeded : table descriptor list
-11105 Internal limit exceeded : patch table
-11106 Internal limit exceeded : label table
-11107 Internal limit exceeded : traversal stack
-11108 Internal limit exceeded : sco list
-11109 Internal limit exceeded : boolean stack
-11111 Internal limit exceeded : semantic stack
-11113 Internal limit exceeded : statement too complex
-11114 Required temporary table row length is <%>, only <%> is possible
-11115 Internal limit exceeded : restriction group pool
-11118 Internal limit exceeded : scan queue

B.2.5 SQL statement errors

These errors arise from syntactic or semantic errors in SQL statements. In general, syntactic errors in embedded SQL programs are detected by the preprocessor, so errors cannot arise at run-time. Dynamically submitted SQL statements are however parsed at run-time, and the syntax error codes are returned after attempting PREPARE for a syntactically incorrect source statement.

Semantic errors can arise at run-time from both dynamic and static SQL statements.

-12001 Too many errors, error collection terminated
-12101 Syntax error, <%> assumed missing
-12102 Syntax error, <%> ignored
-12103 Syntax error, <%> assumed to mean <%>
-12104 Invalid construction
-12106 Internal parser error, analysis aborted
-12107 Syntax analysis resumed here
-12108 Multiple statements not allowed
-12120 Table name too long
-12121 String literal too long
-12122 Numeric literal too long
-12123 Invalid password string
-12124 Invalid hexadecimal literal
-12125 Reserved word may not be used as an identifier
-12126 Invalid name
-12128 Result of concatenation too long
-12129 Table definition does not include any column specification
-12131 Table definition includes more than one PRIMARY KEY specification
-12132 Only one column allowed in column list
-12152 <%> not allowed in EXECUTE mode
-12154 User name too long
-12156 Column name too long
-12157 Synonym name too long
-12158 Correlation name too long
-12159 Cursor name too long
-12160 Databank name too long
-12161 Shadow name too long
-12162 Host Variable or Parameter Marker name too long
-12163 File name too long
-12164 Label name too long
-12165 Index name too long
-12166 Object name too long
-12167 View name too long
-12168 Domain name too long
-12169 Too many identifier names given

-12174 Syntax error in escape clause, expecting comma before PRODUCT or CONFORMANCE specification
-12175 Syntax error in escape clause, invalid CONFORMANCE specification
-12176 Syntax error in escape clause, invalid YEAR specification
-12177 Syntax error in escape clause, invalid PRODUCT specification
-12178 Syntax error in escape clause, invalid VENDOR specification
-12179 Syntax error in escape clause, expecting VENDOR or YEAR after '--(*'
-12180 Syntax error, unexpected token '*)--'
-12181 Syntax error in escape clause, terminating '*)--' missing
-12182 Syntax error in escape clause
-12200 Table <%> not found, table does not exist or no access privilege
-12201 Table reference <%> is ambiguous
-12202 <%> is not a column of an inserted table, updated table or any table identified in a FROM clause
-12203 <%> is neither an object table of an INSERT, UPDATE or DELETE statement, nor specified in a FROM clause
-12204 Column reference <%> ambiguous
-12205 Column <%> not referenced in GROUP BY clause
-12207 DISTINCT specified more than once in a subselect
-12208 SELECT clause of a subquery specifies more than one column
-12209 Column <%> identified in HAVING clause but not included in GROUP BY clause
-12210 Operand of set function includes a set function
-12212 Operand of set function includes a correlated reference specified in an expression
-12213 Set function not specified in a SELECT clause or HAVING clause
-12214 Invalid operand type, expected type is <%>
-12215 Operand not of <%> type
-12216 Operands are not comparable
-12217 Set function containing DISTINCT may not be specified within an expression
-12220 Expression must be a column
-12221 SELECT clause contains both column expressions and set function expressions
-12223 ORDER BY clause invalid because it includes a column name that is not part of the result table
-12224 ORDER BY clause invalid because it includes an integer which does not identify a column of the result table
-12225 The ORDER BY clause is invalid because it includes an ambiguous column reference
-12226 Set function argument not bound in HAVING context
-12227 Duplicate column reference in FOR UPDATE OF clause
-12230 Invalid numeric literal
-12231 Update or insert value is NULL, but the object column <%> cannot contain NULL values

-12232 Insert value must be a constant expression or NULL
-12233 The number of insert values is not the same as the number of object columns
-12234 Update or insert value not compatible with the data type of the object column <%>
-12236 Column name <%> does not identify a unique column of the result table
-12238 Column <%> cannot be updated because it is derived from a set function or expression
-12239 The use of NULL in a SELECT clause is only allowed in a UNION
-12240 Statement contains too many table references
-12242 The corresponding columns of the operands of a UNION do not have compatible column descriptions
-12243 Result table contains a column for which the type cannot be determined
-12244 Operands of a UNION do not have the same number of columns
-12245 FOR UPDATE clause may not be specified because the result table cannot be modified
-12246 Column <%> in the FOR UPDATE clause is not part of the identified table or view
-12250 A host variable or parameter marker is not allowed in a view definition
-12251 CREATE VIEW statement must include a column list because the SELECT clause contains an expression
-12252 CREATE VIEW statement must include a column list because the SELECT clause contains duplicate column names
-12253 The number of columns specified for the view is not the same as specified by the SELECT clause
-12254 WITH CHECK OPTION cannot be used for the specified view because it cannot be modified
-12258 <%> operation not permitted because the view cannot be modified
-12259 <%> operation not permitted because the joined table cannot be modified
-12261 INSERT statement not permitted because the object column <%> is derived from an expression
-12262 The type of the parameter marker cannot be determined
-12263 Parameter markers and host variables not allowed in EXECUTE IMMEDIATE environment
-12264 Parameter markers may not be specified in SELECT clause
-12265 Literal or computed value overflow
-12266 Decimal divide operation invalid because the result would have a negative scale
-12267 Duplicate column reference in INSERT column list
-12268 Duplicate column reference in UPDATE set clauses
-12269 Duplicate column reference in View Parameter
-12270 <%> does not have <%> privilege on object <%>
-12271 Duplicate column reference in GROUP BY clause

-12273 The types of the results of a CASE expression are not type compatible
-12274 The type of the CASE expression result cannot be determined
-12275 At least one result in a CASE expression must be non-null
-12277 Invalid CAST data type specification
-12278 Invalid EXTRACT field specification
-12280 Invalid datetime literal
-12281 Invalid interval literal
-12282 Invalid interval qualifier
-12283 Invalid use of interval qualifier
-12300 Syntax error in escape clause
-12301 Translated native SQL string was truncated
-12302 The function <%> is not supported
-12303 Unrecognized first word, <%> in escape clause expected 'CALL', 'FN','OJ', 'D', 'T' or 'TS'
-12304 Unrecognized scalar function <%>
-12305 Invalid data type <%> in function CONVERT
-12306 Syntax error, incomplete escape clause
-12307 Syntax error, unmatched apostrophe in string literal
-12308 Syntax error, unmatched quote in delimited identifier
-12309 Unknown first parameter <%> in scalar function <%>
-12310 Argument missing in scalar function <%>
-12311 Too many arguments in scalar function <%>
-12500 A databank named <%> already exists (or filename already used)
-12501 Table <%> does not exist
-12502 <%> does not have <%> privilege
-12503 <%> does not have <%> privilege on object <%>
-12504 Statement not allowed within transaction
-12505 <%> is not a USER or PROGRAM ident
-12506 No privilege
-12507 <%> does not have any databank available
-12509 An ident cannot REVOKE a privilege from itself
-12510 An ident cannot GRANT a privilege to itself
-12511 Duplicate column specification
-12512 Invalid type description
-12516 Qualified column name required
-12517 Object <%> does not exist
-12518 Circular grant of membership between groups not permitted
-12520 An ident cannot GRANT a privilege to itself
-12523 Databank <%> does not exist
-12526 Default value for NOT NULL column <%> missing
-12530 Operand not of type <%>
-12531 Operands not comparable
-12533 Literal or computed value overflow
-12534 Invalid numeric literal
-12535 Invalid identifier, keyword VALUE expected

-12536 Name <%> in PRIMARY KEY specification not recognized as a column name of current table definition
-12537 <%> must be unqualified
-12538 Default value not compatible with domain definition
-12539 Host variable construction illegal in this context
-12540 <%> is not a column of the specified table(s)
-12542 Default value is outside the range specified by domain definition
-12544 Too many columns specified in <%> statement
-12545 Primary key column <%> may not be updated because the table is in a NULL databank
-12546 Column <%> is not type compatible with the corresponding column of the referenced table
-12547 Number of columns specified in the foreign key is not the same as the number of columns in the primary key of the referenced table
-12549 Databank option may not be changed to NULL since <%> contains tables with FOREIGN KEY, REFERENCE, UNIQUE or trigger specifications
-12550 Table <%> includes a FOREIGN KEY or UNIQUE constraint and may therefore not be created in a databank with NULL option
-12551 Table <%> is in a databank with NULL option and may therefore not be used as referential constraint
-12553 Explicit grant membership on PUBLIC is not permitted
-12554 PUBLIC cannot be member of another group
-12556 <%> cannot be shadowed because it is a NULL databank
-12557 Shadow named <%> already exists
-12558 Ident named <%> already exists
-12559 Index named <%> already exists
-12560 Table, View or Synonym named <%> already exists
-12561 Domain <%> not found, domain does not exist or no usage privilege
-12563 Shadow <%> does not exist
-12564 Ident <%> does not exist
-12565 Maximum row length exceeded by index or key table
-12566 Maximum row length exceeded by base table
-12568 EXISTS construction illegal in this context
-12569 ALL or ANY construction illegal in this context
-12570 Set function construction illegal in this context
-12571 Subquery construction illegal in this context
-12572 Too many columns given in FOREIGN KEY clause
-12573 Name <%> in CHECK clause not recognized as a column name of current table definition
-12574 Name <%> in column constraint not recognized as current column name
-12577 Default value not compatible with column specification
-12579 No such unique constraint in referenced table
-12581 Too many columns given in UNIQUE constraint
-12582 UNIQUE constraint equivalent to PRIMARY KEY constraint

-12583 UNIQUE constraint equivalent to previously given UNIQUE constraint
-12585 Name <%> in FOREIGN KEY clause not recognized as a column name of current table definition
-12590 Table contains too many columns
-12591 Cannot create unique index
-12592 Dependencies exist, RESTRICT specified
-12593 Column <%> does not exist
-12594 Column <%> cannot be dropped as it is the only column in table
-12595 Column <%> cannot be dropped, dependencies exist
-12596 Default value for column <%> does not exist
-12597 Change of datatype is not allowed for a column included in a key or index
-12598 The datatype for a column can not be changed to a domain
-12599 The proposed datatype change is not supported
-12600 Change of datatype is not supported for a column used by a view, routine or trigger
-12601 Statement does not support backup of <%>
-12602 The same file name is given for backup and incremental backup
-12603 Database is already OFFLINE
-12604 Database is already ONLINE
-12605 Cannot RESET LOG, because database is ONLINE
-12606 Databank <%> is already OFFLINE
-12607 Databank <%> is already ONLINE
-12608 Cannot RESET LOG, because databank <%> is ONLINE
-12609 Shadow <%> is already OFFLINE
-12610 Shadow <%> is already ONLINE
-12611 Cannot RESET LOG, because shadow <%> is ONLINE
-12612 Shadow <%> is already specified
-12613 Cannot set more than one shadow OFFLINE for databank having shadow <%>
-12614 Statistics cannot be updated for <%> because it is a view
-12615 Filename already used by databank or shadow
-12616 Cannot SET DATABASE OFFLINE, because another user is connected
-12617 Cannot SET DATABANK <%> OFFLINE, because the databank is in use
-12618 INCREMENTAL backups can only be used in conjunction with EXCLUSIVE
-12620 A domain named <%> already exists
-12621 A schema named <%> already exists
-12622 The schema name for the index table must be the same as the schema name for the table
-12623 A PRIMARY KEY constraint for this table has already been defined
-12624 The UNIQUE constraint cannot be created because the columns contain duplicates
-12625 Referential constraint criteria not fulfilled

-12626 Check constraint criteria not fulfilled
-12627 One or more specified columns does not exist in table
-12628 Referenced table or column not found
-12629 FOREIGN KEY not referencing a compatible UNIQUE or PRIMARY KEY
-12630 Constraint <%> cannot be dropped, dependencies exist
-12631 PRIMARY KEY or UNIQUE column constraint not valid when records exist
-12701 <%> is a reserved word, and cannot be used as the name for a symbol
-12702 <%> is a global variable, and cannot be used as the name for a variable or parameter
-12703 The class <%> is already present in the handler declaration
-12704 The SQLSTATE <%> is already present in the handler declaration
-12705 The condition identifier <%> is already present in the handler declaration
-12706 The condition identifier <%> is already used in another handler in this scope
-12707 A condition identifier for the SQLSTATE <%> has already been defined in this scope
-12708 <%> is not a valid SQLSTATE value
-12709 The SQLSTATE <%> associated with the condition identifier <%> is already present in the handler declaration
-12710 An exception handler for the state (<%>) associated with the condition identifier <%> has already been defined in this scope
-12711 An exception handler for the state <%> has already been defined in this scope
-12712 An exception handler for the class <%> has already been defined in this scope
-12713 The default literal is too large for this data type
-12714 The literal value is too large for this data type
-12715 The type of the default value is not compatible with the type of the variable
-12717 Invalid declaration. The maximum precision for this data type is <%>
-12718 The scale cannot exceed the precision
-12720 The number of correlation names for <%>, does not match number of result types
-12721 The parameter <%> for a result set procedure must be declared as IN
-12722 The parameter <%> is declared as IN, and cannot be assigned
-12723 The parameter <%> is declared as OUT, and cannot be used in expressions
-12724 <%> cannot be assigned a value directly
-12725 Result set procedures can only be used in cursor declarations
-12726 The formal argument of the routine is IN but the actual argument is OUT
-12727 The formal argument of the routine is OUT but the actual argument is IN

-12728 Literals or expressions cannot be used for OUT parameters
-12729 Return statements are only allowed in result set procedures or functions
-12730 Recursive call to <%>, not allowed
-12731 <%> statement not allowed in result procedure
-12732 The procedure <%> does not return a result set and cannot be used in a declare cursor for call
-12733 RESIGNAL statement only allowed in exception
-12734 Wrong number of items in result clause, <%> expected
-12735 Wrong number of parameters, <%> expected
-12736 Invalid type for argument, <%> expected
-12737 Too long name <%>
-12738 Duplicate declaration <%>
-12739 x.y.z names not allowed
-12740 Syntax error, <%> assumed to mean <%>
-12741 The label or routine <%> is not defined
-12742 The label <%> is not defined
-12743 The procedure <%> does not exist, (or no execute privilege)
-12744 The variable <%> is not defined
-12745 The condition identifier <%> is not defined
-12746 The cursor <%> is not defined
-12747 Use of domains in PSM control statements not allowed
-12748 Operands are incompatible
-12749 The SQL module <%> already exists
-12750 The procedure <%> already exists
-12751 Duplicate parameter <%>
-12752 The procedure <%> is declared in an SQL module, and cannot be dropped directly
-12753 Failed to read data dictionary
-12754 The length of a host variable cannot exceed 32
-12755 Host variables cannot be used within a procedure
-12756 The data type for the parameter marker cannot be determined
-12757 The number of items in the into clause is less than the number of items in the select list
-12758 The number of items in the into clause is greater than the number of items in the select list
-12759 The number of items in the fetch into clause is less than the number of items in the cursor declaration
-12760 The number of items in the fetch into clause is greater than the number of items in the cursor declaration
-12761 A fetch direction other than next, can only be defined for a scrollable cursor
-12762 The value in a fetch absolute or relative must be an integer
-12763 The argument to fetch absolute must be larger than 0
-12764 A cursor for call is read only and cannot be used in a update or delete where current statement

-12765 The cursor is declared as read only and cannot be used in a update or delete where current statement

-12766 The table name in the <%> statement is not the same as the name used in the cursor declaration

-12767 The column <%> is not specified in the for update list of the cursor declaration

-12768 The size for a data type must be larger than zero

-12769 It is not allowed to declare exception handlers or condition identifiers for the SQLSTATE successful completion ('00000')

-12770 The formal argument of the procedure is declared as INOUT but the actual argument is <%>

-12771 A handler declaration cannot contain both an exception class and SQLSTATE values or condition identifiers

-12772 The procedure does not return a result set and therefore cannot be used with a scroll cursor

-12773 <%> is not a column and is not declared as a variable or parameter

-12774 The label <%> has already been declared

-12775 The statement requires <%> access

-12776 <%> statement is only allowed if the access indication is MODIFIES SQL DATA or READS SQL DATA

-12777 <%> statements are only allowed if the access indication is MODIFIES SQL DATA

-12778 The access indication for a result set procedure must be READS SQL DATA or CONTAINS SQL

-12779 An UNDO exception handler can only be specified in an atomic compound statement

-12781 Only assignment and comparison operations allowed

-12782 The column name <%> has already occurred in this row declaration

-12783 The field <%> is not defined in the row data type

-12784 The row data types do not have the same number of fields

-12785 A row data type variable may not be used as a parameter or result type nor in a DML statement

-12800 Functionality not supported

-12801 Referencing OLD TABLE is not allowed if trigger event is INSERT

-12802 Referencing NEW TABLE is not allowed if trigger event is DELETE

-12803 The compound statement in a triggered action must be atomic

-12804 Referencing OLD or NEW ROW may only be used if FOR EACH ROW is specified

-12805 A column list can only be specified if trigger event is UPDATE

-12806 Duplicate column name in OF list

-12807 It is not allowed to modify the <%> table

-12808 The trigger time for a view must be INSTEAD OF

-12809 The trigger time for a base table can not be INSTEAD OF

-12810 Only the creator of a table is allowed to create a trigger for the table

-12811 It is not allowed to create triggers for tables, located in databanks with NULL option

-12812 Referencing OLD or NEW table is not allowed in FOR EACH ROW triggers
-12813 A trigger with the name <%> already exists
-12830 A function with the name <%> already exists
-12831 The function <%> does not exist, (or no execute privilege)
-12832 The result of the expression is not deterministic while the routine is declared as deterministic
-12833 All privileges used in a trigger must be held with grant option
-12834 The sequence <%> does not exist
-12835 A sequence with the name <%> already exists
-12836 The keyword NULL cannot be used <%>
-12837 The operands to a overlaps predicate must be of a row data type with two elements
-12838 The two elements in an operand to the overlaps predicate must either be of the same type or otherwise it shall be possible to add the second value to first value
-12839 <%> is not allowed in a before trigger
-12840 The simple value specification for a get diagnostics statement must be of integer type
-12841 <%> is not allowed in a trigger
-12842 The increment for a sequence must be non zero
-12843 Invalid values for sequence attributes
-12844 The schema <%> does not exist, (or no privilege)
-12845 The schema name for routines in a module definition must be the same as the schema name for the module
-12846 The value for diagnostics size must be positive
-12847 The ident name in an authorization clause must be the same as the current user
-12848 Constraint name already exists
-12849 The function <%> MODIFIES SQL DATA and can thus not be used in a DML statement
-12850 A trigger must be located in the same schema as the table
-12851 A constraint must have the same schema name as the object to which it belongs
-12852 The schema name for a routine is not the same as the schema name for the module
-12853 Ident name not allowed as a schema with the same name exists
-12854 A non-deterministic expression is not allowed in a check clause
-12855 Default values with a reference to a sequence combined with a check clause is not allowed in an alter table statement

B.2.6 Program-dependent errors

These errors arise as a result of incorrect program construction. In general, corrective action requires revision of the program source code.

-14001 Invalid sequence of SQL statements
-14002 SQL statement invalid because the user is not connected

-14003 CONNECT statement invalid because the user is already connected
-14004 System already closed down
-14005 Cannot perform DISCONNECT in a transaction
-14006 Login failure
-14011 Transaction already started
-14012 Transaction handling required
-14013 No transaction started
-14014 Cannot perform write operations as transaction status is read-only
-14015 Commit or rollback statements are not allowed in an atomic execution context
-14016 Select for update is not allowed for a read-only cursor
-14017 Mixing DDL and DML statements in a transaction is not allowed
-14018 Incremental backup not allowed in BACKUP transaction
-14019 Operation not allowed in BACKUP transaction
-14020 START BACKUP command required to perform an online backup
-14021 Cannot perform ENTER or LEAVE in a transaction
-14022 Cannot perform ENTER operation because program level is already active
-14023 No program level entered, cannot perform leave operation
-14024 Session statements are not allowed in a trigger
-14031 DESCRIBE statement does not identify a prepared statement
-14032 EXECUTE statement does not identify a prepared statement
-14033 PREPARE statement identifies a SELECT statement of an opened cursor
-14034 The cursor is not in a prepared state
-14035 The cursor identified in a FETCH or CLOSE statement is not open
-14036 The cursor cannot be used because its statement name does not identify a prepared SELECT statement
-14037 The cursor identified in the UPDATE or DELETE statement is not open
-14038 UPDATE or DELETE CURRENT statement not allowed for a cursor of a prepared SELECT statement
-14039 Cursor is not scrollable
-14041 The cursor identified in the UPDATE statement is not positioned on a row
-14042 The cursor identified in the DELETE statement is not positioned on a row
-14043 Routine signaled SQLSTATE
-14044 Routine signaled a condition
-14045 Prepared statement not a cursor specification
-14046 The statement RESIGNAL was used outside a exception handler
-14101 Invalid statement identifier
-14201 Compilation did not yield an executable program
-14202 The cursor identified in an OPEN statement is already open, but not declared as REOPENABLE
-14203 Statement position cannot be saved when temporary tables are used in the query

-14210 Cursor for result set procedure may not be reopenable
-14211 With hold option is not available for result set procedures
-14301 SQLDA contains an invalid data address or indicator variable address
-14302 Invalid address of username or password
-14303 Invalid address
-14311 Illegal statement length given for SQL statement
-14312 Input character string too long
-14313 Like pattern string too long
-14314 Username or password too long
-14315 Illegal byte length of floating point number
-14316 Unterminated C string, null byte not found
-14321 Illegal host variable type
-14322 Illegal host variable type in like pattern string
-14323 Username and password must be fixed length character strings
-14324 Illegal indicator variable type
-14331 The number of provided host variables does not match the number of parameters
-14332 Using clause required for dynamic parameters
-14333 Using clause required for result fields
-14401 Column cannot be updated because it is not identified in the UPDATE clause of the SELECT statement of the cursor
-14402 The table identified in the UPDATE or DELETE statement is not the same as that designated by the cursor
-14403 Cannot describe statement without naming information
-14404 Cannot update table <%> because the declare statement is read-only
-14405 Cannot delete from table <%> because the declare statement is read-only
-14406 Unexpected statement type encountered in an UPDATE or DELETE statement
-14501 Database connection is not open
-14601 Invalid cursor state
-14611 Using clause does not match dynamic parameters
-14612 Using clause or Into clause does not match target specifications
-14621 Cursor not found
-14622 Ambiguous cursor name
-14623 Invalid cursor name
-14624 Cursor already allocated for statement
-14631 Invalid SQL statement name
-14641 Invalid SQL descriptor name
-14642 Invalid descriptor index
-14643 Invalid descriptor count
-14651 Invalid exception number
-14700 No return statement was encountered during the execution of a function
-14701 The statement is not allowed in an atomic execution context

-14702 The same column has been updated to different values when executing a trigger
-14703 An exception occurred during the execution of a trigger
-14720 The maximum_value for the sequence <%> has been reached
-14721 There is no current_value for the sequence <%> because the next_value function has not been invoked
-14722 Sequence <%> locked by another user
-14723 Update of attributes for the sequence <%> failed

B.2.7 Databank and table errors

These errors are associated with problems of physical access to databanks and tables. Locking errors should not result from transaction conflicts, but generally indicate either locking at the operating system level or malfunction of the internal Mimer SQL routines. Many of the errors in this class are corrected by action taken at the operating system level. If errors persist in spite of corrective action, contact your Mimer representative.

-16001 Table locked by another user
-16002 Table locked by another cursor
-16003 Table in referential constraint definition locked by another user
-16004 Table in referential constraint definition locked by another cursor
-16005 Log locked by another user
-16006 Backup unit log file locked
-16101 No databank <%> found in dictionary
-16135 Record no longer exists
-16141 Syntax error in filename for databank <%>
-16142 Cannot open databank <%>, file <%> not found
-16143 File protection violation for databank <%>, file <%>
-16144 Cannot open databank <%>, file <%> locked by other user
-16145 Too many databanks open concurrently (direct access I/O limit)
-16146 File create error, disk full
-16147 File create error (quota exceeded) for databank <%>, file <%>
-16148 Device or network connection not ready, databank <%>, file <%>
-16149 Cannot open databank <%> in file <%>
-16150 Tried to access databank <%> on node which is inaccessible because TRANSDB is OFFLINE
-16151 Too many databanks open concurrently
-16152 Tried to open a non-MIMER databank
-16154 Table control area exhausted
-16155 Incompatible version of databank <%>
-16156 Databank <%> belongs to another system databank
-16157 Tried to open a read-only databank with write access
-16159 Old version of the databank <%> cannot be accessed without restoring the databank with the backup and restore utility
-16160 Cannot set TRANSDB shadow OFFLINE on the same node as the master TRANSDB
-16161 Databank <%> disk space exhausted

-16162 Databank LOGDB disk space exhausted
-16163 Databank TRANSDB disk space exhausted
-16164 Databank SQLDB disk space exhausted
-16172 Databank <%> locked by another MIMER/DB user
-16182 Databank <%> corrupt
-16183 Bad parameter
-16184 I/O error
-16185 Internal databank identifier invalid
-16186 Internal table identifier invalid
-16187 Shadow <%> in file <%> has illegal sequence number
-16189 Corrupt bitmap page
-16190 Table root entry not found
-16191 Exclusive access to databank required for attempted operation
-16192 Load not allowed in databank with TRANS or LOG option
-16193 TRANSDB and/or LOGDB not open
-16194 Error occurred in transaction commit phase
-16195 Internal inconsistency detected
-16196 No end of table mark found for tableid
-16197 Shadow <%> is already OFFLINE
-16198 Shadow <%> is already ONLINE
-16199 Result of bitmap page I/O undefined
-16200 Result of index page I/O undefined
-16201 Result of root page I/O undefined
-16202 Result of data page I/O undefined
-16203 Corrupt index page
-16204 Corrupt root page
-16205 Corrupt data page
-16206 Write set corrupt
-16207 Table <%> has invalid record length
-16208 Unable to open databank <%>. SHADOW license required
-16209 Unable to open databank <%>. NETIO license required
-16210 Cleanup control area exhausted
-16211 Not properly closed, dbcheck initiated
-16212 TRANSDB restart directory corrupt
-16213 Error when closing databank file. Please consult multiuser log file
-16214 Blockdata DKBLK1 missing
-16215 Error accessing remote TRANSDB, node will not be accessed further
-16216 Blocksize not supported
-16217 Error when generating internal key
-16218 Operation not allowed. Licensed number of users exceeded
-16219 Too many multi systems started
-16220 Unable to retrieve limit on number of allowed users
-16221 Lost contact with peer
-16222 Record length from update before is invalid
-16224 Transaction state table entries exhausted

-16225 Transaction state identifier invalid
-16226 Invalid function code
-16227 Commit set corrupt
-16228 Restart set corrupt
-16229 Cancel requested
-16230 Transaction cache inconsistent
-16231 Shadow <%> is inaccessible due to incomplete CREATE SHADOW or SET ONLINE operation
-16232 Database upgrade required
-16233 Operation not allowed. Licensed number of users exceeded
-16234 Execution interrupted by scheduler
-16401 Routine cannot be dropped because it is in use
-16402 Routine cannot be used because it is being dropped

B.2.8 Miscellaneous errors

These errors arise from miscellaneous problems which do not fall into the other classes. If the corrective action is not indicated by the error description, contact your Mimer representative for help.

-18001 Blockdata BLKDS2 not included
-18002 Cannot log in, error in SYSDB initialization
-18003 No privilege to open log file
-18004 Databank LOGDB already opened by another MIMER/DB user. Could not be opened exclusively to drop log records
-18005 Unknown language
-18006 Language not properly installed
-18007 Unable to fetch message text
-18008 Restore in wrong sequence attempted
-18009 Mismatching version of Embedded SQL and MIMER/DB
-18010 Invalid log record found during restore operation
-18011 Mismatching version of MIMER/DB and utilities
-18012 The transformation of a TRANSDB shadow to master was interrupted before completion. Please login to UTIL to complete the transformation
-18013 MIMER/DB started from SYSDB shadow. Transform SYSDB shadow to master with UTIL, or restart system from master SYSDB
-18014 Alter shadow not allowed in SQL for system databanks. Use utility program instead.
-18015 Open with hold is not possible when temporary tables are used for evaluation of the query
-18016 Cursor could not be opened with hold as it is not prepared with hold
-18017 With hold functionality not supported
-18018 The network server version does not support scroll
-18019 Bad parameters passed to DBAPI4
-18020 Unknown information code = <%> used

-18021	Only SELECT, INSERT, UPDATE, and DELETE operations (excluding where current forms) may be compiled together in a single statement
-18022	Distributed transactions not supported by server
-18023	Requested DTC function not supported by server
-18024	Failed to enlist transaction in distributed transaction
-18025	Unable to retrieve transaction manager's where-abouts
-18026	Failed to import transaction identifier
-18027	Statement already active in other transaction
-18028	Cannot initiate a new PSM debugger session, because the number of request threads is insufficient
-18029	Execution interrupted by debugger
-18041	Update of primary key columns for a table located in a databank, with NULL option is not allowed
-18042	Primary key columns may not be updated by Level2 applications
-18043	The rowid column may not be updated
-18101	Operation not allowed. SHADOW license required
-18102	Operation not allowed. MIMER/DB license required
-18103	Operation not allowed. MIMER/DB Level2 license required
-18104	Operation not allowed. MIMER/NET Client license required
-18105	Operation not allowed. MIMER/NET Server license required
-18106	Operation not allowed. MIMER/NET PC-Server license required
-18107	Operation not allowed. Beta test version of MIMER requires BETA license.
-18108	Error opening MIMER keyfile
-18121	Operation not allowed. VAR specific MIMER license required
-18122	Authorization failure. Invalid attempt to connect
-18201	SYSDB cannot be backed up using CREATE BACKUP without an ONLINE shadow
-18231	<%> records dropped from LOGDB
-18232	Shadow <%> is OFFLINE
-18233	Unable to access databank <%>, because it is OFFLINE
-18234	Error <%> occurred when trying to access shadow <%>
-18235	Error <%> occurred when trying to access databank <%>
-18236	Statistics updated for <%> tables
-18237	Databank <%> does not have LOG option
-18238	<%> records copied to incremental backup
-18239	Unable to CONNECT, because database is OFFLINE
-18240	Unable to CONNECT, because database is OFFLINE and one connection already exists
-18241	Unable to CONNECT, because SYSDB is OFFLINE
-18242	Unable to CONNECT, because SYSDB is OFFLINE and one connection already exists
-18243	Unable to access databank <%>, because database is OFFLINE
-18244	Could not get exclusive access to the database because one or more connections already exists

-18245 Could not connect to database <%>, a system administrator is executing a statement that requires exclusive access to the database
-18500 Database <%> not found in MIMER_SQLHOSTS
-18501 Database <%> unknown on remote node
-18502 Handshake message invalid, incompatible protocol
-18503 Only remote databases are allowed, specify database which is not local
-18504 The network server version is not compatible
-18505 Local memory pool in network server exhausted (SQLPOOL)
-18506 In the current version only one local (and several remote) databases can be connected at a time
-18507 Unknown connection name <%> specified
-18508 Already connected, connection name <%>
-18509 Database name <%> invalid, too long or contains invalid characters
-18510 Connection name <%> invalid, too long or contains invalid characters
-18512 Illegal reentrant call
-18513 Use another TCP/IP port number
-18514 Too deep address indirection
-18515 Cannot get value of thread-local variable
-18516 The network server version does not support Level2
-18518 Catalog version from betatest. See Release Notes, how to upgrade
-18519 Erroneous contents in MIMER_SQLHOSTS
-18521 Error opening MIMER_SQLHOSTS, filename syntax error
-18522 Error opening MIMER_SQLHOSTS, file not found
-18523 Error opening MIMER_SQLHOSTS, file protection violation
-18524 Error opening MIMER_SQLHOSTS, file locked
-18525 Error opening MIMER_SQLHOSTS, too many opened files
-18526 Error opening MIMER_SQLHOSTS, file create error, disk space exhausted
-18527 Error opening MIMER_SQLHOSTS, other error (-7)
-18528 Error opening MIMER_SQLHOSTS, other error (-8)
-18529 Error opening MIMER_SQLHOSTS, other error (-9)
-18530 Error opening MIMER_SQLHOSTS, illegal access options
-18550 Invalid network package format
-18551 Unknown request code in network package (<%>)
-18552 Network package longer than expected
-18553 Internal data structures corrupt (DSNEE4)
-18554 The UTILITY program does not have client/server support
-18594 Query timeout period expired
-18595 Network partner disconnected
-18601 Could not connect to database <%>, unknown node '<%>'
-18602 Could not connect to database <%>, unknown protocol '<%>'
-18603 Could not connect to database <%>, unknown interface '<%>'
-18604 Could not connect to database <%>, unknown service '<%>'

-18605 Could not connect to database <%>, chosen protocol not supported on ALPHA/VMS '<%>'
-18606 Could not connect to database <%>, network type not supported '<%>'
-18607 Could not connect to database <%>, remote node is unreachable '<%>'
-18608 Bad parameter NETID=<%> passed to network package
-18609 Invalid parameter RECLEN=<%> passed to network package
-18610 Invalid parameter BUFFER=<%> passed to network package
-18611 Too many concurrent network connections
-18612 Connection refused '<%>'
-18613 Unexpected network event '<%>'
-18614 The underlying network protocol does not have enough capabilities '<%>'
-18615 Network service busy '<%>'
-18616 Local or remote system resources are insufficient '<%>'
-18617 Connection timed out '<%>'
-18618 Insufficient privileges for attempted network operation '<%>'
-18619 Unexpected network error '<%>'
-18620 Network operation would block (asynch mode)
-18621 Could not load network library
-18622 Required routines missing from network library
-18901 Multi-user system not started
-18902 MIMER logins are currently disabled, try again later
-18903 Access denied to MIMER multi-user system
-18904 Unable to attach to multi-user system, no response
-18905 Operation not allowed. Licensed number of users exceeded
-18920 Machine dependent error-18920
-18921 Machine dependent error-18921
-18922 Machine dependent error-18922
-18923 Machine dependent error-18923
-18924 Machine dependent error-18924
-18925 Machine dependent error-18925
-18926 Machine dependent error-18926
-18927 Machine dependent error-18927
-18928 Machine dependent error-18928
-18929 Machine dependent error-18929

B.2.9 Internal errors

These errors arise from malfunction in internal Mimer SQL routines. Contact your Mimer representative for help.

-19001 Program level list corrupt
-19002 No program level found
-19003 Statement list corrupt
-19004 Output parameter list corrupt

-19005	Table list corrupt
-19006	Unable to find log file, LOGDB corrupt
-19007	Inconsistency detected when trying to update dictionary
-19008	Unable to open SYSTEM base tables
-19009	Dictionary table SYSTEM.USERS corrupt
-19010	Unable to extract correct information from SYSTEM.DATABANKS
-19011	Unable to extract correct information from SYSTEM.TABLE_CONSTRAINTS
-19012	Could not find foreign key tableid (<%>) in SYSTEM.TABLE_CONSTRAINTS
-19015	Sysid record in SYSTEM.OBJECTS not found
-19017	Invalid MAE program
-19018	Invalid operation code <%> at PC=<%>
-19019	Missing IMC instruction
-19020	Invalid function code passed to instruction <%>
-19021	No databank control block found for <%>
-19022	Bad function code passed to DSCDB3
-19023	Invalid pointer to naming structure
-19024	Severity message program corrupt
-19025	Invalid table descriptor
-19026	Invalid table descriptor, log status invalid
-19027	Base table must be opened before index tables
-19028	Table root entry not found
-19029	Unable to change position on write set because no mark is set
-19030	Invalid length for allocation of program space
-19031	Invalid table type
-19032	No table control block found
-19033	Cannot delete databank file outside transaction
-19034	Bad function code passed to DSCRD2
-19035	Invalid index descriptor
-19036	Error detected when closing table, hash chain corrupt
-19037	Invalid internal table type encountered
-19038	Write set inconsistency encountered
-19039	Invalid internal statement identifier
-19040	Invalid internal system identifier
-19041	Invalid internal user identifier
-19042	The static statement cannot be compiled because it is already identified with some other statement
-19043	The statement cannot be prepared because it is already identified with a static statement
-19044	Transaction control block chain corrupt
-19045	Shadow <%> cannot be transformed because it is OFFLINE
-19046	Databank <%> is referenced but not opened
-19047	Table has not been opened with sufficient access to allow current operation

-19048 Databank <%>, no shadow is found in dictionary with sequence number = <%>
-19049 The internal update operation has not been prepared with the old record
-19050 Catalog version not compatible with server
-19051 Compiled LIKE pattern corrupt
-19052 Could not store lookup path, inconsistency detected
-19053 Output descriptor overflow
-19054 Unable to initialize database system
-19055 Unable to generate an internal key
-19056 Inconsistent user identifier (not logged in)
-19057 Scroll program corrupt
-19058 Extended name not supported in static SQL
-19059 Invalid error message descriptor
-19061 Loss of significance for VARCHAR length
-19062 Positive overflow for VARCHAR length
-19063 Negative overflow for VARCHAR length
-19065 Bad parameter for VARCHAR length
-19066 Illegal operand for VARCHAR length
-19067 Bad record number
-19068 No matching record
-19069 Corrupt cancel state
-19071 Unrecognized data types for conversion
-19072 Invalid read set record
-19073 Insufficient internal descriptor area
-19074 Internal inconsistency encountered in TCACHE
-19075 Invalid internal DDU identifier
-19076 Internal inconsistency encountered in table list
-19077 Internal inconsistency encountered in DU1
-19078 Invalid parameter encountered in MOS
-19079 Internal inconsistency encountered in DSETH3
-19080 Thread initialization failure
-19081 Hash table missing in system control block
-19082 Internal inconsistency encountered in DSGSH2
-19083 Internal inconsistency encountered in MCOMEM
-19084 Runtime assertion failed
-19085 Invalid descriptor encountered on server
-19086 Invalid statement status encountered on server
-19087 PSM debugger resources already allocated
-19088 PSM debugger resources already deallocated
-19089 Invalid parameter encountered in DDU
-19091 Invalid function code passed to routine
-19092 Invalid function code 2 passed to routine
-19093 Invalid function code 3 passed to routine
-19201 System error : <%>- Area outside MAE data storage
-19202 System error : <%>- Attempt to qqwsal() in closed area

-19203 System error : <%>- Cost value out of range
-19204 System error : <%>- Error converting into Mimer format
-19205 System error : <%>- Error from MDRCCI call
-19206 System error : <%>- Error when reading databank option
-19207 System error : <%>- Expression switch case not recognized
-19208 System error : <%>- Factor was left unused
-19209 System error : <%>- Failed to get a slave RST
-19210 System error : <%>- Generation stack underflow
-19211 System error : <%>- Group is not allocated
-19212 System error : <%>- Host variable not defined
-19213 System error : <%>- Host variable number mismatch
-19214 System error : <%>- Illegal Set Func. mode switch case
-19215 System error : <%>- Illegal Status switch case <%>
-19216 System error : <%>- Index table not found
-19217 System error : <%>- Invalid base pointer
-19218 System error : <%>- Invalid object type
-19219 System error : <%>- Invalid pointer
-19220 System error : <%>- Main switch case not recognized
-19221 System error : <%>- Multiple offset assignment
-19222 System error : <%>- Multiple restriction groups
-19223 System error : <%>- No area opened
-19224 System error : <%>- Nonexistent member
-19225 System error : <%>- No Tbl_desc for SCO
-19226 System error : <%>- NOT stack overflow
-19227 System error : <%>- NOT stack underflow
-19228 System error : <%>- Offset outside MAE data storage
-19229 System error : <%>- qqcbix() with illegal operator
-19230 System error : <%>- qqcunx() with illegal operator
-19231 System error : <%>- Error from MDRCFC call
-19232 System error : <%>- qqrlst() with NULL list
-19233 System error : <%>- qqwlst() with NULL list
-19234 System error : <%>- Query result stack underflow
-19235 System error : <%>- Query stack underflow
-19236 System error : <%>- Rule matrix index out of range
-19237 System error : <%>- Scan kind not implemented
-19238 System error : <%>- Scan stack underflow
-19239 System error : <%>- Selectivity factor value out of range
-19240 System error : <%>- Semantic stack underflow
-19241 System error : <%>- Set range violation
-19242 System error : <%>- Set size incompatibility
-19243 System error : <%>- Stack underflow
-19244 System error : <%>- Statement switch case not recognized
-19245 System error : <%>- Switch case not recognized
-19246 System error : <%>- Too complicated UNION query
-19247 System error : <%>- Too many nested subqueries
-19248 System error : <%>- Traversal stack underflow

-19249 System error : <%>- Unexpected EXPRESSION in HOST variables
-19250 System error : <%>- Unexpected expression operand
-19251 System error : <%>- Unexpected expression subtype
-19252 System error : <%>- Unexpected node class
-19253 System error : <%>- Unexpected SELECT ITEM
-19254 System error : <%>- Unexpected statement subclass
-19255 System error : <%>- Unexpected DD return code <%>
-19256 System error : <%>- Unknown Host Variable type
-19257 System error : <%>- Unknown statement type
-19258 System error : <%>- WS stack overflow
-19259 System error : <%>- X stack overflow
-19260 System error : <%>- X stack underflow
-19261 System error : <%>- Error logging is not enabled
-19262 System error : <%>- Source position line or column is negative
-19263 System error : <%>- Message insert string too long
-19264 System error : <%>- Error logging is already enabled
-19265 System error : <%>- MAE constant storage overflow
-19266 System error : <%>- Selectivity rule number out of range
-19267 System error : <%>- No entry for index id
-19268 System error : <%>- qqcfnx() with illegal operator
-19269 Unexpected duplicate row found in temporary table
-19270 System error : <%>- Scan queue underflow
-19280 System error : <%>- Error from MDRTDC call
-19290 Out of memory
-19291 Invalid attribute type
-19292 Error when trying to store procedure in dictionary
-19293 Error when trying to share program
-19300 <%> unhandled production
-19301 Internal inconsistency detected in PSM
-19901 Function not yet implemented

B.2.10 Communication errors

Error codes from the communication kernel layer (network routines):

-21001 Already listening on service <%>
-21002 Error trying to ASSIGN channel for TCP/IP communication
-21003 Error when creating socket
-21004 Error when binding socket address for service <%>
-21005 Error when getting port number for service <%>
-21006 Illegal protocol specified: <%>
-21007 Error when looking up host name: <%>
-21008 Error when connecting to database <%> on <%> using <%> to service <%>
-21009 Illegal channel id specified
-21010 Error when reading data from network channel
-21011 Error when writing data to network channel

-21012 Channel is not open
-21013 Channel is not accessible from this process
-21014 Error when creating mailbox
-21015 Error when declaring network object for service <%>
-21016 Unimplemented feature
-21017 Error when accepting new channel
-21018 Error when doing local listen for database <%> on path <%>
-21019 Too many channels used
-21020 Multiple read requests on channel
-21021 Multiple write requests on channel
-21022 Local write when not owning buffer
-21023 Cancel request illegal on channel
-21024 No available channel id number
-21025 Tried to open too many local servers
-21026 Database server for database <%> not running
-21027 Incompatible buffer versions
-21028 Failed to do a LOCAL connection to the server for database <%>
-21029 Illegal reentrant request on channel
-21030 Network request would block caller
-21031 Too large network I/O requested
-21032 Could not find DSINI4 in single-user library
-21033 Could not find DSHND4 in single-user library
-21034 Could not find DSGMD4 in single-user library
-21035 Could not find DSUMP4 in single-user library
-21036 The channel is closed
-21037 The specified network interface is not supported
-21038 Could not lock communication buffers in memory
-21039 Error trying to ASSIGN channel for DECNET communication
-21040 Could not map library for single-user mode
-21041 Could not initialize CK package
-21042 Error when performing initial communication with database server
-21043 Server rejected connection to database <%> on <%> using <%> to service <%>
-21044 Server rejected named pipe connection to database <%>
-21045 The address family for network protocol was unknown
-21046 Error when creating named pipe server objects
-21047 Error when setting up TCP server objects
-21048 Unexpected communication error
-21049 Error when reading/writing data to/from network channel
-21050 Error when closing communication with database server
-21051 All local communication slots are in use
-21052 Database server request failed
-21053 Database or network service not started Error when connecting to database <%> on <%> using <%> to service <%>
-21054 Database server for database <%> not started

-21055 The MIMER network service on <%> for <%> does not currently accept new connections. Try again later
-21056 Local communication has been disabled for database server <%>
-21057 Named pipe communication has not been enabled for database server <%>
-21058 TCP/IP communication has not been enabled for database server <%>
-21100 Command timed out
-21101 Error mapping MCS (MIMER Control Storage)
-21102 Error when doing system communication through the MCS
-21103 MCS communication area is busy. Try again later
-21104 Server for database <%> is already started
-21105 Illegal directory specified for SYSDB82.DBF
-21106 Error in parameter file
-21107 Error when starting database server process
-21108 Error when looking up database name
-21109 Error when creating memory pool in database server
-21110 Could not allocate space from SQLPool
-21111 Error when initiating the ENQ/DEQ package
-21112 Error when attaching a thread to the ENQDEQ area
-21113 Error when initiating server I/O package
-21114 Error when setting default directory for database server I/O package
-21115 Could not start database server thread
-21116 Protocol error- received new request before completion of last request
-21117 Could not create proper execution environment
-21118 Database server not operational
-21119 Notification thread failed. Server can no longer respond to mimcontrol
-21120 Illegal directory path
-21121 Could not create new directory
-21122 Channel closed by administrator
-21123 Invalid channel number specified
-21124 Error when initiating request (rq) queue
-21125 Could not lock the bufferpool in memory
-21126 Database server halted. Failed to generate automatic database dumps
-21127 Database server halted. Dump files from the failed database are placed under <%>
-21128 Error when stopping database server process
-21129 Error when deleting memory pool in database server
-21130 Error getting database server parameters
-21131 Must be superuser to perform this function
-21132 The environment variable MIMER_HOME must point to the MIMER distribution
-21133 An illegal combination of command switches was specified
-21134 The database parameter must be specified
-21135 Permission denied

-21180 Error opening SQLHOSTS file
-21181 Error opening SQLHOSTS file - file name syntax error
-21182 Error opening SQLHOSTS file - file not found
-21183 Error opening SQLHOSTS file - file protection violation
-21184 Error opening SQLHOSTS file - file is locked
-21185 Error opening SQLHOSTS file - too many files are opened
-21186 Error opening SQLHOSTS file - file creation error (diskspace exhausted)
-21187 Error opening SQLHOSTS file - machine dependent code -7
-21188 Error opening SQLHOSTS file - machine dependent code -8
-21189 Error opening SQLHOSTS file - other error
-21190 Error opening SQLHOSTS file - illegal access option
-21191 Could not find a local definition for the specified database name

Error codes used by the server when creating database dumps. These codes are never returned to application programs:

-21200 Error when creating dumpfile
-21201 Error when writing dumpfile

Error codes reflecting problems in the layer that creates and interprets network packets:

-21300 Too large network buffer requested on client side

B.2.11 JDBC errors

These errors arise when the Mimer JDBC Driver fails for some reason. The error codes are in the range -22000 to -22999. When using Java, the error message is always included in the exception that is thrown.

To get the complete and accurate list of error codes, execute the following command:

```
$ java com.mimer.jdbc.Driver -errors
```

C DEPRECATED FEATURES

Some non-standard features in earlier versions of Mimer SQL are deprecated, but retained for backward compatibility. Where these features have equivalents in the standard implementation, only the standard form is documented in the main body of this manual. Use of the standard forms is strongly recommended.

C.1 INCLUDE SQLCA

The SQLCA communication area is deprecated feature in Mimer SQL.

For backward compatibility reasons the use of the SQLCA is still supported.

Applications should now use the SQLSTATE variable and the GET DIAGNOSTICS statement to get all the information previously obtained from SQLCA.

See Chapter 3 for a description of SQLSTATE and GET DIAGNOSTICS.

C.2 SQLCODE

The use of SQLCODE to retrieve the internal Mimer SQL return code for an exception has been replaced by the use of SQLSTATE and GET DIAGNOSTICS with the NATIVE_ERROR option (see Section 10.3) in Mimer SQL. For compatibility reasons, return codes can still be retrieved in SQLCODE, as in earlier versions of Mimer SQL. Mimer SQL will assume the existence of an SQLCODE variable in the application if no INCLUDE SQLCA statement is found and neither SQLSTATE nor SQLCODE is declared between BEGIN DECLARE SECTION and END DECLARE SECTION.

The values of SQLCODE are the same as the values for the internal Mimer SQL return codes described in Appendix B.

C.3 SQLDA

The SQL descriptor area SQLDA, which was used in earlier versions of Mimer SQL, has now been replaced by a standardized SQL descriptor area. The SQLDA area was allocated and maintained by constructions in the host language. The new SQL descriptor area is allocated and maintained by standardized embedded SQL statements.

The former SQL descriptor area SQLDA is still supported in Mimer SQL for compatibility reasons.

C.4 Parameter marker representation

In earlier versions of Mimer SQL it was documented that a parameter marker in a dynamic SQL statement could be represented by either a question mark or a parameter name preceded by a colon. The non-standard representation of a parameter name preceded with a colon is now deprecated, but it is still supported for compatibility reasons.

C.5 VARCHAR(size)

In earlier versions of Mimer SQL a VARCHAR structure was documented, which was used in the C language for handling variable-length character strings. This VARCHAR structure was defined as:

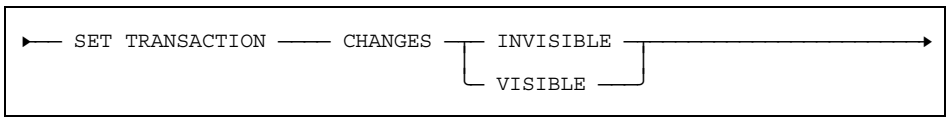
```
#define VARCHAR(x)
        struct {
            short       len;
            char        text[x];
        }
```

where the character string is stored in *text,* and the length of the text is stored in *len.*

C.6 SET TRANSACTION

Following the introduction of the SET TRANSACTION READ and SET TRANSACTION ISOLATION LEVEL options, the SET TRANSACTION CHANGES options no longer apply.

The following options are supported for backward compatibility only:



C.7 DBERM4

The DBERM4 routine, which could be used to retrieve the internal Mimer SQL return code and error message text for an exception, is now deprecated. The GET DIAGNOSTICS statement should now be used to retrieve these exception information items (see Section 10.3).

D APPLICATION PROGRAM EXAMPLES

This chapter illustrates the use of ODBC and embedded SQL in application programs in each of the host languages supported by Mimer SQL (the C code examples can be used with C++). The source code for the application examples is supplied with the standard distribution of Mimer SQL.

The Section D.2 of this chapter illustrates an example of how dynamic SQL can be used with Mimer SQL. This example is only supplied in the C language.

D.1 EXAMPLE program

This simple EXAMPLE program is written according to international SQL standards and it should be possible to use them on any standard-compliant database management system without modification.

The program logs in to the default database with the user ident SYSADM and password SYSADM (change the program as appropriate). The program will then select and print all the rows of the X/Open standard system catalog view INFORMATION_SCHEMA.TABLES, i.e. lists all tables in the system.

D.1.1 Building the EXAMPLE program

Unix

The following example assumes Mimer SQL is installed in the directory /opt/mimer821A and illustrates how the C version of the EXAMPLE program is compiled and built by using the distributed example makefile:

```
$ mkdir example           # Do everything in a sub directory
$ cd example
$ cp /opt/mimer821A/examples/ex_makefile ./makefile
$ cp /opt/mimer821A/examples/example.ec .
```

Take a copy of the makefile and update it so that the MYPROG symbol is set to "example", as the name of the program to be created (avoid trailing spaces).

```
MYPROG = example
```

Then make the example program:

```
$ export MIMER_HOME=/opt/mimer821A
$ make
.
.
```

VMS

The following example illustrates how the FORTRAN version of the EXAMPLE program is compiled and linked under VMS. The programs written in the other languages can be linked in a similar manner.

```
$ ESQL/FORTRAN MIMEXAMPLES8:EXAMPLE   ! Preprocess source code
$ FORTRAN EXAMPLE                     ! Compile preprocessed code
$ LINK EXAMPLE,MIMLIB8:MIMER/OPT      ! Link an executable program
```

Win

On Windows platforms, the examples files are installed in the **dev** directory which is located below the installation directory. The dev directory contains a makefile called **makefile.mak**, which will build all the example programs at the same time. The examples have been tested using Microsoft Visual C.

These instructions assume that Microsoft Visual C is used and that the make command is being executed from the dev directory, appropriate adjustments must be made if this is not the case.

```
$ nmake -f makefile.mak
```

D.1.2 C source code for the EXAMPLE program

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

/*
**  Example of X/Open CAE SQL (1992) compliant embedded program.
**
**  The program tries to connect to the DEFAULT database as SYSADM
**  with password SYSADM.
**  It will then declare a cursor for the standard view
**  INFORMATION_SCHEMA.TABLES, and fetch all records from it.
**
**  If any error occurs, the print_sqlerror routine will be called
**  to print an error message.
*/


/*
**  X/Open does not allow the same variable to be defined in several
**  DECLARE SECTIONs, so we declare SQLSTATE here so that all routines
**  in the file can use the same variable.
*/
exec sql BEGIN DECLARE SECTION;
char SQLSTATE[6];
exec sql END DECLARE SECTION;


void print_sqlerror()
/*
**  print_sqlerror prints an error message for the latest error.
**  Programmed according to the X/Open CAE specification.
*/
{
    exec sql BEGIN DECLARE SECTION;
    int i;
    int exceptions;
    VARCHAR message[255];
    exec sql END DECLARE SECTION;

    exec sql GET DIAGNOSTICS :exceptions = NUMBER; /* How many exceptions? */
    for (i=1; i<=exceptions; i++) {
             exec sql GET DIAGNOSTICS EXCEPTION :i
                         :message = MESSAGE_TEXT;
             printf("%s\n", message);
    }
}
```

```
main()
{
    exec sql BEGIN DECLARE SECTION;
    VARCHAR schema[129];
    VARCHAR table[129];
    char type[21];
    exec sql END DECLARE SECTION;

    exec sql DECLARE MYCURSOR CURSOR FOR
             SELECT TABLE_SCHEMA,TABLE_NAME,TABLE_TYPE
         FROM INFORMATION_SCHEMA.TABLES;

    exec sql WHENEVER SQLERROR goto error_exit;

    exec sql CONNECT TO '' USER 'SYSADM' USING 'SYSADM'; /* Connect to DEFAULT */

    exec sql OPEN MYCURSOR;

    while (1) {
                exec sql FETCH MYCURSOR INTO :schema,:table,:type;
                if (strcmp(SQLSTATE, "02000") == 0) break; /* No more rows */
                printf("%s %s %s\n", schema, table, type);
    }

    exec sql CLOSE MYCURSOR;
    exec sql COMMIT;

    exec sql DISCONNECT DEFAULT;
    exit(0);

error_exit:
    print_sqlerror();

    exec sql WHENEVER SQLERROR CONTINUE;

    exec sql ROLLBACK;
    exec sql DISCONNECT DEFAULT;
    exit(0);
    return 0;
}
```

D.1.3 COBOL source code for the EXAMPLE program

```
 IDENTIFICATION DIVISION.
 PROGRAM-ID. EXAMPLE.
*
**  Example of X/Open CAE SQL (1992) compliant embedded program.
**
**  The program tries to connect to the DEFAULT database as SYSADM
**  with password SYSADM.
**  It will then declare a cursor for the standard view
**  INFORMATION_SCHEMA.TABLES, and fetch all records from it.
*

 DATA DIVISION.
 WORKING-STORAGE SECTION.
     EXEC SQL BEGIN DECLARE SECTION END-EXEC.
 01 SQLSTATE PICTURE X(5).
 01 TABLE-SCHEMA PICTURE X(128).
 01 TABLE-NAME PICTURE X(128).
 01 TABLE-TYPE PICTURE X(20).

 01 EXCEPTIONS PIC S9(10) COMP.
 01 ERROR-MESSAGE PIC X(254).
 01 LINE-NUMBER PIC S9(10) COMP.
     EXEC SQL END DECLARE SECTION END-EXEC.


 PROCEDURE DIVISION.
 MAIN SECTION.

 DO-IT.
     EXEC SQL DECLARE MYCURSOR CURSOR FOR
         SELECT TABLE_SCHEMA,TABLE_NAME,TABLE_TYPE
         FROM INFORMATION_SCHEMA.TABLES END-EXEC.

     EXEC SQL WHENEVER SQLERROR GO TO ERROR-EXIT END-EXEC.

**   CONNECT TO DEFAULT DATABASE
     EXEC SQL CONNECT TO '' USER 'SYSADM' USING 'SYSADM' END-EXEC.

     EXEC SQL OPEN MYCURSOR END-EXEC.

     PERFORM FETCH-AND-DISPLAY UNTIL SQLSTATE = "02000".

     EXEC SQL CLOSE MYCURSOR END-EXEC.
     EXEC SQL COMMIT END-EXEC.
     EXEC SQL DISCONNECT DEFAULT END-EXEC.
     STOP RUN.
```

```
FETCH-AND-DISPLAY.
    EXEC SQL FETCH MYCURSOR INTO :TABLE-SCHEMA,
                                 :TABLE-NAME,
                                 :TABLE-TYPE     END-EXEC.

    IF SQLSTATE NOT = "02000",
        DISPLAY TABLE-SCHEMA, TABLE-NAME, TABLE-TYPE.


ERROR-EXIT.
    EXEC SQL ROLLBACK END-EXEC.
    EXEC SQL WHENEVER SQLERROR CONTINUE  END-EXEC.
    EXEC SQL GET DIAGNOSTICS :EXCEPTIONS = NUMBER END-EXEC.
    PERFORM DISPLAY-ERROR-LINE VARYING LINE-NUMBER
              FROM 1 BY 1
              UNTIL LINE-NUMBER IS GREATER THAN EXCEPTIONS.
    EXEC SQL DISCONNECT DEFAULT END-EXEC.
    STOP RUN.

DISPLAY-ERROR-LINE.
    EXEC SQL GET DIAGNOSTICS EXCEPTION :LINE-NUMBER
                    :ERROR-MESSAGE = MESSAGE_TEXT
    END-EXEC.

    DISPLAY ERROR-MESSAGE.

END PROGRAM EXAMPLE.
```

D.1.4 FORTRAN source code for the EXAMPLE program

```
      PROGRAM SQLEX
C
C  Example of SQL-92 compliant embedded FORTRAN program.
C
C  The program tries to connect to the DEFAULT database as SYSADM
C  with  password SYSADM.
C  It will then create a cursor to the standard view
C  INFORMATION_SCHEMA.TABLES, and fetch all records from it.
C
C  If any error occurs, the PSQLER routine will be called to
C  print an error message.
C
      EXEC SQL BEGIN DECLARE SECTION
      CHARACTER*5  SQLSTATE
      CHARACTER*128 SCHEMA
      CHARACTER*128 TABLE
      CHARACTER*20 TYPE
      EXEC SQL END DECLARE SECTION

      EXEC SQL DECLARE MYCURSOR CURSOR FOR
     +     SELECT TABLE_SCHEMA,TABLE_NAME,TABLE_TYPE
     +     FROM INFORMATION_SCHEMA.TABLES

      EXEC SQL WHENEVER SQLERROR GOTO 9000

C     Connect to the default database
      EXEC SQL CONNECT TO '' USER 'SYSADM' USING 'SYSADM'

      EXEC SQL OPEN MYCURSOR

      EXEC SQL FETCH MYCURSOR INTO :SCHEMA,:TABLE,:TYPE
      DO WHILE (SQLSTATE .NE. '02000')
         WRITE(6,100) SCHEMA,TABLE,TYPE
 100     FORMAT (X,A,X,A,X,A)
         EXEC SQL FETCH MYCURSOR INTO :SCHEMA,:TABLE,:TYPE
      END DO

      EXEC SQL CLOSE MYCURSOR

      GOTO 9999

 9000 CONTINUE
      CALL PSQLER

 9999 CONTINUE
      EXEC SQL WHENEVER SQLERROR CONTINUE
      EXEC SQL COMMIT
      EXEC SQL DISCONNECT DEFAULT
      END
```

```
      SUBROUTINE PSQLER
C
C  PSQLER prints an error message for the latest error.
C  Programmed according to the X/Open CAE specification
C

      EXEC SQL BEGIN DECLARE SECTION
      CHARACTER*5  SQLSTATE
      INTEGER I,ERRORS,MSGLEN
      CHARACTER*254 MESSAGE
      EXEC SQL END DECLARE SECTION

      EXEC SQL GET DIAGNOSTICS :ERRORS = NUMBER
      DO 10 I=1,ERRORS
         EXEC SQL GET DIAGNOSTICS EXCEPTION :I
     +         :MESSAGE = MESSAGE_TEXT,
     +         :MSGLEN  = MESSAGE_LENGTH
         WRITE (6,100) MESSAGE(:MSGLEN)
 100     FORMAT(X,A)
 10   CONTINUE

      RETURN
      END
```

D.2 DSQLSAMP program using dynamic SQL

The DSQLSAMP program demonstrates how a C program can be constructed using dynamic SQL syntax according to international standards.

The source code file for the DSQLSAMP program is dsql.ec which contains a collection of routines that define a simple but convenient API for dynamic SQL, which can be called from other C programs.

The dsqlsamp.c file contains a program that calls the routines defined in dsql.ec. The DSQLSAMP program allows the user to enter an SQL statement that will be executed directly, in a manner similar to the BSQL program (see the Mimer SQL User's Manual for details about BSQL).

D.2.1 Building the DSQLSAMP program

Unix

The following example assumes Mimer SQL is installed in the directory /opt/mimer821A and illustrates how the DSQLSAMP program is compiled and built by using the distributed example makefile:

```
$ mkdir dsql        # Do everything in a sub directory
$ cd dsql
$ cp /opt/mimer821A/examples/ex_makefile ./makefile
$ cp /opt/mimer821A/examples/dsql* .
```

Take a copy of the makefile and update it so that the MYPROG symbol is set to "dsqlsamp" and the MYFUNCS symbol is set to "dsql.o" (avoid trailing spaces).

```
MYPROG = dsqlsamp
MYFUNCS = dsql.o
```

Then make the dsqlsamp program:

```
$ export MIMER_HOME=/opt/mimer821A
$ make
```

VMS

The following example illustrates how the DSQLSAMP program is compiled and linked under VMS. The program is supplied in C.

```
$ ESQL/C MIMEXAMPLES8:DSQL                ! Preprocess source code
$ COPY MIMEXAMPLES8:DSQL.H SYS$DISK:[]    ! Get header file
$ COPY MIMEXAMPLES8:DSQLSAMP.C SYS$DISK:[] ! Get main program
$ CC DSQL                                 ! Compile preprocessed code
$ CC DSQLSAMP                             ! Compile sample program
$ LINK DSQLSAMP,DSQL,MIMLIB8:MIMER/OPT    ! Link executable program
```

Win

On Windows platforms, the examples files are installed in the **dev** directory which is located below the installation directory. The dev directory contains a makefile called **makefile.mak**, which will build all the example programs at the same time. The examples have been tested using Microsoft Visual C.

These instructions assume that Microsoft Visual C is used and that the make command is being executed from the dev directory, appropriate adjustments must be made if this is not the case.

```
$ nmake -f makefile.mak
```

D.2.2 Source code for the DSQLSAMP program (in C)

```
/*************************************************************************/
/*  dsqlsamp.c - sample program using dynamic SQL driver                 */
/*************************************************************************/
/*                                                                       */
/*               Simple command line oriented SQL executor               */
/*                                                                       */
/*************************************************************************/
/*                                                                       */
/*  Created by Mimer Information Technology AB                           */
/*                                                                       */
/* You have a free right to use, modify, reproduce, and distribute the   */
/* sample files (and/or any modified version) in any way you find        */
/* useful, provided that you agree that Mimer Information Technology AB  */
/*  has no warranty obligations or liability for any sample files which  */
/*  are modified.                                                        */
/*                                                                       */
/*************************************************************************/

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "dsql.h"       /* function prototype definitions */

static   char  sqlbuf [4096];

static   void  error  ();
static   void  getid  (char* prompt,char* text);
static   char* savres (char* result,char* sqlbuf);
static   int*  savlen (int* collen,int colcount,char* sqlbuf);
static   void  show   (char* result,int rowcount,int* collen,int colcount);

/*************************************************************************/
int main()
/*************************************************************************/
{
    char  database [129];
    char  username [129];
    char  password [19];
    char* p;
    char* result;
    int*  collen;
    int   rc;
    int   cursor;
    int   colcount;
    int   rowcount;

    printf("\n     *** Welcome to Mimer Dynamic SQL Sample Program ***\n\n");
    /*
     *  Connect to database
     */
    for (;;)
    {
        getid("Database:",database);
        getid("Username:",username);
        getid("Password:",password); /* NOTE: password is not hidden! */
        printf(" \n");
        if ((rc = DsqlConnect(database,username,password)) > 0) break;
        if (rc < 0) error(), exit(0);
        printf("You have entered wrong username or password.\n");
        printf("Please, try again.\n\n");
    }
    printf(" - End an SQL statement with ';'\n");
    printf(" - Exit the program by giving an empty command\n\n");
```

```
/*
 *  SQL command loop
 */
for (;;)
{
    /*
     *  Read an SQL statement ending with ';'
     */
    printf("SQL>");     /* Display command prompter   */
    for (p = sqlbuf;; p++, *p++ = ' ')
    {
        *p = '\0';
        fgets(p,255,stdin);
        p += strlen(p);
        while ((*p == '\0' || *p == ' ') && p > sqlbuf) p--;
        if (*sqlbuf == '\0' || *p == ';') break;
        printf("   >");
    }
    printf(" \n");
    if (*p == '\0')     /* Empty command, ask if exit */
    {
        printf("Quit? (y/n) ");
        fgets(p,255,stdin);
        printf(" \n");
        if (*p == 'y' || *p == 'Y') break;
        continue;
    }
    *p = '\0';          /* Eliminate the ending ';'   */
    /*
     *  Execute SQL-statement
     */
    cursor = DsqlExecute(sqlbuf);
    if (cursor > 0)
    {
        /*
         *  It was a select statement, get column names
         */
        if ((colcount = DsqlColNames(cursor,sqlbuf,sizeof(sqlbuf))) > 0)
        {
            result = savres(NULL,sqlbuf);
            collen = savlen(NULL,colcount,sqlbuf);
            /*
             *  Fetch the resulting rows into memory
             *  and calculate the minimum possible width of the columns
             */
            rowcount = 0;
            while ((rc = DsqlFetch(cursor,sqlbuf,sizeof(sqlbuf))) > 0)
            {
                result = savres(result,sqlbuf);
                collen = savlen(collen,colcount,sqlbuf);
                rowcount++;
            }
            if (rc < 0 || DsqlClose(cursor) < 0) error();
            /*
             *  Show the result set nicely formatted
             */
            show(result,rowcount,collen,colcount);
        }
        else
        {
            error();
        }
    }
```

```
            else if (cursor == 0)
            {
                printf("*** SQL statement executed successfully! ***\n\n");
            }
            else
            {
                error();
            }
        }
        /*
         *  Disconnect and exit
         */
        if (DsqlDisconnect() < 0) error();
        printf("     *** Exit from Mimer Dynamic SQL Sample Program ***\n\n");
        exit(0);
        return 0;
}

/*************************************************************************/
/*                                                                       */
/*  Print SQL error message                                              */
/*                                                                       */
/*************************************************************************/
static void error()
/*************************************************************************/
{
    DsqlError(sqlbuf,sizeof(sqlbuf));
    printf("%s\n",sqlbuf);
}

/*************************************************************************/
/*                                                                       */
/*  Prompt for connect identifiers                                       */
/*                                                                       */
/*************************************************************************/
static void getid(char* prompt,char* text)
/*************************************************************************/
{
    printf("%s",prompt);
    sqlbuf[0] = '\0';
    fgets(sqlbuf,255,stdin);
    sqlbuf[128] = '\0';
    strcpy(text,sqlbuf);
}
```

```
/************************************************************************/
/*                                                                      */
/*  Save a result row in memory                                         */
/*                                                                      */
/************************************************************************/
static char* savres(char* result,char* sqlbuf)
/************************************************************************/
{
    static int reslen;
           int buflen = strlen(sqlbuf) + 1;

    if (result == NULL)
    {
        result = malloc(buflen + buflen - 1);
        memcpy(result,sqlbuf,buflen);
        reslen = buflen - 1;
    }
    else if((result = realloc(result,reslen + buflen)) == NULL)
    {
      printf("** Fatal: not enough memory, exiting... **\n\n");
      exit(0);
    }
    memcpy(&result[reslen],sqlbuf,buflen);
    reslen += buflen - 1;
    return result;
}

/************************************************************************/
/*                                                                      */
/*  Keep track of the minimum possible width of the columns             */
/*                                                                      */
/************************************************************************/
static int* savlen(int* collen,int colcount,char* sqlbuf)
/************************************************************************/
{
    char* p = strtok(sqlbuf,"\t\n");
    int   i,n;

    if (collen == NULL) collen = calloc(sizeof(int),colcount);
    for (i = 0; i < colcount; i++)
    {
        n = p ? strlen(p) : 0;
        if (collen[i] < n) collen[i] = n;
        p = strtok(NULL,"\t\n");
    }
    return collen;
}
```

```
/************************************************************************/
/*                                                                      */
/*  Print the formatted result set and release memory                   */
/*                                                                      */
/************************************************************************/
static void show(char* result, int rowcount, int* collen, int colcount)
/************************************************************************/
{
    char* p = strtok(result,"\t\n");
    int   i,n,pos;

    printf("*** %d row%s selected ***\n\n", rowcount, rowcount == 1 ? "" : "s");
    for (n = 0; n < rowcount + 2; n++)
    {
        pos = 0;
        memset(sqlbuf,' ',sizeof(sqlbuf));
        for (i = 0; i < colcount; i++)
        {
            if (p) memcpy(&sqlbuf[pos],p,strlen(p));
            if (n == 1) memset(&sqlbuf[pos],'=',collen[i]);
            p = strtok(NULL,"\t\n");
            pos += collen[i] + 1;
        }
        sqlbuf[pos] = '\0';
        printf("%s\n",sqlbuf);
    }
    printf("\n");
    free(result);
    free(collen);
}
```

```
/**********/
/* dsql.h */
/**********/

#ifndef DSQL_H
#define DSQL_H

int DsqlConnect    (char* database,char* userid,char* passwd);
int DsqlDisconnect (void);
int DsqlExecute    (char* sql);
int DsqlColNames   (int stmt,char* buffer,int len);
int DsqlFetch      (int stmt,char* buffer,int len);
int DsqlClose      (int stmt);
int DsqlError      (char* message,int len);

#endif
```

```
/************************************************************************/
/*  dsql.ec - dynamic SQL driver                                        */
/************************************************************************/
/*                                                                      */
/*  This package is a dynamic SQL sample with the following functions:  */
/*                                                                      */
/*      DsqlConnect    - connect to the database                        */
/*      DsqlDisconnect - disconnect from the database                   */
/*      DsqlExecute    - execute an SQL statement                       */
/*      DsqlColNames   - get the column names of a result table         */
/*      DsqlFetch      - fetch the next row from a result table         */
/*      DsqlClose      - close a cursor                                 */
/*      DsqlError      - get error message text                         */
/*                                                                      */
/************************************************************************/
/*                                                                      */
/*  Created by Mimer Information Technology AB.                         */
/*                                                                      */
/* You have a free right to use, modify, reproduce, and distribute the  */
/* sample files (and/or any modified version) in any way you find       */
/* useful, provided that you agree that Mimer Information Technology AB */
/* has no warranty obligations or liability for any sample files which  */
/* are modified.                                                        */
/*                                                                      */
/************************************************************************/

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <ctype.h>
#include "dsql.h"       /* function prototype definitions */

#define TRUE  1
#define FALSE 0

/*
 *  VARCHAR data type definition
 */
#define SQL_VARCHAR 12
/*
 *  SQLSTATE return code definitions
 */
#define SQL_SUCCESS                             "00000"
#define SQL_INSUFFICIENT_ITEM_DESCRIPTOR_AREAS "01005"
#define SQL_NO_DATA                             "02000"
#define SQL_CONNECTION_REJECTED                 "28000"
/*
 *  Shared data areas
 */
exec sql begin declare section;
static char statement  [129];       /* Extended statement name */
static char cursor     [129];       /* Extended cursor name    */
static char descriptor [129];       /* SQL descriptor name     */
static char sqlstate   [6];         /* SQLSTATE                */
static char buffer     [8192];      /* Data buffer             */
exec sql end declare section;

static int  seqno = 0;     /* Sequence number for generating unique names */
```

```
/*************************************************************************/
/*                                                                       */
/*  Connect to the database.                                             */
/*  Parameters:                                                          */
/*                                                                       */
/*  Return codes:                                                        */
/*                                                                       */
/*      1 : successful connection                                        */
/*      0 : wrong username or password                                   */
/*     -1 : error (call DsqlError for message text)                      */
/*************************************************************************/
int DsqlConnect()
/*************************************************************************/
{
    exec sql whenever sqlerror goto exception;
    exec sql begin declare section;
    char  database [129] = "";
    char  username [129] = "";
    char  password [19] = "";
    exec sql end declare section;
    char* p;

    getid("Database:",database);
    getid("Username:",username);
    getid("Password:",password);  /* NOTE: password is not hidden! */

    /*
     *  Connect to database.
     */
    for (p = username; *p; p++) *p = (char)toupper((int)*p);

    exec sql connect to :database user :username using :password;

    return 1;

exception:

    if (strcmp(sqlstate, SQL_CONNECTION_REJECTED) == 0)
    {
        return 0;
    }
    return -1;
}
```

```
/*************************************************************************/
/*                                                                       */
/*  Disconnect from the database.                                        */
/*                                                                       */
/*  Return codes:                                                        */
/*                                                                       */
/*      0 : OK                                                           */
/*     -1 : error (call DsqlError for message text)                      */
/*                                                                       */
/*************************************************************************/
int DsqlDisconnect()
/*************************************************************************/
{
    exec sql whenever sqlerror goto exception;

    exec sql disconnect;
    return 0;

exception:

    return -1;
}

/*************************************************************************/
/*                                                                       */
/*  Execute an SQL statement.                                            */
/*                                                                       */
/*  Parameter:                                                           */
/*                                                                       */
/*      char* sqlstmt : SQL statement to be executed                     */
/*                                                                       */
/*  Return codes:                                                        */
/*                                                                       */
/*     +n : a select statement successfully opened,                      */
/*          returning a cursor identifier                                */
/*      0 : a non-select statement executed successfully                 */
/*     -1 : error (call DsqlError for message text)                      */
/*                                                                       */
/*  Note:                                                                */
/*                                                                       */
/*      It is assumed that the SQL statement does not contain any        */
/*      parameter markers. Unique statement, descriptor, and cursor names */
/*      are used, which make it possible to have several cursors open at  */
/*      the same time.                                                   */
/*                                                                       */
/*************************************************************************/
int DsqlExecute(char* sqlstmt)
/*************************************************************************/
{
    exec sql whenever sqlerror goto exception;
    exec sql begin declare section;
    int  count;
    int  n;
    int  type;
    int  length;
    exec sql end declare section;

    /*
     *  Check argument.
     */
    if (sqlstmt == NULL) return -1;
    /*
     *  Make unique statement and descriptor names.
     */
    seqno++;
```

```
sprintf(statement, "%d",seqno);
sprintf(descriptor,"%d",seqno);
/*
 *  Copy SQL statement to buffer.
 */
strncpy(buffer,sqlstmt,sizeof(buffer));
buffer[sizeof(buffer) - 1] = '\0';
/*
 *  Prepare SQL statement.
 */
exec sql prepare :statement from :buffer;
/*
 *  Allocate descriptor.
 */
exec sql allocate descriptor :descriptor;
/*
 *  Describe output.
 */
exec sql describe output :statement
         using sql descriptor :descriptor;
if (strcmp(sqlstate, SQL_INSUFFICIENT_ITEM_DESCRIPTOR_AREAS) == 0)
{
    /*
     *  The descriptor area was insufficient.
     *  Allocate a new one with appropriate size.
     */
    exec sql get descriptor :descriptor :count = count;
    exec sql deallocate descriptor :descriptor;
    exec sql allocate descriptor :descriptor
             with max :count;
    exec sql describe output :statement
             using sql descriptor :descriptor;
}
/*
 *  Get descriptor count to see if this was a select statement or not.
 */
exec sql get descriptor :descriptor :count = count;
if (count == 0)
{
    /*
     *  Non-select statement. Execute.
     */
    exec sql execute :statement;
    /*
     *  Deallocate statement and descriptor.
     */
    exec sql deallocate prepare :statement;
    exec sql deallocate descriptor :descriptor;
    /*
     *  Return successful completion of non-select statement.
     */
    return 0;
}
else
{
    /*
     *  Select statement. Set all columns to VARCHAR(512).
     *                    Mimer SQL automatic type conversion will
     *                    handle numeric data.
     */
    type   = SQL_VARCHAR;
    length = 512;
```

```
            for (n = 1; n <= count; n++)
            {
                exec sql set descriptor :descriptor value :n type   = :type,
                                                           length = :length;
            }
            /*
             *  Make a unique cursor name.
             */
            sprintf(cursor,"%d",seqno);
            /*
             *  Allocate and open cursor.
             */
            exec sql allocate :cursor cursor for :statement;
            exec sql open :cursor;
            /*
             *  Return cursor identifier.
             */
            return seqno;
        }

exception:
    return -1;
}
```

```
/************************************************************************/
/*                                                                      */
/*  Get the column names of a result table.                             */
/*                                                                      */
/*  Parameters:                                                         */
/*                                                                      */
/*      int   c      : cursor identifier returned from DsqlExecute      */
/*      char* record : buffer for the result string                     */
/*      int   len    : length of the buffer (must be >= 32)             */
/*                                                                      */
/*  Return codes:                                                       */
/*                                                                      */
/*     +n : OK - number of columns                                      */
/*     -1 : error (call DsqlError for message text)                     */
/*                                                                      */
/*  Note:                                                               */
/*                                                                      */
/*      The column names are returned in a tab ('\t') separated string  */
/*      ending with a newline ('\n'). If the string is too large, it is */
/*      truncated.                                                      */
/*                                                                      */
/************************************************************************/
int DsqlColNames(int c,char* record,int len)
/************************************************************************/
{
    exec sql whenever sqlerror goto exception;
    exec sql begin declare section;
    int  count;
    int  n;
    char name [129];
    exec sql end declare section;
    int  length;

    /*
     *  Check arguments.
     */
    if (record == NULL || len < 32) return -1;
    /*
     *  Build descriptor name.
     */
    sprintf(descriptor,"%d",c);
    /*
     *  Get number of columns.
     */
    exec sql get descriptor :descriptor :count = count;
    for (n = 1; n <= count; n++)
    {
        /*
         *  Get one column name.
         */
        exec sql get descriptor :descriptor value :n
                 :name = name;
        /*
         *  Trim spaces and append a tab.
         */
        length = strlen(name);
        while (length > 1 && name[length - 1] == ' ') length--;
        if (length > len - 2) length = len - 2;
        memcpy(record,name,length);
        record += length;
        len    -= length;
```

```
            if (len == 2) break;
            if (n < count)
            {
                *record++ = '\t';
                len--;
            }
        }
        *record++ = '\n';
        *record   = '\0';
        return count;

exception:

        return -1;
}
```

```
/************************************************************************/
/*                                                                      */
/*  Fetch the next row from a result table.                             */
/*                                                                      */
/*  Parameters:                                                         */
/*                                                                      */
/*      int   c      : cursor identifier returned from DsqlExecute      */
/*      char* record : buffer for the result string                     */
/*      int   len    : length of the buffer (must be >= 32)             */
/*                                                                      */
/*  Return codes:                                                       */
/*                                                                      */
/*     +n : OK - number of columns                                      */
/*      0 : No more data (End of File)                                  */
/*     -1 : error (call DsqlError for message text)                     */
/*                                                                      */
/*  Note:                                                               */
/*                                                                      */
/*      The column values are returned in a tab ('\t') separated string */
/*      ending with a newline ('\n'). It is assumed that the data does  */
/*      not contain tab, newline or null characters. A column value of  */
/*      NULL will be return as '?'. If the string is too large, it is   */
/*      truncated.                                                      */
/*                                                                      */
/************************************************************************/
int DsqlFetch(int c,char* record,int len)
/************************************************************************/
{
    exec sql whenever sqlerror goto exception;
    exec sql begin declare section;
    int  count;
    int  length;
    int  isnull;
    int  n;
    exec sql end declare section;

    /*
     *  Check arguments.
     */
    if (record == NULL || len < 32) return -1;
    /*
     *  Build cursor and descriptor names.
     */
    sprintf(cursor,    "%d",c);
    sprintf(descriptor,"%d",c);
    /*
     *  Fetch next row from result table.
     */
    exec sql fetch next from :cursor
             into sql descriptor :descriptor;
```

```
    if (strcmp(sqlstate,SQL_SUCCESS) == 0)
    {
        /*
         *  Get number of columns.
         */
        exec sql get descriptor :descriptor :count = count;
        for (n = 1; n <= count; n++)
        {
            /*
             *  Get one column value.
             */
            exec sql get descriptor  :descriptor value :n :buffer = data,
                                     :length = returned_length,
                                     :isnull = indicator;
            /*
             *  Check NULL indicator, trim spaces, append tab.
             */
            if (isnull == -1)
            {
                *record++ = '?';
                 len--;
            }
            else
            {
                while (length > 1 && buffer[length - 1] == ' ') length--;
                if (length > len - 2) length = len - 2;
                memcpy(record,buffer,length);
                record += length;
                len    -= length;
            }
            if (len == 2) break;
            if (n < count)
            {
                *record++ = '\t';
                len--;
            }
        }
        *record++ = '\n';
        *record   = '\0';
        return count;
    }
    else if (strcmp(sqlstate,SQL_NO_DATA) == 0)
    {
        /*
         *  No more data.
         */
        return 0;
    }

exception:

    return -1;
}
```

```
/*************************************************************************/
/*                                                                       */
/*  Close a cursor.                                                      */
/*                                                                       */
/*  Parameter:                                                           */
/*                                                                       */
/*      int c : cursor identifier returned from DsqlExecute              */
/*                                                                       */
/*  Return codes:                                                        */
/*                                                                       */
/*      0 : OK                                                           */
/*     -1 : error (call DsqlError for message text)                      */
/*                                                                       */
/*************************************************************************/
int DsqlClose(int c)
/*************************************************************************/
{
    exec sql whenever sqlerror goto exception;

    /*
     *  Build cursor, statement and descriptor names.
     */
    sprintf(cursor,    "%d",c);
    sprintf(statement, "%d",c);
    sprintf(descriptor,"%d",c);
    /*
     *  Close cursor.
     */
    exec sql close :cursor;
    /*
     *  Deallocate statement and descriptor.
     */
    exec sql deallocate prepare :statement;
    exec sql deallocate descriptor :descriptor;
    return 0;

exception:

    return -1;
}
```

```
/************************************************************************/
/*                                                                      */
/*  Get error message text.                                             */
/*                                                                      */
/*  Parameters:                                                         */
/*                                                                      */
/*      char* message : buffer for the result string                    */
/*      int   len     : length of the buffer (must be >= 32)            */
/*                                                                      */
/*  Return codes:                                                       */
/*                                                                      */
/*      0 : OK                                                          */
/*     -1 : error (cannot get message text)                             */
/*                                                                      */
/*  Note:                                                               */
/*                                                                      */
/*      The message text is returned in a newline ('\n') separated      */
/*      string. If the string is too large, it is truncated.            */
/*                                                                      */
/************************************************************************/
int DsqlError(char* message,int len)
/************************************************************************/
{
    exec sql whenever sqlerror goto exception;
    exec sql begin declare section;
    int   count;
    int   length;
    char  state [6]; /* returned_sqlstate may not be stored sqlstate */
    int   n;
    exec sql end declare section;
    char* p;

    /*
     *  Check arguments.
     */
    if (message == NULL || len < 32) return -1;
    /*
     *  Get number of errors.
     */
    exec sql get diagnostics :count = number;
    for (n = 1; n <= count; n++)
    {
        /*
         *  Get diagnostics of an error.
         */
        exec sql get diagnostics exception :n :buffer = message_text,
                                              :length = message_length,
                                              :state  = returned_sqlstate;
        /*
         *  Prepare message area.
         */
        if (length > len - 18) length = len - 18;
        memcpy(message,buffer,length);
        p = strchr(message,':');
        if (p) *p = '\n';
        if (length > 64)
        {
            p = strchr(&message[64],' ');
            if (p) *p = '\n';
        }
```

```
            message += length;
            len     -= length;
            *message++ = '\n';
            /*
             *  Add SQLSTATE code.
             */
            strcpy(message,"SQLSTATE:");
            message += 9;
            strcpy(message,state);
            message += 5;
            *message++ = '\n';
            *message++ = '\n';
            len -= 17;
            if (len < 32) break;
        }
        *message = '\0';
        return 0;

    exception:

        strcpy(message,"cannot get diagnostics\n\n");
        return -1;
    }
```

D.3 FREQCALL program using dynamic SQL and a stored procedure

The FREQCALL source code demonstrates how to embed a call to a stored procedure. The procedure used is included in the example database (described in Section 3.9 of the Mimer SQL System Management Handbook), which must be installed in order to execute the programs successfully.

D.3.1 Building the FREQCALL program

Unix

The following example assumes Mimer SQL is installed in the directory /opt/mimer821A and illustrates how the C version of the FREQCALL program is compiled and built by using the distributed example makefile:

```
$ mkdir freqcall          # Do everything in a sub directory
$ cd freqcall
$ cp /opt/mimer821A/examples/ex_makefile ./makefile
$ cp /opt/mimer821A/examples/freqcall.ec .
```

Take a copy of the makefile and update it so that the MYPROG symbol is set to "freqcall", as the name of the program to be created (avoid trailing spaces).

```
MYPROG = freqcall
```

Then make the freqcall program:

```
$ export MIMER_HOME=/opt/mimer821A
$ make
.
.
```

VMS

The following example illustrates how the C version of the FREQCALL program is compiled and linked under VMS. The programs written in the other languages can be linked in a similar manner.

```
$ ESQL/C MIMEXAMPLES8:FREQCALL          ! Preprocess source code
$ CC FREQCALL                           ! Compile sample program
$ LINK FREQCALL,MIMLIB8:MIMER/OPT       ! Link executable program
```

Win

On Windows platforms, the examples files are installed in the **dev** directory which is located below the installation directory. The dev directory contains a makefile called **makefile.mak**, which will build all the example programs at the same time. The examples have been tested using Microsoft Visual C.

These instructions assume that Microsoft Visual C is used and that the make command is being executed from the dev directory, appropriate adjustments must be made if this is not the case.

```
$ nmake -f makefile.mak
```

D.3.2 C source code for the FREQCALL program

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

/*
**  Example of Embedded Stored Procedure Calls.
**
**  This program calls the procedure freeq, to find out the number of
**  vacant rooms at a given hotel.
*/

exec sql BEGIN DECLARE SECTION;
static char sqlstate[6];
exec sql END DECLARE SECTION;

void print_sqlerror()
/*
**  print_sqlerror prints an error message for the latest error.
**  Programmed according to the X/Open CAE specification.
*/
{
    exec sql BEGIN DECLARE SECTION;
    int i;
    int exceptions;
    VARCHAR message[255];
    exec sql END DECLARE SECTION;

    exec sql GET DIAGNOSTICS :exceptions = NUMBER; /* How many exceptions? */
    for (i=1; i<=exceptions; i++) {
        exec sql GET DIAGNOSTICS EXCEPTION :i
                                           :message = MESSAGE_TEXT;
        printf("%s\n", message);
    }
}

main()
{
    exec sql BEGIN DECLARE SECTION;
    char query[200] = "call freeq(?,?,?,?,?)";
    char hotelcode[30];
    char roomtype[30];
    char arrive[30], depart[30];
    int  rooms;
    exec sql END DECLARE SECTION;

    exec sql WHENEVER SQLERROR goto error_exit;

    exec sql CONNECT TO ' ' USER 'HOTELADM' USING 'HOTELADM';

    /*
     * Make it easy for us. Ask for free rooms on the SKY hotel,
     * roomtype SSGLS during new years eve of the new millennium.
     */
    strcpy(hotelcode,"SKY");
    strcpy(roomtype,"SSGLS");
    strcpy(arrive,"1999-12-31");
    strcpy(depart,"2000-01-01");

    exec sql PREPARE CALL FROM :query;
    exec sql EXECUTE CALL INTO :rooms
                USING :hotelcode, :roomtype, :arrive, :depart;
    printf("%d available rooms.\n",rooms);
```

```
    exec sql commit;
    exec sql DISCONNECT;
    exit(0);

error_exit:
    print_sqlerror();

    exec sql WHENEVER SQLERROR CONTINUE;
    exec sql rollback;
    exec sql DISCONNECT;
    exit(0);
    return 0;
}
```

D.3.3 COBOL source code for the FREQCALL program

```
 IDENTIFICATION DIVISION.
 PROGRAM-ID. FREQCALL.
*
*  Example of Embedded Stored Procedure Calls.
*
*  This program calls the procedure freeq, to find out the number of
*  vacant rooms at a given hotel.
*

 DATA DIVISION.
 WORKING-STORAGE SECTION.
     EXEC SQL BEGIN DECLARE SECTION END-EXEC.
 01 SQLSTATE PICTURE X(5).
 01 QUERY PICTURE X(200).
 01 HOTELCODE PICTURE X(30).
 01 ROOMTYPE PICTURE X(30).
 01 ARRIVE PICTURE X(30).
 01 DEPART PICTURE X(30).
 01 ROOMS PIC S9(10) COMP.
 01 EXCEPTIONS PIC S9(10) COMP.
 01 LINE-NUMBER PIC S9(10) COMP.
 01 ERROR-MESSAGE PICTURE X(254).
     EXEC SQL END DECLARE SECTION END-EXEC.

 PROCEDURE DIVISION.
 BIG.

     EXEC SQL WHENEVER SQLERROR GO TO ERROR-EXIT END-EXEC.

     EXEC SQL CONNECT TO '' USER 'HOTELADM' USING 'HOTELADM'
         END-EXEC.

*
*  Make it easy for us. Ask for free rooms on the SKY hotel,
*  roomtype SSGLS during new years eve of the new millennium.
*

     MOVE 'SKY' TO HOTELCODE.
     MOVE 'SSGLS' TO ROOMTYPE.
     MOVE '1999-12-31' TO ARRIVE.
     MOVE '2000-01-01' TO DEPART.

     MOVE 'call freeq(?,?,?,?,?)' TO QUERY.

     EXEC SQL PREPARE CALL FROM :QUERY END-EXEC.
     EXEC SQL EXECUTE CALL INTO :ROOMS
                  USING :HOTELCODE, :ROOMTYPE,
                        :ARRIVE, :DEPART END-EXEC.

     DISPLAY ROOMS WITH CONVERSION " available rooms.".

     EXEC SQL COMMIT END-EXEC.
     EXEC SQL DISCONNECT DEFAULT END-EXEC.
     STOP RUN.
```

```
ERROR-EXIT.
    EXEC SQL WHENEVER SQLERROR CONTINUE  END-EXEC.
    EXEC SQL GET DIAGNOSTICS :EXCEPTIONS = NUMBER  END-EXEC.
    PERFORM DISPLAY-ERROR-LINE VARYING LINE-NUMBER
               FROM 1 BY 1
               UNTIL LINE-NUMBER IS GREATER THAN EXCEPTIONS.
    EXEC SQL DISCONNECT DEFAULT END-EXEC.
    STOP RUN.

DISPLAY-ERROR-LINE.
    EXEC SQL GET DIAGNOSTICS EXCEPTION :LINE-NUMBER
                 :ERROR-MESSAGE = MESSAGE_TEXT END-EXEC.
    DISPLAY ERROR-MESSAGE.

END PROGRAM FREQCALL.
```

D.3.4 FORTRAN source code for the FREQCALL program

```
      PROGRAM FREQCALL
C
C  Example of Embedded Stored Procedure Calls.
C
C  This program calls the procedure freeq, to find out the number of
C  vacant rooms at a given hotel.
C
      EXEC SQL BEGIN DECLARE SECTION
      CHARACTER*5   SQLSTATE
      CHARACTER*200 QUERY
      CHARACTER*30  HOTELCODE
      CHARACTER*30  ROOMTYPE
      CHARACTER*30  ARRIVE
      CHARACTER*30  DEPART
      INTEGER       ROOMS
      EXEC SQL END DECLARE SECTION

      EXEC SQL WHENEVER SQLERROR GOTO 9000

      EXEC SQL CONNECT TO '' USER 'HOTELADM' USING 'HOTELADM'

C
C  Make it easy for us. Ask for free rooms on the SKY hotel,
C  roomtype SSGLS during new years eve of the new millennium.
C

      HOTELCODE = 'SKY'
      ROOMTYPE = 'SSGLS'
      ARRIVE = '1999-12-31'
      DEPART = '2000-01-01'

      QUERY = 'call freeq(?,?,?,?,?)'

      EXEC SQL PREPARE CALL FROM :QUERY
      EXEC SQL EXECUTE CALL INTO :ROOMS
     +           USING :HOTELCODE, :ROOMTYPE, :ARRIVE, :DEPART

      WRITE(6,100) ROOMS,' available rooms.'
 100  FORMAT (I,A,X,A)

      GOTO 9999

 9000 CONTINUE
      CALL PRINT_SQLERROR

 9999 CONTINUE
      EXEC SQL WHENEVER SQLERROR CONTINUE
      EXEC SQL COMMIT
      EXEC SQL DISCONNECT DEFAULT
      END
```

```
      SUBROUTINE PRINT_SQLERROR
C
C  PRINT_SQLERROR prints an error message for the latest error.
C  Programmed according to the X/Open CAE specification.
C

      EXEC SQL BEGIN DECLARE SECTION
      CHARACTER*5   SQLSTATE
      INTEGER       I,ERRORS,MSGLEN
      CHARACTER*254 MESSAGE
      EXEC SQL END DECLARE SECTION

      EXEC SQL GET DIAGNOSTICS :ERRORS = NUMBER
      DO 10 I=1,ERRORS
         EXEC SQL GET DIAGNOSTICS EXCEPTION :I
     +         :MESSAGE = MESSAGE_TEXT,
     +         :MSGLEN  = MESSAGE_LENGTH
         WRITE (6,100) MESSAGE(:MSGLEN)
 100     FORMAT(X,A)
 10   CONTINUE

      RETURN
      END
```

D.4 WAKECALL program using dynamic SQL and a result-set procedure

The WAKECALL source code demonstrates how to embed a call to a result-set procedure. The procedure used is included in the example database (described in Section 3.9 of the Mimer SQL System Management Handbook), which must be installed in order to execute the programs successfully.

D.4.1 Building the WAKECALL program

Unix

The following example assumes Mimer SQL is installed in the directory /opt/mimer821A and illustrates how the C version of the WAKECALL program is compiled and built by using the distributed example makefile:

```
$ mkdir wakecall          # Do everything in a sub directory
$ cd wakecall
$ cp /opt/mimer821A/examples/ex_makefile ./makefile
$ cp /opt/mimer821A/examples/wakecall.ec .
```

Take a copy of the makefile and update it so that the MYPROG symbol is set to "wakecall", as the name of the program to be created (avoid trailing spaces).

```
MYPROG = wakecall
```

Then make the wakecall program:

```
$ export MIMER_HOME=/opt/mimer821A
$ make
.
.
```

VMS

The following example illustrates how the C version of the WAKECALL program is compiled and linked under VMS. The programs written in the other languages can be linked in a similar manner.

```
$ ESQL/C MIMEXAMPLES8:WAKECALL            ! Preprocess source code
$ CC WAKECALL                             ! Compile sample program
$ LINK WAKECALL,MIMLIB8:MIMER/OPT         ! Link executable program
```

Win

On Windows platforms, the examples files are installed in the **dev** directory which is located below the installation directory. The dev directory contains a makefile called **makefile.mak**, which will build all the example programs at the same time. The examples have been tested using Microsoft Visual C.

These instructions assume that Microsoft Visual C is used and that the make command is being executed from the dev directory, appropriate adjustments must be made if this is not the case.

```
$ nmake -f makefile.mak
```

D.4.2 C source code for the WAKECALL program

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

/*
**  Example of Embedded Stored Procedure Calls.
**
**  This program calls the procedure wake_up, which returns
**  a list of those rooms which have requested wake up calls within
**  the specified amount of time.
*/

exec sql BEGIN DECLARE SECTION;
static char sqlstate[6];
exec sql END DECLARE SECTION;

void print_sqlerror()
/*
**  print_sqlerror prints an error message for the latest error.
**  Programmed according to the X/Open CAE specification.
*/
{
    exec sql BEGIN DECLARE SECTION;
    int i;
    int exceptions;
    VARCHAR message[255];
    exec sql END DECLARE SECTION;

    exec sql GET DIAGNOSTICS :exceptions = NUMBER; /* How many exceptions? */
    for (i=1; i<=exceptions; i++) {
        exec sql GET DIAGNOSTICS EXCEPTION :i
                                           :message = MESSAGE_TEXT;
        printf("%s\n", message);
    }
}


main()
{
    exec sql BEGIN DECLARE SECTION;
    char roomnr[30];
    exec sql END DECLARE SECTION;

    exec sql DECLARE MYCURSOR CURSOR FOR CALL wake_up(interval '30' minute);

    exec sql WHENEVER SQLERROR goto error_exit;

    exec sql CONNECT TO ' ' USER 'HOTELADM' USING 'HOTELADM';

    exec sql OPEN MYCURSOR;

    while (1) {
        exec sql FETCH MYCURSOR INTO :roomnr;
        if (strcmp(sqlstate, "02000") == 0) break; /* No more rows */
        printf("Room %s\n", roomnr);
    }

    exec sql CLOSE MYCURSOR;

    exec sql commit;
    exec sql DISCONNECT DEFAULT;
    exit(0);
```

```
error_exit:
    print_sqlerror();

    exec sql WHENEVER SQLERROR CONTINUE;

    exec sql rollback;
    exec sql DISCONNECT;
    exit(0);
    return 0;
}
```

D.4.3 COBOL source code for the WAKECALL program

```
 IDENTIFICATION DIVISION.
 PROGRAM-ID. WAKECALL.
*
**  Example of Embedded Stored Procedure Calls.
**
**  This program calls the procedure wake_up, which returns
**  a list of those rooms which have requested wake up calls within
**  the specified amount of time.
*

 DATA DIVISION.
 WORKING-STORAGE SECTION.
     EXEC SQL BEGIN DECLARE SECTION END-EXEC.
 01 SQLSTATE PICTURE X(5).
 01 ROOMNR PICTURE X(30).

 01 EXCEPTIONS PIC S9(10) COMP.
 01 ERROR-MESSAGE PIC X(254).
 01 LINE-NUMBER PIC S9(10) COMP.
     EXEC SQL END DECLARE SECTION END-EXEC.


 PROCEDURE DIVISION.
 MAIN SECTION.

 DO-IT.
     EXEC SQL DECLARE MYCURSOR CURSOR FOR
              CALL wake_up(interval '30' minute) END-EXEC.

     EXEC SQL WHENEVER SQLERROR GO TO ERROR-EXIT END-EXEC.

     EXEC SQL CONNECT TO ''
              USER 'HOTELADM' USING 'HOTELADM' END-EXEC.

     EXEC SQL OPEN MYCURSOR END-EXEC.

     PERFORM FETCH-AND-DISPLAY UNTIL SQLSTATE = "02000".

     EXEC SQL CLOSE MYCURSOR END-EXEC.
     EXEC SQL COMMIT END-EXEC.
     EXEC SQL DISCONNECT DEFAULT END-EXEC.
     STOP RUN.


 FETCH-AND-DISPLAY.
     EXEC SQL FETCH MYCURSOR INTO :ROOMNR END-EXEC.

     IF  SQLSTATE NOT = "02000",
         DISPLAY "Room " ROOMNR.


 ERROR-EXIT.
     EXEC SQL WHENEVER SQLERROR CONTINUE  END-EXEC.
     EXEC SQL GET DIAGNOSTICS :EXCEPTIONS = NUMBER  END-EXEC.
     PERFORM DISPLAY-ERROR-LINE VARYING LINE-NUMBER
               FROM 1 BY 1
               UNTIL LINE-NUMBER IS GREATER THAN EXCEPTIONS.
     EXEC SQL ROLLBACK END-EXEC.
     EXEC SQL DISCONNECT DEFAULT END-EXEC.
     STOP RUN.

 DISPLAY-ERROR-LINE.
```

```
            EXEC SQL GET DIAGNOSTICS EXCEPTION :LINE-NUMBER
                           :ERROR-MESSAGE = MESSAGE_TEXT  END-EXEC.
            DISPLAY ERROR-MESSAGE.

        END PROGRAM WAKECALL.
```

D.4.4 FORTRAN source code for the WAKECALL program

```
      PROGRAM WAKECALL
C
C  Example of Embedded Stored Procedure Calls.
C
C  This program calls the procedure wake_up, which returns
C  a list of those rooms which have requested wake up calls within
C  the specified amount of time.
C
      EXEC SQL BEGIN DECLARE SECTION
      CHARACTER*5  SQLSTATE
      CHARACTER*30 ROOMNO
      EXEC SQL END DECLARE SECTION

      EXEC SQL DECLARE MYCURSOR CURSOR FOR
     +     CALL wake_up(interval '30' minute)

      EXEC SQL WHENEVER SQLERROR GOTO 9000

      EXEC SQL CONNECT TO '' USER 'HOTELADM' USING 'HOTELADM'

      EXEC SQL OPEN MYCURSOR

      EXEC SQL FETCH MYCURSOR INTO :ROOMNO
      DO WHILE (SQLSTATE .NE. '02000')
         WRITE(6,100) 'Room ', ROOMNO
 100     FORMAT (X,A,X,A)
         EXEC SQL FETCH MYCURSOR INTO :ROOMNO
      END DO

      EXEC SQL CLOSE MYCURSOR

      GOTO 9999

 9000 CONTINUE
      CALL PSQLER

 9999 CONTINUE
      EXEC SQL WHENEVER SQLERROR CONTINUE
      EXEC SQL COMMIT
      EXEC SQL DISCONNECT DEFAULT
      END


      SUBROUTINE PSQLER
C
C  PSQLER prints an error message for the latest error.
C  Programmed according to the X/Open CAE specification.
C

      EXEC SQL BEGIN DECLARE SECTION
      CHARACTER*5  SQLSTATE
      INTEGER I,ERRORS,MSGLEN
      CHARACTER*254 MESSAGE
      EXEC SQL END DECLARE SECTION
```

```
      EXEC SQL GET DIAGNOSTICS :ERRORS = NUMBER
      DO 10 I=1,ERRORS
         EXEC SQL GET DIAGNOSTICS EXCEPTION :I
     +         :MESSAGE = MESSAGE_TEXT,
     +         :MSGLEN  = MESSAGE_LENGTH
         WRITE (6,100) MESSAGE(:MSGLEN)
100      FORMAT(X,A)
10    CONTINUE

      RETURN
      END
```

D.5 BLOBSAMP program using Dynamic SQL for handling binary data

The BLOBSAMP program demonstrates the use of the VARCHAR data type to store BLOB's (**B**inary **L**arge **OB**jects). The program copies the contents of a file into the database as a BLOB, it then performs a SELECT to retrieve it again and compares the result with the original file. The username, password and file name should be specified as input parameters to the program and the program assumes that the ident in the database identified by *username* has access to a databank where a table can be created.

The BLOBSAMP program expects parameters to be specified on the command-line. If no parameters are the usage text below is printed:

```
Usage: blobsamp username password filename
```

D.5.1 Building the BLOBSAMP program

Unix

The following example assumes MIMER is installed in the directory /opt/mimer821A and illustrates how the BLOBSAMP program is compiled and built by using the distributed example makefile:

```
$ mkdir blobsamp          # Do everything in a sub directory
$ cd blobsamp
$ cp /opt/mimer821A/examples/ex_makefile ./makefile
$ cp /opt/mimer821A/examples/blobsamp.ec .
```

Take a copy of the makefile and update it so that the MYPROG symbol is set to "blobsamp", as the name of the program to be created (avoid trailing spaces).

```
MYPROG = blobsamp
```

Then make the blobsamp program:

```
$ export MIMER_HOME=/opt/mimer821A
$ make
.
.
```

VMS

The following example illustrates how the BLOBSAMP program is compiled and linked under VMS. The program is supplied in C.

```
$ ESQL/C MIMEXAMPLES8:BLOBSAMP            ! Preprocess source code
$ CC BLOBSAMP                             ! Compile sample program
$ LINK BLOBSAMP,MIMLIB8:MIMER/OPT         ! Link executable program
$ BLOBSAMP:=$disk:[current.dir]BLOBSAMP   ! Define BLOBSAMP so it
                                          ! can be run
```

Win

On Windows platforms, the examples files are installed in the **dev** directory which is located below the installation directory. The dev directory contains a makefile called **makefile.mak**, which will build all the example programs at the same time. The examples have been tested using Microsoft Visual C.

These instructions assume that Microsoft Visual C is used and that the make command is being executed from the dev directory, appropriate adjustments must be made if this is not the case.

```
$ nmake -f makefile.mak
```

D.5.2 Source code for the BLOBSAMP program (in C)

```
/************************************************************************/
/* blobsamp.ec - sample program for put/get binary data using ESQL      */
/************************************************************************/
/*                                                                      */
/*  Created by Mimer Information Technology AB.                         */
/*                                                                      */
/*  You have a free right to use, modify, reproduce, and distribute the */
/*  sample files (and/or any modified version) in any way you find      */
/*  useful, provided that you agree that Mimer Information Technology AB*/
/*  has no warranty obligations or liability for any sample files which */
/* are modified.                                                        */
/*                                                                      */
/************************************************************************/
/*                                                                      */
/*  This sample program just shows how to read a file and insert it into*/
/*  the table:                                                          */
/*                                                                      */
/*              BLOB(ID   int,                                          */
/*                   SEQ  int,                                          */
/*                   DATA varbinary(15000) not null,                    */
/*                   primary key(ID,SEQ))                               */
/*                                                                      */
/*  and then how to fetch the data from the table BLOB and compare it   */
/*  with the file.                                                      */
/*                                                                      */
/************************************************************************/

#include <stdlib.h>
#include <stdio.h>
#include <string.h>

#define SQL_NO_DATA "02000"

#define SQL_BINARY_VARCHAR -13  /* MIMER-specific type code */

exec sql begin declare section;
static char buffer[15001];      /* data buffer */
static char sqlstate[6];
static char sqlselect[35] = "select DATA from BLOB where ID = ?";
static char sqlinsert[31] = "insert into BLOB values(?,?,?)";
exec sql end declare section;

static int putblob (int,char*);
static int chkblob (int,char*);
static int error   (char*,int);

/************************************************************************/
int main(int argc,char** argv)
/************************************************************************/
{
    exec sql whenever sqlerror goto exception;
    exec sql begin declare section;
    char  uid[129];
    char  pwd[19];
    exec sql end declare section;
    char* filename;
    int   id = 1;

    if (argc != 4)
    {
        printf("\nUsage: blobsamp username password filename\n\n");
        return 0;
    }
```

```
    strcpy(uid,argv[1]);
    strcpy(pwd,argv[2]);
    filename = argv[3];

    exec sql connect to '' user :uid using :pwd;

    exec sql create table BLOB(ID   int,
                               SEQ  int,
                               DATA varbinary(15000) not null,
                               primary key(ID,SEQ));
    exec sql commit;

    if (putblob(id,filename) == 0) /* Store file contents in table BLOB   */
    {
        chkblob(id,filename);      /* Check that we can read the data     */
    }
    exec sql drop table BLOB;
    exec sql commit;

    exec sql disconnect;
    return 0;

exception:

    error(buffer,sizeof(buffer));
    printf("%s\n",buffer);
    exec sql disconnect;
    return 0;
}
```

```
/*************************************************************************/
/*                                                                       */
/*  Put the contents of a file into the database                         */
/*                                                                       */
/*  Parameters:                                                          */
/*                                                                       */
/*      int   blobid   : identifier for the data                         */
/*      char* filename : name of the file                                */
/*                                                                       */
/*  Return codes:                                                        */
/*                                                                       */
/*      0 : OK                                                           */
/*     -1 : error                                                        */
/*                                                                       */
/*************************************************************************/
static int putblob(int blobid,char* filename)
/*************************************************************************/
{
    exec  sql whenever sqlerror goto exception;
    exec  sql begin declare section;
    int   id;
    int   seq;
    int   type;
    int   length;
    exec  sql end declare section;
    FILE* file;

    if ((file = fopen(filename,"rb")) == NULL)
    {
        printf("open error\n");
        return -1;
    }
    id   = blobid;
    type = SQL_BINARY_VARCHAR;

    exec sql prepare statement from :sqlinsert;
    exec sql allocate descriptor 'input' with max 3;
    exec sql describe input statement using sql descriptor 'input';

    for (seq = 1; !feof(file); seq++)
    {
        length = fread(buffer,1,15000,file);
        if (ferror(file))
        {
            printf("read error\n");
            return -1;
        }

        exec sql set descriptor 'input' value 1 data   = :id;
        exec sql set descriptor 'input' value 2 data   = :seq;
        exec sql set descriptor 'input' value 3 data   = :buffer,
                                                type   = :type,
                                                length = :length;

        exec sql execute statement using sql descriptor 'input';
    }
    exec sql commit;
    exec sql deallocate descriptor 'input';
    exec sql deallocate prepare statement;
    fclose(file);
    return 0;

exception:

    error(buffer,sizeof(buffer));
```

```
    printf("%s\n",buffer);
    return -1;
}
```

```
/*************************************************************************/
/*                                                                       */
/*  Check that stored data are exactly the same as the file contents.    */
/*                                                                       */
/*  Parameters:                                                          */
/*                                                                       */
/*      int   blobid   : identifier for the data                         */
/*      char* filename : name of the file                                */
/*                                                                       */
/*  Return codes:                                                        */
/*                                                                       */
/*      0 : OK                                                           */
/*     -1 : error                                                        */
/*                                                                       */
/*************************************************************************/
static int chkblob(int blobid,char* filename)
/*************************************************************************/
{
    exec  sql whenever sqlerror goto exception;
    exec  sql begin declare section;
    int   id;
    int   type;
    int   length;
    exec  sql end declare section;
    FILE* file;
    char  record[15000];


    if ((file = fopen(filename,"rb")) == NULL)
    {
        printf("open error\n");
        return -1;
    }
    id   = blobid;
    type = SQL_BINARY_VARCHAR;

    exec sql prepare statement from :sqlselect;

    exec sql allocate descriptor 'input'  with max 1;
    exec sql allocate descriptor 'output' with max 1;

    exec sql describe input  statement using sql descriptor 'input';
    exec sql describe output statement using sql descriptor 'output';

    exec sql set descriptor 'input'  value 1 data = :id;

    exec sql declare c cursor for statement;
    exec sql open c using sql descriptor 'input';
    for (;;)
    {
        exec sql fetch next from c into sql descriptor 'output';

        if (strcmp(sqlstate,SQL_NO_DATA) == 0) break;

        exec sql get descriptor 'output' value 1 :buffer = data,
                                                 :type   = type,
                                                 :length = returned_length;

        if (fread(record,1,15000,file)   != (size_t)length ||
            memcmp(buffer,record,length) != 0)
        {
            printf("compare error\n");
            return -1;
        }
    }
```

```
    exec sql close c;
    exec sql deallocate descriptor 'output';
    exec sql deallocate descriptor 'input';
    exec sql deallocate prepare statement;
    fclose(file);
    printf("compare ok!\n");
    return 0;

exception:

    error(buffer,sizeof(buffer));
    printf("%s\n",buffer);
    return -1;
}
```

```
/************************************************************************/
/*                                                                      */
/*  Get error message text.                                             */
/*                                                                      */
/*  Parameters:                                                         */
/*                                                                      */
/*      char* message : buffer for the result string                    */
/*      int   len     : length of the buffer (must be >= 32)            */
/*                                                                      */
/*  Return codes:                                                       */
/*                                                                      */
/*      0 : OK                                                          */
/*     -1 : error (cannot get message text)                             */
/*                                                                      */
/*  Note:                                                               */
/*                                                                      */
/*      The message text is returned in a newline ('\n') separated      */
/*      string. If the string is too large, it is truncated.            */
/*                                                                      */
/************************************************************************/
static int error(char* message,int len)
/************************************************************************/
{
    exec sql whenever sqlerror goto exception;
    exec sql begin declare section;
    int   count;
    int   length;
    char  state [6]; /* returned_sqlstate may not be stored in sqlstate */
    int   n;
    exec sql end declare section;
    char* p;

    /*
     *  Check arguments.
     */
    if (message == NULL || len < 32) return -1;
    /*
     *  Get number of errors.
     */
    exec sql get diagnostics :count = number;
    for (n = 1; n <= count; n++)
    {
        /*
         *  Get diagnostics of an error.
         */
        exec sql get diagnostics exception :n :buffer = message_text,
                                              :length = message_length,
                                              :state  = returned_sqlstate;
        /*
         *  Prepare message area.
         */
        if (length > len - 18) length = len - 18;
        memcpy(message,buffer,length);
        p = strchr(message,':');
        if (p) *p = '\n';
        if (length > 78)
        {
            p = strchr(&message[78],' ');
            if (p) *p = '\n';
        }
        message += length;
        len     -= length;
        *message++ = '\n';
        /*
         *  Add SQLSTATE code.
```

```
            */
            strcpy(message,"SQLSTATE:");
            message += 9;
            strcpy(message,state);
            message += 5;
            *message++ = '\n';
            *message++ = '\n';
            len -= 17;
            if (len < 32) break;
        }
        *message = '\0';
        return 0;

exception:

        strcpy(message,"cannot get diagnostics\n\n");
        return -1;
}
```

D.6 OSQLSAMP program using dynamic SQL

To be able to compile the ODBC example program, an ODBC SDK is needed. This package should include the necessary header files, such as sqlext.h and sql.h, or corresponding.

The osql.c file contains a collection of routines that define a simple but convenient API for dynamic SQL, using ODBC according to international standards. These routines can be called from other C programs, in this case osqlsamp.c.

The osqlsamp.c (same as dsqlsamp.c - see Section D.2) file contains a program that calls the routines in osql.c. The osqlsamp program allows the user to enter a SQL statement that will be executed directly, similar to the BSQL program (see the *Mimer SQL User's Manual*).

D.6.1 Building the OSQLSAMP program

Unix

The following example assumes MIMER is installed in the directory /opt/mimer821A and illustrates how the OSQLSAMP program is compiled and built by using the distributed example makefile:

```
$ mkdir osql        # Do everything in a sub directory
$ cd osql
$ cp /opt/mimer821A/examples/ex_makefile ./makefile
$ cp /opt/mimer821A/examples/osql.c .
$ cp /opt/mimer821A/examples/dsql.h .
$ cp /opt/mimer821A/examples/dsqlsamp.c osqlsamp.c
```

In the following example the ODBC Driver Manager is presumed to be installed under **/usr/local**.

Update the following macros in the makefile:

The INCLUDE symbol should be updated to include the search path for the ODBC Driver Manager include files.

The MYPROG symbol should be set to the main program name, in this case "osqlsamp". The MYFUNCS symbol should hold the name of the object file(s) for the subroutines, in this case "osql.o" (avoid trailing spaces).

The ODBCLIB symbol should be updated to include the search path for the ODBC Driver Manager library and the name of that library.

```
INCLUDE = -I$(MIMINC) -I. $(EXTEND) $(EXTEND_SHL)
-I/usr/local/include
MYPROG = osqlsamp
MYFUNCS = osql.o
ODBCLIB = -L/usr/local/lib - liodbc $(CLIB) $(LIBC)
```

Note! The names of ODBC Driver Manager libraries and include files may vary between different products. These names must match the makefile and the C-source header include statements, respectively.

Then make the osqlsamp program:

```
$ export MIMER_HOME=/opt/mimer821A
$ make
```

To be able to execute the osqlsamp program the **.odbc.ini** file must be installed in the user HOME directory and updated according to the following example (for details, see documentation for the ODBC Driver Manager in use). The **.odbc.ini** file should include lines as in the following example in order to access a Mimer SQL database:

```
[ODBC Data Sources]
hotel=MIMER database

[hotel]
Driver = /usr/lib/libmimer.so

[default]
Driver = /usr/lib/libmimer.so
Database = hotel
```

Note! If the ODBC Data Source name is not equal to the Mimer SQL database name the "Database" parameter should be used in the .odbc.ini file, as shown for the "default" ODBC Data Source above.

VMS

The OSQLSAMP program is not currently distributed on VMS.

Win

On Windows platforms, the examples files are installed in the **dev** directory which is located below the installation directory. The dev directory contains a makefile called **makefile.mak**, which will build all the example programs at the same time. The examples have been tested using Microsoft Visual C.

These instructions assume that Microsoft Visual C is used and that the make command is being executed from the dev directory, appropriate adjustments must be made if this is not the case.

```
$ nmake -f makefile.mak
```

D.6.2 Source code for the OSQLSAMP program (in C)

The main program source is the same as for DSQLSAMP and is in dsqlsamp.ec, the following code, in osql.c replaces the code in dsql.c.

```
/************************************************************************/
/*  osql.c - ODBC sql driver                                            */
/************************************************************************/
/*                                                                      */
/*  This package is an ODBC SQL sample with the following functions:    */
/*                                                                      */
/*      DsqlConnect    - connect to data source                         */
/*      DsqlDisconnect - disconnect from the data source                */
/*      DsqlExecute    - execute an SQL statement                       */
/*      DsqlColNames   - get the column names of a result table         */
/*      DsqlFetch      - fetch the next row from a result table         */
/*      DsqlClose      - close a cursor                                 */
/*      DsqlError      - get error message text                         */
/*                                                                      */
/************************************************************************/
/*                                                                      */
/*  Created by Mimer Information Technology AB.                         */
/*                                                                      */
/*  You have a free right to use, modify, reproduce, and distribute the */
/*  sample files (and/or any modified version) in any way you find      */
/*  useful, provided that you agree that Mimer Information Technology AB */
/*  has no warranty obligations or liability for any sample files which */
/*  are modified.                                                       */
/*                                                                      */
/************************************************************************/
```

Win

```
#include <windows.h>
```

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <ctype.h>
#include "sqlext.h"
#include "dsql.h"       /* function prototype definitions */

#define SQL_BADARG  -3
#define SQL_HANDLES -4

#define MAXHSTMTS   32

#ifndef TRUE
#define TRUE        1
#define FALSE       0
#endif
/*
 *  Local functions
 */
static int    check     (RETCODE);
static int    sethstmt  (HSTMT);
static HSTMT  gethstmt  (int);
static void   drophstmt (int);
```

```
/*
 *  Shared data areas
 */
static HENV   henv;
static HDBC   hdbc;
static HSTMT  hstmt;
static HSTMT  hstmts [MAXHSTMTS];
static SDWORD errcode;
static char   state  [6];
static char   sqlmsg [SQL_MAX_MESSAGE_LENGTH];
static char   buffer [8192];
static char*  format = "[DSQL Driver]%s";

/*****************************************************************************/
/*                                                                           */
/*  Connect to the data source.                                              */
/*                                                                           */
/*  Return codes:                                                            */
/*                                                                           */
/*      1 : successful connection                                            */
/*      0 : wrong username or password                                       */
/*     -1 : error (call DsqlError for message text)                          */
/*                                                                           */
/*****************************************************************************/
int DsqlConnect()
/*****************************************************************************/
{
    RETCODE rc;
    UCHAR   strout[256];
    SWORD   length;
    SQLHWND hwnd;
```

Unix

```
    hwnd = 0;
```

VMS

```
    hwnd = 0;
```

Win

```
    hwnd = GetDesktopWindow();

    /*
     *  Allocate environment.
     */
    if (henv == NULL)
    {
        SQLAllocEnv(&henv);
        if (henv == NULL)
        {
            strcpy(sqlmsg,"cannot allocate ODBC environment\n");
            return -1;
        }
    }
    /*
     *  Allocate a connection handle.
     */
    if (check(SQLAllocConnect(henv,&hdbc)) < 0) return -1;
```

```
    /*
     *  Connect to data source.
     */
```

Unix

VMS

```
    rc = check(SQLDriverConnect(hdbc,hwnd,(UCHAR*)"",0,strout,255,&length,
             SQL_DRIVER_PROMPT));
    if (rc != 0)
    {
        if (check(SQLAllocConnect(henv,&hdbc)) < 0) return -1;

        rc = check(SQLDriverConnect(hdbc,0,(UCHAR*)"DSN=default",SQL_NTS,
                   strout,255,&length,SQL_DRIVER_COMPLETE));
        if (rc != 0)
        {
            if (rc == -2) return 0;
            if (*sqlmsg == '\0')
                strcpy(sqlmsg,"SQLDriverConnect failed...\n");
            return -1;
        }
    }
```

Win

```
    rc = check(SQLDriverConnect(hdbc,hwnd,(UCHAR*)"",0,strout,255,&length,
             SQL_DRIVER_PROMPT));
    if (rc != 0)
    {
        if (rc == -2) return 0;
        return -1;
    }
```

```
    return 1;
}

/*************************************************************************/
/*                                                                       */
/*  Disconnect from the data source.                                     */
/*                                                                       */
/*  Return codes:                                                        */
/*                                                                       */
/*      0 : OK                                                           */
/*     -1 : error (call DsqlError for message text)                      */
/*                                                                       */
/*************************************************************************/
int DsqlDisconnect()
/*************************************************************************/
{
    /*
     *  Disconnect.
     */
    if (check(SQLDisconnect(hdbc)) < 0) return -1;
    /*
     *  Free connection handle.
     */
    if (check(SQLFreeConnect(hdbc)) < 0) return -1;
    /*
     *  Free environment.
     */
    SQLFreeEnv(henv);
    hdbc = SQL_NULL_HDBC;
    henv = SQL_NULL_HENV;
    return 0;
}
```

```
/************************************************************************/
/*                                                                      */
/*  Execute an SQL statement.                                           */
/*                                                                      */
/*  Parameter:                                                          */
/*                                                                      */
/*      char* sqlstmt : SQL statement to be executed                    */
/*                                                                      */
/*  Return codes:                                                       */
/*                                                                      */
/*     +n : a select statement successfully opened,                     */
/*          returning a cursor identifier                               */
/*      0 : a non-select statement executed successfully                */
/*     -1 : error (call DsqlError for message text)                     */
/*                                                                      */
/*  Note:                                                               */
/*                                                                      */
/*      It is assumed that the SQL statement does not contain any       */
/*      parameter markers. Unique statement, descriptor, and cursor names */
/*      are used, which make it possible to have several cursors open at  */
/*      the same time.                                                  */
/*                                                                      */
/************************************************************************/
int DsqlExecute(char* sqlstmt)
/************************************************************************/
{
    SWORD count;
    /*
     *  Check argument.
     */
    if (sqlstmt == NULL) return check(SQL_BADARG);
    /*
     *  Allocate a statement handle.
     */
    if (check(SQLAllocStmt(hdbc,&hstmt)) < 0) return -1;
    /*
     *  Execute SQL statement.
     */
    if (check(SQLExecDirect(hstmt,(UCHAR*)sqlstmt,SQL_NTS)) < 0) return -1;
    /*
     *  Get number of columns to see if this was a select statement or not.
     */
    if (check(SQLNumResultCols(hstmt,&count)) < 0) return -1;
    if (count == 0)
    {
        /*
         *  Non-select statement. Free statement handle.
         */
        if (check(SQLFreeStmt(hstmt,SQL_DROP)) < 0) return -1;
        hstmt = SQL_NULL_HSTMT;
        return 0;
    }
    else
    {
        /*
         *  Select statement. Return index for handle.
         */
        return sethstmt(hstmt);
    }
}
```

```
/************************************************************************/
/*                                                                      */
/*  Get the column names of a result table.                             */
/*                                                                      */
/*  Parameters:                                                         */
/*                                                                      */
/*      int   c      : cursor identifier returned from DsqlExecute      */
/*      char* record : buffer for the result string                     */
/*      int   len    : length of the buffer (must be >= 32)             */
/*                                                                      */
/*  Return codes:                                                       */
/*                                                                      */
/*     +n : OK - number of columns                                      */
/*     -1 : error (call DsqlError for message text)                     */
/*                                                                      */
/*  Note:                                                               */
/*                                                                      */
/*      The column names are returned in a tab ('\t') separated string  */
/*      ending with a newline ('\n'). If the string is too large, it is */
/*      truncated.                                                      */
/*                                                                      */
/************************************************************************/
int DsqlColNames(int c,char* record,int len)
/************************************************************************/
{
    char   name [19];
    SWORD  count;
    SWORD  n,cb,ct,cs,cn;
    UDWORD cp;
    int    length;
    /*
     *  Check arguments.
     */
    if (record == NULL || len < 32) return check(SQL_BADARG);
    /*
     *  Get statement handle.
     */
    hstmt = gethstmt(c);
    /*
     *  Get number of columns.
     */
    if (check(SQLNumResultCols(hstmt,&count)) < 0) return -1;
    for (n = 1; n <= count; n++)
    {
        /*
         *  Get one column name.
         */
        if (check(SQLDescribeCol(hstmt,(UWORD)n,
                                 (UCHAR*)name,(SWORD)sizeof(name),
                                 &cb,&ct,&cp,&cs,&cn)) < 0) return -1;
        /*
         *  Trim spaces and append a tab.
         */
        length = (int)cb;
        if (length == 0) *name = ' ', length = 1;
        while (length > 1 && name[length - 1] == ' ') length--;
        if (length > len - 2) length = len - 2;
        memcpy(record,name,length);
        record += length;
        len    -= length;
        if (len == 2) break;
```

```
        if (n < count)
        {
            *record++ = '\t';
            len--;
        }
    }
    *record++ = '\n';
    *record   = '\0';
    return count;
}

/*****************************************************************************/
/*                                                                           */
/*  Fetch the next row from a result table.                                  */
/*                                                                           */
/*  Parameters:                                                              */
/*                                                                           */
/*      int   c      : cursor identifier returned from DsqlExecute          */
/*      char* record : buffer for the result string                         */
/*      int   len    : length of the buffer (must be >= 32)                 */
/*                                                                           */
/*  Return codes:                                                            */
/*                                                                           */
/*     +n : OK - number of columns                                           */
/*      0 : No more data (End of File)                                       */
/*     -1 : error (call DsqlError for message text)                          */
/*                                                                           */
/*  Note:                                                                    */
/*                                                                           */
/*      The column values are returned in a tab ('\t') separated string     */
/*      ending with a newline ('\n'). It is assumed that the data does       */
/*      not contain tab, newline or null characters. A column value of       */
/*      NULL will be return as '?'. If the string is too large, it is        */
/*      truncated.                                                           */
/*                                                                           */
/*****************************************************************************/
int DsqlFetch(int c,char* record,int len)
/*****************************************************************************/
{
    RETCODE rc;
    SWORD   n;
    SWORD   count;
    SDWORD  cb;
    int     length;
    /*
     *  Check arguments.
     */
    if (record == NULL || len < 32) return check(SQL_BADARG);
    /*
     *  Get statement handle.
     */
    hstmt = gethstmt(c);
    /*
     *  Fetch next row from result table.
     */
    rc = check(SQLFetch(hstmt));
    if (rc == 0)
    {
        /*
         *  Get number of columns.
         */
        if (check(SQLNumResultCols(hstmt,&count)) < 0) return -1;
```

```
            for (n = 1; n <= count; n++)
            {
                /*
                 *  Get one column value.
                 */
                if (check(SQLGetData(hstmt,n,SQL_C_CHAR,
                                     (PTR)buffer,(SDWORD)len,&cb)) < 0) return -1;
                /*
                 *  Check NULL indicator, trim spaces and append a tab.
                 */
                if ((length = (int)cb) == SQL_NULL_DATA)
                {
                    *record++ = '?';
                     len--;
                }
                else
                {
                    if (length == 0) *buffer = ' ', length = 1;
                    while (length > 1 && buffer[length - 1] == ' ') length--;
                    if (length > len - 2) length = len - 2;
                    memcpy(record,buffer,length);
                    record += length;
                    len    -= length;
                }
                if (len == 2) break;
                if (n < count)
                {
                    *record++ = '\t';
                    len--;
                }
            }
            *record++ = '\n';
            *record   = '\0';
            return count;
        }
        else if (rc > 0)
        {
            /*
             *  No more data.
             */
            return 0;
        }
        return -1;
    }

    /*************************************************************************/
    /*                                                                       */
    /*  Close a cursor.                                                      */
    /*                                                                       */
    /*  Parameter:                                                           */
    /*                                                                       */
    /*      int c : cursor identifier returned from DsqlExecute              */
    /*                                                                       */
    /*  Return codes:                                                        */
    /*                                                                       */
    /*      0 : OK                                                           */
    /*     -1 : error (call DsqlError for message text)                      */
    /*                                                                       */
    /*************************************************************************/
    int DsqlClose(int c)
    /*************************************************************************/
    {
        drophstmt(c);
        return check(SQLFreeStmt(hstmt,SQL_DROP));
    }
```

```
/************************************************************************/
/*                                                                      */
/*  Get error message text.                                             */
/*                                                                      */
/*  Parameters:                                                         */
/*                                                                      */
/*      char* message : buffer for the result string                    */
/*      int   len     : length of the buffer (must be >= 32)            */
/*                                                                      */
/*  Return codes:                                                       */
/*                                                                      */
/*      0 : OK                                                          */
/*     -1 : error (cannot get message text)                             */
/*                                                                      */
/*  Note:                                                               */
/*                                                                      */
/*      The message text is returned in a newline ('\n') separated      */
/*      string. If the string is too large, it is truncated.            */
/*                                                                      */
/************************************************************************/
int DsqlError(char* message,int len)
/************************************************************************/
{
    if (message == NULL || len < 32) return -1;
    strncpy(message,sqlmsg,len);
    message[len] = '\0';
    return 0;
}

/************************************************************************/
static int check(RETCODE rc)
/************************************************************************/
{
    char* msg;
    SWORD max;
    SWORD len;
    int   login;

    if (rc == SQL_ERROR || rc == SQL_SUCCESS_WITH_INFO)
    {
        msg   = sqlmsg;
        max   = (SWORD)sizeof(sqlmsg);
        login = FALSE;
        while (SQLError(henv,hdbc,hstmt,
                        (UCHAR*)state,&errcode,
                        (UCHAR*)msg,max,&len) == 0)
        {
            msg += len; max -= len;
            len  = (SWORD)sprintf(msg,"\nSQLSTATE:%s\n\n",state);
            msg += len; max -= len;
            if (strcmp(state,"28000") == 0) login = TRUE;
        }
        if (login) return -2;
        if (hstmt && rc == SQL_ERROR)
        {
            SQLFreeStmt(hstmt,SQL_DROP);
            hstmt = SQL_NULL_HSTMT;
        }
        else if (hstmt == NULL && hdbc && rc == SQL_ERROR)
        {
            SQLFreeConnect(hdbc);
            hdbc = SQL_NULL_HDBC;
        }
    }
```

```
    switch (rc)
    {
    case SQL_SUCCESS_WITH_INFO:
    case SQL_SUCCESS:
        return 0;

    case SQL_NO_DATA_FOUND:
        return 1;

    case SQL_STILL_EXECUTING:
        sprintf(sqlmsg,format,"Still executing");
        return -1;

    case SQL_NEED_DATA:
        sprintf(sqlmsg,format,"Need data");
        return -1;

    case SQL_HANDLES:
        sprintf(sqlmsg,format,"Too many handles");
        return -1;

    case SQL_BADARG:
        sprintf(sqlmsg,format,"Bad argument");
        return -1;

    case SQL_INVALID_HANDLE:
        sprintf(sqlmsg,format,"Invalid handle");
        return -1;

    case SQL_ERROR:
    default:
        return -1;
    }
}

/*****************************************************************************/
static int sethstmt(HSTMT hstmt)
/*****************************************************************************/
{
    int id;

    for (id = 0; id < MAXHSTMTS; id++)
    {
        if (hstmts[id] == 0)
        {
            hstmts[id] = hstmt;
            return id + 1;
        }
    }
    return check(SQL_HANDLES);
}

/*****************************************************************************/
static HSTMT gethstmt(int id)
/*****************************************************************************/
{
    if (id > 0 && id < MAXHSTMTS + 1)
    {
        return hstmts[id - 1];
    }
    return NULL;
}
```

```
/*****************************************************************************/
static void drophstmt(int id)
/*****************************************************************************/
{
    if (id > 0 && id < MAXHSTMTS + 1)
    {
        hstmts[id - 1] = NULL;
    }
}
```

INDEX

A

access clause for routines 8-6
access rights 4-2
  for cursors 5-4
access rights and routines 8-28
accessing data 5-1
active connection 4-5
ALLOCATE DESCRIPTOR 7-4
arrays 3-1
assignment - SET statement 8-13

B

back-up protection 6-10
BEGIN DECLARE SECTION 3-1

C

C (programming language)
  comments A-2
  data types A-4
  host variables A-2
  line continuation A-2
  null terminated strings A-3
  preprocessor output A-5
  see also host languages 2-2
  statement delimiters A-2
CALL statement 8-17
calling procedures 8-17
CASE statement 8-14
closing cursors 5-4
COBOL
  comments A-8
  data types A-9
  host variables A-8
  line continuation A-7
  preprocessor output A-10
  see also host languages 2-2
  statement delimiters A-7
comments 2-4
  in C A-2
  in COBOL A-8
  in FORTRAN A-11
comments in routines 8-18
COMMIT 6-4
committing transactions 6-1
compiler 2-7
compound sql statement 8-6

concurrency control 6-1
condition names 8-25
conditional execution - CASE statement 8-14
conditional execution - IF statement 8-13
connecting to a database 4-3
connection name 4-3
connections
  and transactions 6-4
  cursors 5-5
continue handlers 8-27
current row 5-9
cursor-independent data manipulation 5-8
cursors 5-1
  access rights 5-4
  and program idents 4-5
  closing 5-4
  declaration 5-2
  declaring in PSM 8-19
  evaluating SELECT statement 5-3
  for join conditions 5-6
  for UPDATE and DELETE 5-9
  in dynamic SQL 7-9
  in multiple connections 5-5
  in transactions 6-8
  opening 5-3
  position in result set 5-4, 5-9
  resource allocation 5-5
  restrictions on use in PSM 8-20
  stacking 5-7
cursors in routines 8-19

D

data errors B-7
data types
  in C A-4
  in COBOL A-9
  in FORTRAN A-13
databank access errors B-22
databank options 6-10
database concept 4-1
database connection 4-3
deadlock 6-3
DEALLOCATE DESCRIPTOR 7-4
declaration of SQLSTATE 3-3
DECLARE SECTION 3-1
declaring condition names 8-25
declaring cursors 5-2, 8-19
declaring exception handlers in routines 8-25
declaring host variables 5-2
declaring routine parameters 8-5
declaring routine variables 8-9
DELETE 5-8
delimiters
  in C A-2
deprecated features C-1
  parameter markers C-2
  SET TRANSACTION CHANGES C-2
  SQLCODE C-1

SQLDA C-1
VARCHAR(size) C-2
DESCRIBE INPUT 7-7
describing dynamic SQL statements 7-6
designing transactions 6-2
deterministic clause for routines 8-6
diagnostics area 3-4
disconnecting 4-4
disk crash 6-11
dormant cursors 5-5
drop and revoke involving routines 8-29
dynamic SQL 7-1
cursors 7-9
describing statements 7-6
example framework 7-10
executing statements 7-8
preparing statements 7-5
statement object form 7-5
statement source form 7-5
statements 7-3
dynamic SQL program example D-1

E

embedded SQL 2-2
program structure 2-7
restrictions on host language code 2-4
scope 2-2
END DECLARE SECTION 3-1
ending transactions 6-4
ENTER 4-5
error handling 10-1
in transactions 6-9
exception conditions 10-1
exception handlers 8-25
EXEC SQL 2-3
EXECUTE on routine 8-28
executing dynamic SQL statements 7-8
exit handlers 8-26
extended dynamic cursors 7-5
extended dynamic statements 7-5

F

FETCH 5-3
FORTRAN
comments A-11
host variables A-12
line continuation A-11
preprocessor output A-14
see also host languages 2-2
statement delimiters A-11
statement margins A-11
statement numbers A-11
functions
invoking 8-17

G

general exception handlers 8-25
GET DIAGNOSTICS 3-4, 8-19
GRANT OPTION 4-3

group idents 4-2

H

handlers 8-25
host language included code 2-3
host languages 2-2, A-1
host variables 3-2
  arrays 3-1
  declaration 3-1, 5-2
  declarations 2-7
  in C A-2
  in COBOL A-8
  in cursor declarations 5-3
  in dynamic SQL 7-2
  in FORTRAN A-12
  in SQL statements 3-1
  names 2-4

I

IDENT privilege 4-2
identifying SQL statements 2-3
IF statement 8-13
implicit connection 4-4
included code
  host language 2-3
indicator variables 3-2
INSERT 5-8
interactive environments 7-1
internal Mimer SQL return codes B-4
interrupted transactions 6-11
invoking functions 8-17
invoking procedures 8-17
ISOLATION LEVEL's in transactions 6-5
Iteration
  LOOP statement 8-16
  REPEAT statement 8-16
  WHILE statement 8-16

J

Java 2-1
JDBC 2-1
join retrievals
  using cursors 5-6

L

LEAVE 4-5
LEAVE RETAIN 4-5
LEAVE statement 8-7
limits B-9
line continuation
  in C A-2
  in COBOL A-7
  in FORTRAN A-11
locking 6-3
LOG databank option 6-10
logging 6-10
LOOP statement 8-16
loops 6-2

M

managing exception conditions in routines 8-23
minus sign
in COBOL variable names A-8
modules 8-12
multiple connections 4-4
multiple tables in data retrieval 5-6

N

NULL 3-2
NULL databank option 6-10
null terminated strings A-3

O

object form of dynamic SQL statements 7-5
OPEN 5-3
optimistic concurrency control 6-1
optimization 2-7
OS_USER idents 4-1

P

parameter markers 7-2
in SELECT statements 7-10
parameters in routines 8-5
Parts explosion 5-7
Persistent Stored Modules 8-1
positioning cursors 5-4
preparing dynamic SQL statements 7-5
preprocessing 2-4
WHENEVER statements 10-2
preprocessor output
in C A-5
in COBOL A-10
in FORTRAN A-14
privileges 4-2
procedure-control-statement (see Chapter 6 of the *Mimer SQL Reference Manual*)
procedures
invoking 8-17
returning result sets 8-21
program construction errors B-19
program examples D-1
program idents 4-2, 4-5
program structure 2-7
PSM 8-1

Q

quotation marks
in C A-2

R

READ ONLY transactions 6-5
READ WRITE transactions 6-5
read-only cursors 5-3, 5-9
read-set 6-1
REPEAT statement 8-16
RESIGNAL statement 8-24
resignaling exceptions in routines 8-24
result set procedure CALL 5-1
result set procedures 8-21

retrieving data 5-3
retrieving single rows 5-5
RETURN statement 8-22
revoke and drop involving routines 8-29
ROLLBACK 6-4
routines
  access clause 8-6
  access rights 8-28
  beware using drop and revoke 8-29
  comments in 8-18
  declaring condition names 8-25
  declaring cursors 8-19
  declaring exception handlers 8-25
  declaring variables 8-9
  deterministic clause 8-6
  invoking 8-17
  managing exception conditions 8-23
  parameters 8-5
  restrictions 8-18
  scope in 8-6
  SQL constructs in 8-13
run-time errors 10-2

S

scope in routines 8-6
scope of embedded SQL 2-2
SCROLL 5-3
scrollable cursor 5-3
SELECT INTO 5-5
SELECT statement 5-1
semantic errors 10-1
SET statement 8-13
SET TRANSACTION 6-3
SIGNAL statement 8-24
signaling exceptions in routines 8-24
single row SELECT 5-5
source form of dynamic SQL statements 7-5
specific exception handlers 8-26
SQL compiler 2-7
SQL constructs in routines 8-13
SQL descriptor area 3-4, 7-4
  COUNT field 7-4
  structure 7-4
SQL statement identifier 2-3
SQL statements
  errors B-10
  using host variables 3-1
SQLSTATE 3-3, 10-2, B-2
  class 3-3
  fields 3-3
  return codes B-2
  subclass 3-3
stacking cursors 5-7
starting transactions 6-3
statement delimiters
  in C A-2
  in COBOL A-7
  in FORTRAN A-11

statement margins
in FORTRAN A-11
statement numbers in FORTRAN: A-11
stored functions and procedures 8-1
stored procedures and functions 8-1
stored routines and modules (PSM) 8-1
subprogram names 2-4
subroutine names 2-4
syntax errors 10-1
system failure 6-11

T

table access errors B-22
TRANS databank option 6-10
transaction consistency - ISOLATION LEVEL 6-5
transaction diagnostics size 6-6
transaction optimization 6-5
transactions 6-1
and cursors 6-8
build-up 6-1
design 6-2
error handling 6-9
interrupted 6-11
with loops 6-2
tree structure
traversing 5-7
types A-13

U

updatable cursors 5-9
UPDATE 5-8
UPDATE CURRENT 5-9
USAGE ON DOMAIN privilege 4-2

V

variables
declaring 8-9
host (see under ‘host variables’)

W

warnings B-4
WHENEVER 10-2
in transactions 6-10
WHILE statement 8-16
white-space A-2, A-7, A-11
write-set 6-1