Rexx/SQL

Table of Contents

Table of Contents

A Rexx interface to SQL databases

Version 2.5

10 October 2006

1. Introduction

This document defines an interface to provide access to SQL databases for Rexx programs. Rexx/SQL consists of a number of external Rexx functions which provide the necessary capabilities to connect to, query and manipulate data in any SQL database. This document is designed to assist in the implementation of this interface for any SQL-based database system that provides an appropriate 3GL API.

An appendix to this document is included for each implementation of this interface providing implementation-specific features. Where implementations may differ, this is highlighted in the function definitions to assist the user where source code compatibility between different database vendors is required.

2. Overview

Rexx/SQL consists of Rexx external functions that allows a Rexx program to communicate with a SQL database.

Actions requested of the database are made by calling these external functions. Information returned to the Rexx program as a result of these actions is done principally through the Rexx variable pool.

The Rexx external functions are:

- **SQLCONNECT** - connect to the SQL database
- **SQLDISCONNECT** - break the connection to the SQL database made by SQLCONNECT
- **SQLDEFAULT***(1)* - switch the default connection to another open connection
- **SQLCOMMAND** - issue a SQL statement to the connected database
- **SQLPREPARE** - allocate a work area for a SQL statement and prepare it for processing
- **SQLDISPOSE** - deallocate a work area for a statement
- **SQLOPEN** - open a cursor for a prepared SELECT statement
- **SQLCLOSE** - close an opened cursor
- **SQLFETCH** - fetch the next row from the open cursor
- **SQLGETDATA** - extracts part of a column from a fetched row
- **SQLEXECUTE** - execute a prepared statement
- **SQLCOMMIT** - commit the current transaction
- **SQLROLLBACK** - rollback the current transaction
- **SQLDESCRIBE***(1)* - describe expressions from a SELECT statement
- **SQLVARIABLE***(2)* - set or retrieve default run-time values
- **SQLGETINFO***(2)* - retrieve Rexx/SQL information for a connection
- **SQLDATASOURCES***(1)* - obtain a list of the data sources available
- **SQLTABLES***(1)* - obtain a list of tables from the current data source
- **SQLCOLUMNS***(1)* - obtain a list of columns in tables from the current data source

*(1)*Function may not be supported in all implementations.
*(2)*Values that can be set can vary between implementations.

Status values set by the Rexx external functions are:

- **SQLCA.SQLCODE** - result code of last SQL operation
- **SQLCA.SQLERRM** - text of any error message associated with the above result code
- **SQLCA.SQLSTATE** - a detailed status string (N/A on some ports)
- **SQLCA.SQLTEXT** - text of the last SQL statement
- **SQLCA.ROWCOUNT** - number of rows affected by the last SQL operation
- **SQLCA.FUNCTION** - name of the Rexx external function last called
- **SQLCA.INTCODE** - Rexx/SQL interface error number
- **SQLCA.INTERRM** - text of last Rexx/SQL interface error

At the start of each external function call, the SQLCA. stem values are reset if the previous external function call caused an error or warning. This is done to make Rexx/SQL faster by not resetting these values if they don't need to be. The SQLCA. stem should be considered read-only and the values not changed or DROPped by the Rexx/SQL program.

3. Functions

This section provides the full syntax and usage of each function that comprises Rexx/SQL.

SQLCONNECT([*connection name*], [*username*], [*password*], [*database*], [*host*])

Establishes a connection to the database server. The newly established connection is made the default database connection.

Arguments:

connection name
: This is an optional name for the connection to be opened. If you need to have multiple connections opened at once, you will need to specify a connection name. For those implementations that do not support multiple connections, this argument is not supported.

username
: This is the name used to connect to the database.

password
: This is the password associated with the *username*.

database
: This is the name that the database to which connection is required is known.

host
: This is the name of the host on which the *database* resides. The format of this host string depends on the database vendor and operating system.

Some arguments may be mandatory depending on the platform. See the appendices for more details.

Returns:

success: zero
failure: a negative number

SQLDISCONNECT([*connection name*])

Closes a connection with the database server. All open cursors for the database connection are closed. All allocated work areas for the database connection are deallocated.

Arguments:

connection name

An optional connection name, as specified in the **SQLCONNECT** function. If no connection name is specified, the default (and only) connection is disconnected.

Returns:

success: zero
failure: a negative number

SQLDEFAULT([*connection name*])

Sets the default database connections to be that which is specified or if no connection name is specified, the current connection name is returned.

Arguments:

connection name
An optional connection name specifying the database connection to be made the default connection.

Returns:

with no argument:
the name of the current database connection or an empty string if no database connection is current.
with an argument:
success: zero
failure: a negative number

SQLCOMMAND(*statement name,sql statement*[,*bind1*[,*bind2*[,...[,*bindN*]]]])

Executes an SQL statement as a single step. The statement is executed in the default work area for the default database connection. No bind values may be passed for **DDL** statements. Bind values may optionally be passed for **DML** statements.

Arguments:

statement name
A name to identify the *sql statement* and used to name the compound variable created when *sql statement* is a SELECT statement. The results of the SELECT statement are returned in compound variables with this name as the stem.

sql statement
Any valid **DDL** or **DML** statement. For **DML** statements, the statement may contain placemarkers to which values may be bound.
The format of these placemarkers is implementation dependant.

bind1...bindN
Values supplied to bind to the placemarkers.
The format of bind values is implementation dependant.

Returns:

success: zero
failure: a negative number

When the *sql statement* is a SELECT, all column values are returned as Rexx *arrays*. The compound variable name is composed of the statement name followed by a period, followed by the column name specified in the SELECT statement, followed by a number corresponding to the row number. As with all Rexx arrays, the number of elements in the array is stored in the zeroth element. If no *statement name* is specified, a default string is used; usually **SQL**. See 5. Implementation Notes for information when this is not the case.

If the column selected consists of a constant, or includes a function, a valid Rexx variable may not be able to be generated. See the implementation specific Appendixes for details on how each implementation handles this.

After a successful **DML** statement, the variable **SQLCA.ROWCOUNT** is set to the number of rows affected by the statement.

Because the contents of all columns for all rows are returned from a SELECT statement, the statement may return many rows and exhaust available memory. Therefore, the use of the SQLCOMMAND function should be restricted to queries that return a small number of rows. For larger queries, use a combination of SQLPREPARE, SQLOPEN, SQLFETCH, and SQLCLOSE.

Example:

```
rc = sqlcommand(s1,"select ename, empno from emp")
```

If the SELECT statement returns 3 rows then:

- S1.ENAME.0 = 3
- S1.ENAME.1 = "SCOTT"
- S1.ENAME.2 = "SMITH"
- S1.ENAME.3 = "BROWN"
- S1.EMPNO.0 = 3
- S1.EMPNO.1 = "1234"
- S1.EMPNO.2 = "1437"
- S1.EMPNO.3 = "1555"

SQLPREPARE(*statement name,sql statement*)

Allocates a work area to a SQL statement and prepares the statement for processing.

If the statement is a **DDL** or **DML** statement then it must be executed by a subsequent call. For **DDL**, INSERT, UPDATE and DELETE commands, the statement must be executed by calling SQLEXECUTE. For a SELECT command, the statement must be executed as a cursor. This requires calling SQLOPEN followed by multiple calls to SQLFETCH and optionally calling SQLCLOSE.

Arguments:

statement name
A name to identify the *sql statement*.

sql statement
Any valid **DDL** or **DML** statement. For **DML** statements, the statement may contain placemarkers to which values may be bound.
The format of these placemarkers is implementation dependant.

Returns:

success: zero
failure: a negative number

SQLDISPOSE(*statement name*)

Deallocates a work area from a statement and frees all internal resources associated with the statement. If a cursor is open for the nominated statement an implicit close is issued.

Arguments:

statement name

A name to identify the *sql statement* to be disposed.

Returns:

success: zero
failure: a negative number

SQLOPEN(*statement name*[,*bind1*[,*bind2*[,...[,*bindN*]]]])

Opens a cursor for the nominated statement. The statement must be a query (a SELECT statement) and must have been prepared prior to opening (with SQLPREPARE). Opening the cursor, binds any supplied values to the corresponding placemarkers and then executes the SELECT statement. The first row is made ready to be fetched. If a cursor was already open for the named statement then it will be automatically closed prior to reopening the cursor.

Arguments:

statement name
: A name to identify the *sql statement*.

bind1...bindN
: Values supplied to bind to the placemarkers.
: **The format of bind values is implementation dependant.**

Returns:

success: zero
failure: a negative number

SQLCLOSE(*statement name*)

Ends execution of a cursor. This frees much of the database server resources associated with a cursor. The statement does not have to be reparsed if the cursor is later reopened unless the statement has been disposed (ie by calling SQLDISPOSE for the *statement name*).

Arguments:

statement name

A name to identify the *sql statement*.

Returns:

success: zero
failure: a negative number

SQLFETCH(*statement name*,[*number rows*])

Fetches the next row (or rows) for the nominated statement. There must be an open cursor for the named statement. If the optional *number rows* is not specified, a *single row* fetch is carried out, otherwise a multi row fetch is carried out.

For *single row* fetches, a compound variable is created for each column name identified in the *sql statement* parsed in the SQLPREPARE call, with the stem being *statement name* and the tail corresponding to the column name.

For *multi row* fetches, a Rexx *array* is created for each column name in the *sql statement* parsed in the SQLPREPARE call. See SQLCOMMAND for a full description of the format of the variables. Variable tails always start with 1.

Arguments:

statement name
    A name to identify the *sql statement*.
number rows
    An optional number specifying how many rows are to be fetched.

Returns:

success: a number >= zero.
    a value of zero indicates no more rows are available to be fetched.
    for *single row* fetches, a value > zero represents the row number of the row just fetched.
    for *multi row* fetches, a value > zero indicates the number of rows fetched. Normally this value equals number rows. If this value is less than number rows, no more rows are available to be fetched. This value can never be greater than number rows. The variable **SQLCA.ROWCOUNT** is set to the value returned.
failure: a negative number

SQLGETDATA(*statement name*,*column name*,[*start byte*],[*number bytes*][,*file name*])

Extracts part (or all) of a data column from the current, fetched row. The *column name* specifed corresponds to a column name within the current query. The column contents are returned either into a Rexx compound variable or directly written to a file (if the optional *file name* argument is passed). The format of the compound variable, consists of the stem being *statement name* and the tail corresponding to the *column name*.

Arguments:

statement name
: A name to identify the *sql statement*.

column name
: The name of the column within the current query for which a portion of data is to be retrieved. Some implementations do not support this function. To determine if an implementation supports this, call SQLVARIABLE with the *SUPPORTSSQLGETDATA* argument. Still other implementations restrict which columns can be retrieved in parts, based on the datatype of the column.

start byte
: The starting byte from which to retrive column contents. The first byte of a column is 1. This argument is optional. A value of 0 can be passed; see *file name* argument for further details.

number bytes
: The number of bytes to retrieve. An error occurs if this value exceeds 65536. This argument is optional. A value of 0 can be passed; see *file name* argument for further details.

file name
: This is the name of an operating system file, into which the complete contents of the column is written. The *start byte* and *number bytes* arguments must either be specified as zero, or not specified at all, to allow this option.

Returns:

success: a number >= zero which corresponds to the number of bytes retrieved.
a value of zero indicates no more data are available to be retrieved.
failure: a negative number

Example:

```
rc = sqlprepare(s1,"select ename, empno, empaddr  from emp")
rc = sqlopen(s1)
Do Forever
   rc = sqlfetch(s1)
   If rc <0 Then Abort()
   If rc = 0 Then Leave
   Do i = 0
      rc = sqlgetdata(s1,'ename',(i*100)+1,100)
      If rc  <0 Then Abort()
      If rc = 0 Then Leave
      Say 'Column: ename:' s1.ename
   End
   rc = sqlgetdata(s1,'empaddr',,,'/tmp/empaddr.txt')
End
```

SQLEXECUTE(*statement name*[,*bind1*[,*bind2*[,...[,*bindN*]]]])

Executes a prepared statement for non-SELECT **DML** statements (i.e. INSERT, UPDATE and DELETE).

Arguments:

statement name
A name to identify the *sql statement*.
bind1...bindN
Values supplied to bind to the placemarkers.
The format of bind values is implementation dependant.

Returns:

success: zero
The variable **SQLCA.ROWCOUNT** is set to the number of rows affected by the **DML** statement executed.
failure: a negative number

SQLCOMMIT()

Commit the current transaction.

Arguments:

none

Returns:

success: zero
failure: a negative number

SQLROLLBACK()

Rollback the current transaction.

Arguments:

none

Returns:

success: zero
failure: a negative number
warning: a positive number

SQLDESCRIBE(*statement name* [,*stem name*])

Describes the expressions returned by a SELECT statement. The statement should first be prepared (with SQLPREPARE) and then described. Creates a compound variable for each column in the select list of the sql statement, with a stem equal to the *statement name*, followed by 'COLUMN' and with at least the following column attributes: NAME, TYPE, SIZE, SCALE, PRECISION, NULLABLE.

- NAME - name of the column
- TYPE - the datatype of the column represented as a database-specific string
- SIZE - the size of the column as known to the database
- SCALE - the overall size of the column
- PRECISION - the column's precision; usually the number of decimal places
- NULLABLE - **1** if the column allows NULL values, **0** otherwise

The full list of column attributes can be obtained by calling SQLVARIABLE with the *DESCRIBECOLUMNS* parameter. **See the database-specific appendix for the meaning of other column attributes returned.**

Arguments:

statement name
: A name to identify the *sql statement*.

stem name
: An optional name specifying the stem name of the Rexx variables created.

Returns:

success: a positive number, or zero, indicating the number of expressions in the select list of the SELECT statement
failure: a negative number

Example:

```
rc = sqlprepare(s2,"select ename, empno from emp")
rc = sqldescribe(s2,"AA")
```

results in the following Rexx variables being set:

- AA.COLUMN.NAME.1 == "ENAME"
- AA.COLUMN.NAME.2 == "EMPNO"
- AA.COLUMN.TYPE.1 == "VARCHAR2"
- AA.COLUMN.TYPE.2 == "NUMBER"
- AA.COLUMN.SIZE.1 == "20"
- AA.COLUMN.SIZE.2 == "6"
- AA.COLUMN.PRECISION.1 == "20"
- AA.COLUMN.PRECISION.2 == "40"
- AA.COLUMN.SCALE.1 == "0"
- AA.COLUMN.SCALE.2 == "0"
- AA.COLUMN.NULLABLE.1 == "1"
- AA.COLUMN.NULLABLE.2 == "0"

The values returned are implementation dependant.

SQLVARIABLE(*variable name*[,*variable value*])

Set or get the value for the specified variable.

The following variables are available in all implementations:

- **VERSION** *(readonly)* the version of Rexx/SQL, consisting of:
    - ♦ *package name* - usually **REXXSQL**
    - ♦ *Rexx/SQL version* - numerical version; eg. 1.0
    - ♦ *Rexx/SQL date* - Rexx standard date format; eg. 10 Jun 1995
    - ♦ *OS platform* - current operating system
    - ♦ *database platform* - type of the current database

  eg. REXXSQL 1.0 10 Jun 1995 OS/2 ORACLE
- **DEBUG** *(setable)* level of debugging requested.
    - ♦ *0* - no debugging information displayed (defualt)
    - ♦ *1* - Rexx variables displayed as set
      Equivalent to *-v* command line flag.
    - ♦ *2* - function entry/exit information displayed
      Equivalent to *-d* command line flag.
    - ♦ *3* - both level 1 and 2 debugging information displayed
      Equivalent to *-dv* command line flags.
- **ROWLIMIT** *(setable)*
  this is used to limit the number of rows fetched by a SELECT statement passed to SQLCOMMAND. A value of zero indicates no limit. The default value is zero.
- **LONGLIMIT** *(setable)*
  this is used to limit the number of bytes retrieved by a SELECT statement that returns a *long* datatype. The default value is 32768.
- **SAVESQL** *(setable)*
  this is used to indicate if the text of the last SQL statement is to be saved. If this variable is set to 1, then **SQLCA.SQLTEXT** will have the value of the last SQL statement; if set to 0 **SQLCA.SQLTEXT** will equal "". The default for this variable is 1.
- **AUTOCOMMIT** *(setable)*
  this is used to set AUTOCOMMIT ON or OFF. With AUTOCOMMIT ON, each statement is automatically commited to the database; hence the duration of a transaction is limited to a single statement. With AUTOCOMMIT OFF (the default), a transaction lasts until an explicit end to the transaction is made (calling SQLCOMMIT or SQLROLLBACK) or by an implicit commit (eg, in Oracle, **DDL** statements cause an implicit commit of the transaction. A value of 1 turns AUTOCOMMIT ON; zero turns it OFF. This variable is ignored by those databases that do not provide transaction processing. The setting of *AUTOCOMMIT* can be changed anytime during a connection.
- **IGNORETRUNCATE** *(setable)*
  this is used to determine what action is to occur when a column value is truncated. The default action; ie IGNORETRUNCATE OFF, will result in an error message and the Rexx/SQL function will fail. With IGNORETRUNCATE ON, the data will be truncated but no error will result. A value of 1 turns IGNORETRUNCATE ON; a value of 0 turns IGNORETRUNCATE off.
- **NULLSTRINGOUT** *(setable)*
  this is used to enable the user to specify the string that is returned for column values that are NULL. By default, an empty string is returned. This does not enable the differentiation between a NULL column value and a column that has an empty string; "" as its value. The length of this string is limited to 30 characters.

- **NULLSTRINGIN** *(setable)*
  this is used to enable the user to specify the string that represents a null column value in bind variables. By default, whenever an empty string; "" is passed as the value of a bind variable, Rexx/SQL will convert this to a NULL column value. The default will not allow differentiation between a NULL column value and an empty string, hence it would be a good idea to define this variable whenever you envisage supplying NULL values as bind variables.
- **SUPPORTSPLACEMARKERS** *(readonly)*
  this returns **1** if the database supports placemarkers in a SQL statement. A value of **0** indicates the database does not support placemarkers.
- **STANDARDPLACEMARKERS** *(setable)*
  this is used to enable the specification of **?** as a placemarker in a SQL statement on those databases that support placemarkers other than **?**. Specifying a *variable value* of **1**, enables this feature; a value of **0** disables it.
- **SUPPORTSDMLROWCOUNT** *(readonly)*
  this returns **1** if the database can respond with the number of rows affected by a DML statement; ie DELETE, UPDATE and INSERT. A value of **0** indicates the database cannot determine the number of rows affected, and will always return 0.

Arguments:

variable name
The name of the variable who's value is to be set or retrieved.
The names of variables may be implementation dependant.

variable name
The name of the variable who's value is to be retrieved.

Returns:

with *variable name* specified:
zero if a valid *variable name* specified and it is able to be set;
a negative number if the *variable name* is invalid or the *variable name* is not able to be set.

with *variable value* NOT specified:
the current value of the variable or a negative number if the *variable name* is invalid.

SQLGETINFO([*connection name*],*variable name [,stem name]*)

Retrieves values from the connected database. This function is similar to SQLVARIABLE, but requires a database connection.

Like SQLVARIABLE this function returns its information in the return string, but by specifying a stem name, the results are returned in that stem variable. The stem name **must** include a trailing period.
The *stem name* option is useful for the **DATATYPES** option, as many databases have datatype names that contain spaces.

Arguments:

connection name
: This is the connection name used when retrieving variables that require a database connection. If the connection name is not used and the variable requires a database connection, the current connection is used.

variable name
: The name of the variable who's value is to be retrieved.

Returns:

with *stem name* specified:
: the value of the variable name option in a compound variable. The compound variable consists of a stem corresponding to *connection name* and a tail corresponding to *variable name*. A return code of 0 indicates the function completed successfully, or a negative number representing an internal or database error.

with *stem name* NOT specified:
: the value of the variable name option

The following options are valid for this function:

- **DATATYPES**
  this returns a space seperated list of column attributers appropriate for the database. See SQLDESCRIBE and the database-specific appendix for a list and description of these attributes.
- **DESCRIBECOLUMNS**
  this returns a space seperated list of column attributers appropriate for the database. See SQLDESCRIBE and the database-specific appendix for a list and description of these attributes.
- **SUPPORTSTRANSACTIONS**
  this returns **1** if the database supports *transactions.* ie. the SQLCOMMIT and SQLROLLBACK functions actually do something. A value of **0** indicates the database does not support *transactions.*
- **SUPPORTSSQLGETDATA**
  this returns **1** if the database supports the Rexx/SQL function; SQLGETDATA. A value of **0** indicates the database does not support SQLGETDATA.
- **SUPPORTSTHREADS**
  this returns **1** if the database client software (what Rexx/SQL uses to interface to the database) is thread-safe. A value of **0** indicates the database client software may or may not be thread-safe.
- **DBMSNAME**
  this returns the name of the database currently connected to and optionally a version number of that database.
- **DBMSVERSION**

this returns the version number of the database currently connected to, provided the database client software provides this information.

Example: The following example shows the use of SQLGETINFO with and without a stem variable parameter:

```
Say "Getting mSQL datatypes with stem..."
rc = sqlgetifno("c1","DATATYPES","dt.")
Do i = 1 To dt.0
   Say i '-' dt.i
End
Say "Getting mSQL datatypes without stem..."
Say sqlgetifno(c1,"DATATYPES")
```

The output from the above code sample is:

```
Getting mSQL datatypes with stem...
1 - INT
2 - CHAR
3 - REAL
Getting mSQL datatypes without stem...
INT CHAR REAL
```

SQLDATASOURCES(*stem name*)

Obtains a list of the data sources available. The list of data source names is returned as a compound variable in the *stem name*

Arguments:

stem name
A stem that will contain the list of data sources.

Returns:

success: a number >= zero.
a value of zero indicates success.
failure: a negative number

Stem variable names set:
stem name.DSN_NAME.n
Data source name
stem name.DSN_DESCRIPTION.n
Data source description

Example:
The following example shows the use of SQLDATASOURCES:

```
Say "Getting ODBC Data Sources..."
rc = sqldatasources( "!ds." )
Do i = 1 To !ds.dsn_name.0
   Say 'Name:' !ds.dsn_name.i 'Description:' !ds.dsn_description.i
End
```

The output from the above code sample on my machine is:

```
Getting ODBC Data Sources...
Name: Visual FoxPro Tables Description: Microsoft Visual FoxPro Driver
Name: Visual FoxPro Database Description: Microsoft Visual FoxPro Driver
Name: MQIS Description: SQL Server
Name: SOLID Description: SOLID ODBC Driver 3.50
Name: EASY1 Description: SQL Server
```

SQLTABLES(*stem name*,*qualifier name*,*owner name*,*table name*,*table type*)

Obtains a list of the tables in the current database. There must be an open connection for the database. The list of tables names is returned as a compound variable in the *stem name*

Arguments:

stem name
: A stem that will contain the list of tables.

table qualifier
: An optional string value specifying the table qualifier. You can use standard SQL wildcard characters in the string on some platforms.

table owner
: An optional string value specifying the table owner. You can use standard SQL wildcard characters in the string on some platforms.

table name
: An optional string value specifying the table name. You can use standard SQL wildcard characters in the string on some platforms.

table type
: An optional string value specifying the table type. The table type can be an empty string, which will result in all types of database objects being returned, or or you can use various values like *TABLE*, *VIEW* or *SYSTEM TABLE*. On some platforms a combination of the values separated by commas (,) can be specified.

Returns:

success: a number >= zero.
: a value of zero indicates success.

failure: a negative number

Stem variable names:

stem-name.TABLE_CATALOG.n
: Table qualifier

stem-name.TABLE_OWNER.n
: Table owner

stem-name.TABLE_NAME.n
: Table name

stem-name.TABLE_TYPE.n
: Table type

stem-name.TABLE_DESCRIPTION.n
: Table description

SQLCOLUMNS(*stem name*,*qualifier name*,*owner name*,*table name*,*table type*)

Obtains a list of the columns in tables in the current database. There must be an open connection for the database. The list of column names is returned as a compound variable in the *stem name*

Arguments:

stem name
A stem that will contain the list of tables.

table qualifier
An optional string value specifying the table qualifier. You can use standard SQL wildcard characters in the string on some platforms.

table owner
An optional string value specifying the table owner. You can use standard SQL wildcard characters in the string on some platforms.

table name
An optional string value specifying the table name. You can use standard SQL wildcard characters in the string on some platforms.

table type
An optional string value specifying the table type. The table type can be an empty string, which will result in all types of database objects being returned, or or you can use various values like *TABLE*, *VIEW* or *SYSTEM TABLE*. On some platforms a combination of the values separated by commas (,) can be specified.

Returns:

success: a number >= zero.
a value of zero indicates success.
failure: a negative number

Stem variable names:

stem-name.TABLE_CATALOG.n
Table qualifier

stem-name.TABLE_OWNER.n
Table owner

stem-name.TABLE_NAME.n
Table name

stem-name.COLUMN_NAME.n
Column name

stem-name.COLUMN_TYPE.n
Data type name for the column

stem-name.COLUMN_SIZE.n
Display size of column

stem-name.COLUMN_PRECISION.n
Precision of decimal column

stem-name.COLUMN_SCALE.n
Decimal scale factor

stem-name.COLUMN_NULLABLE.n
Column nullable flag

stem-name.COLUMN_DESCRIPTION.n
Column description

SQLLOADFUNCS()

This function is used to load all the Rexx/SQL external functions and to initialise internal data used by the package. It **must** be called every time before using other Rexx/SQL functions. This function is called after the function has been loaded with the Rexx builtin function rxfuncadd().

Although this function is necessary only for dynamic library implementations of Rexx/SQL, it can be called by the executable version of Rexx/SQL. In this case it does nothing.

Arguments:

none

Returns:

success: zero
failure: a negative number

SQLDROPFUNCS()

This function is used to terminate Rexx/SQL and free up all resources that have been used.

It should be called at the end of every Rexx/SQL program. In particular, this function should be called after a syntax error has been caught with SIGNAL ON SYNTAX.

Arguments:

none

Returns:

success: zero
failure: a negative number

LITEGETINSERTID()

Returns the last unique identifier for a newly inserted row where the column is defined as INTEGER PRIMARY KEY and the value inserted into that column is NULL.

Applicable to SQLite3 databases only.

Arguments:

none

Returns:

success: the unique identifier
failure: a negative number

Example:

```
Call sqlconnect( 'c1', , , 'test.db' )
Call sqlcommand( 't1', 'create table table1 (pkid1 integer primary key, col1 string )'
Call sqlcommand( 't1', 'create table table2 (pkid2 integer primary key, fkid1 integer, col
Call sqlcommand( 'i1', 'insert into table1 values( null, "string value") )'
id = litegetinsertid()
Call sqlcommand( 'i2', 'insert into table2 values( null,' id ', "string value")' )
Call sqldisconnect( 'c1' )
```

MYGETINSERTID()

Returns the last unique identifier for an inserted or updated row where the column is defined as AUTO_INCREMENT.

Applicable to MySQL databases only.

Arguments:

none

Returns:

success: the unique identifier
failure: a negative number

Example:

```
Call sqlconnect( 'c1', 'username', 'password', 'localhost' )
Call sqlcommand( 't1', 'create table table1 (pkid1 integer primary key, col1 string )'
Call sqlcommand( 't1', 'create table table2 (pkid2 integer primary key, fkid1 integer, col
Call sqlcommand( 'i1', 'insert into table1 values( null, "string value") )'
id = mygetinsertid()
Call sqlcommand( 'i2', 'insert into table2 values( null,' id ', "string value")' )
Call sqldisconnect( 'c1' )
```

4. Errors

All functions return a negative number if an error occurred. Zero or positive return values indicate success.

When an error occurs in the Rexx/SQL interface, the function returns a negative number corresponding to one of the numbers below and the variable **SQLCA.INTCODE** is set to that number. The variable **SQLCA.INTERRM** is also set to the corresponding message. If a database error occurs, **SQLCA.SQLCODE** and **SQLCA.SQLERRM** are set to the appropriate values.

Internal Errors:

```
-1  - Database Error
-6  - identifier is too long; max length is n
-7  - value is not a valid integer.
-8  - internal error
-9  - no message available for SQLCODE n
-10 - out of memory
-11 - unknown variable variable.
-12 - variable variable is not settable.
-13 - statement statement is not a query.
-14 - <parameter> is not a valid integer.
-15 - Conversion/truncation occurred on column column: Expecting n, got m
-16 - unable to set Rexx variable
-18 - extraneous argument - argument
-19 - null ("") variable name.
-20 - connection already open with name connection.
-21 - connection connection is not open.
-22 - no connections open.
-23 - statement name omitted or null
-24 - statement statement does not exist
-25 - no connection is current
-26 - statement has not been opened or executed
-27 - reached maximum number of connections: n
-28 - not connected with sufficient privileges
-51 - zero length identifier
-52 - garbage in identifier name
-61 - n bind variables passed. m expected
-62 - bind values must be paired
-63 - invalid substitution variable name at bind pair n.
-64 - invalid datatype datatype specified
-71 - Too many columns specified in SELECT
-75 - no database name supplied
-76 - <connect string> must be only argument
-83 - Column column-name not present in statement
-84 - parameter parameter MUST be supplied
-85 - Column <column> does not have a 'LONG' datatype
-86 - action on file file-name failed: <reason>
-87 - parameter parameter must be <= size
-88 - Column <%s> not present in statement.
-89 - Column <%s> has NULL value; cannot call this function.
-91 - stem name MUST have trailing '.'
-97 - value is not a valid boolean.
-98 - invalid value of "value" for parameter n; value should be one of "value"
```

5. Implementation Notes

To enable multiple database access on those platforms that support the dynamic loading of Rexx external functions, implementation-specific function names and status values should be provided as a compile-time option. It is expected that a separately built library be provided with the *standard* function names together with the a library containing the database platform-specific functions and status values.

For example, the Win32 Oracle implementation provides a dynamic library called **REXXSQL.DLL** which contains the *standard* function names like SQLCONNECT and *standard* status values like **SQLCA.SQLCODE**. It also provides an implementation-specific dynamic library called **REXXORA.DLL** with an equivalent **ORACONNECT** and **ORACA.SQLCODE**. This use of standard and implementation specific names also applies to default statement names and stem variable names. Basically, wherever the string **SQL** appears in function names or Rexx variables names, an implementation specific abbreviation will be used.

This provision of database platform specific external functions will enable access to different vendor databases in the one Rexx program.

The following database-specific abbreviations are used or recommended:

- **ORA** Oracle
- **DB2** IBM DB2
- **SYB** Sybase
- **SAW** Sybase SQL Anywhere
- **MIN** Mini SQL (mSQL)
- **MY** MySQL
- **ODBC** Generic ODBC interface
- **UDBC** Openlink UDBC interface
- **SOL** Solid Server
- **VEL** Velocis (now Birdstep)
- **ING** Ingres
- **WAT** Watcom
- **INF** Informix
- **PRO** Progress
- **POS** PostgreSQL
- **LITE** SQLite
- **EASY** EasySoft ODBC-ODBC Bridge

6. Using Rexx/SQL

SQL statements fall into two broad categories **DDL** and **DML**. **DDL** is Data Definition Language. These are statements like CREATE TABLE, DROP INDEX. DML statements are Data Manipulation Language statements of which there are two forms; queries (SELECT statements) and data modification statements (INSERT, UPDATE and DELETE statements).

To execute any SQL statement the program must first connect to a database server.

Each statement must be executed in a work area or context area.

For **DDL** statements, the underlying steps are:

- allocate a work area
- parse (prepare) the statement
- execute the statement
- release any resources

For **DML** data modification statements, the underlying steps are:

- allocate a work area
- parse (prepare) the statement
- bind any required values to the placemarkers (if any)
- execute the statement
- release any resourcesb

For **DML** query statements, the underlying steps are:

- allocate a work area
- parse (prepare) the statement
- bind any required values to the placemarkers (if any)
- execute the statement
- fetch each row until end of selection (or done)
- release any resources

Since there is a reasonable overhead in allocating work areas and in parsing statements these should be minimised. The Rexx/SQL interface provides the means of doing this. The SQLPREPARE function allocates a work area to a statement and parses the statement. Work areas are deallocated from a statement when the SQLDISPOSE call is issued. While a statement is allocated to a work area it remains prepared (that is parsed and optimised). Because statement names are global, preparing a different statement with the same name as an existing statement disposes the existing one. After a statement has been prepared with SQLPREPARE , it is bound to a work area and remains bound until the statement is disposed of with SQLDISPOSE . The statement can be executed many times by the following means:

- Queries - repeatedly opening and closing the cursor using the functions; SQLOPEN, SQLFETCH and SQLCLOSE. Typically, multiple calls are made to SQLFETCH to retrieve all rows selected in the cursor. SQLCLOSE is optional.
- Data modification statements - repeatedly calling SQLEXECUTE. Each call may supply new bind values. The statement is not reparsed each time.

The following table shows the order in which the database functions are to be called for the different types of SQL statements.

DML		DDL	
SELECT	*INSERT,DELETE etc.*	*CREATE,DROP etc.*	*DESCRIBE*
SQLPREPARE	SQLPREPARE	SQLPREPARE	SQLPREPARE
SQLOPEN	SQLEXECUTE	SQLEXECUTE	SQLDESCRIBE
SQLFETCH (in loop)	SQLDISPOSE	SQLDISPOSE	SQLDISPOSE
SQLCLOSE			
SQLDISPOSE			

Guidelines for Efficient Rexx/SQL

Rexx/SQL provides a couple of different mechanisms with which to execute SQL statements. This section provides some guidelines on what Rexx/SQL functions should be used when. By default use SQLCOMMAND unless otherwise specified below. It is more efficient to call this one Rexx/SQL function than to call the equivalent individual functions described above.

DDL

Always use SQLCOMMAND.

DML query statements

There following are reasons why you might need to consider using the individual Rexx/SQL functions rather than SQLCOMMAND:

1. When you need to execute the same query multiple times with different values of columns in the WHERE clause. See *Other DML Statements* below for more details.
2. When the number of rows expected to be returned is very large. SQLCOMMAND fetches every row from the query into stems for each column. If you are returning a large number of rows this can take quite a long time and use quite a lot of memory for the column contents. Calling SQLFETCH for each row, or fetching a small number of rows, say 100, in each call to SQLFETCH will reduce memory usage. It won't however reduce the time it takes; it will increase it if you eventually return every row.
3. When you don't require the contents of every row in the query. In this case you may have a query that returns many rows, but you are only interested in the first row. Rather than have SQLCOMMAND fetch every row, you can simply call SQLFETCH once to get the contents of the first row of data.

Other DML statements

The most appropriate usage of the individual Rexx/SQL functions is when you need to call the same DML statement repeatedly, but with different values. If you are doing multiple inserts to the one table, it is more efficient to insert those rows with a prepared statement that includes parameter markers, and call SQLEXEC multiple times with the different values for each inserted row. Of course, the database must support placemarkers in SQL statements.

Dynamic Library Implementations

The Rexx external functions in the dynamic library need to be loaded by a call to RxFuncAdd() followed by a call to SQLLOADFUNCS, or its database specific equivalent. eg.

```
            Call RXFuncAdd 'SQLLoadFuncs','rexxsql','SQLLoadFuncs'
            Call SqlLoadFuncs
```

To load all Rexx/SQL external functions using the Oracle specific dynamic library:

```
            Call RXFuncAdd 'ORALoadFuncs','rexxora','ORALoadFuncs'
            Call ORALoadFuncs
```

NB. The case of the function name specified in the third parameter of the RXFuncAdd() function MUST be *exactly* as indicated in the above examples. When using Rexx/SQL on Unix platforms, the second parameter must also be specified in lower case to match the name of the shared library.

Before exiting from a Rexx/SQL program, call SQLDROPFUNCS. This call does not deregister the external functions, rather it frees up all resources used by the current program.

7. Standard placemarkers and bind variables

Most SQL databases provide a mechanism that allows a SQL statement to be prepared once and then executed a number of times with different values for column variables. These variables are generally known as *bind variables*, and the positions in the SQL statement are marked with *placemarkers*. The common *placemarker* is the question mark; **?**. A SQL statement that uses standard placemarkers might look like:

```
query1 = "select name from emp where id = ? and deptno = ?"
```

When providing values for *bind variables* it is necessary not only to provide the value of the *bind variable*, but also the datatype of that *bind variable*. Thus, *bind variables* are always supplied in pairs; the first being the datatype, the second the value.

Assuming the **EMP** table consists of the columns:

ID CHAR(4)
NAME VARCHAR(20)
DEPTNO SMALLINT

then using *bind variables* in a call to SQLCOMMAND would look like:

```
query1 = "select name from emp where id = ? and deptno = ?"
rc = sqlcommand(q1,query1,"CHAR","F1","SMALLINT",10)
```

Some Rexx implementations limit the number of parameters that can be passed to a Rexx external function; (OS/2 Rexx has a limit of 20). To work around this limitation, Rexx/SQL allows the specification of a stem name for the datatype and the value. This stem variable must conform to Rexx's convention of *arrays*; ie. the compound variable with a tail of 0 contains the number of elements in the *array*.

Re-writing the above example using *arrays*:

```
query1 = "select name from emp where id = ? and deptno = ?"
dt.0 = 2
dt.1 = "CHAR"
dt.2 = "SMALLINT"
bv.0 = 2
bv.1 = "F1"
bv.2 = "10"
rc = sqlcommand(q1,query1,"dt.","bv.")
```

Obviously in the above example, with only 2 *bind variables*, it is simpler to use the first method. When using the *array* method, there **must** only be two parameters passed; the two stem variable names, and these stem names **must** include the trailing '.'.

To add a bit more flexibility (and a bit more complication), the format of a *bind variable* can include a reference to an operating system file. This enables Rexx/SQL to insert data into a column directly from a file; useful for storing or retrieving BLOBs. The datatype specification consists of the string 'FILE:' followed by the datatype of the column, and the value portion consists of the operating system file name.

Our example program would now look like:

```
query1 = "select name from emp where id = ? and deptno = ?"
rc = sqlcommand(q1,query1,"FILE:CHAR","./abc","FILE:SMALLINT","/tmp/xyz")
```

Assuming the file: ./abc contains the string "F1" and the file /tmp/xyz contains the string "10" (and no trailing line feed or carriage return characters),then this version of the program would function the same as the previous two.

A word of warning when using this method to insert large files into a database or retrieve BLOBs. Rexx/SQL will take advantage of some database implementations that allow the insertion or extraction of pieces of a column of data. It generally uses a chunk of data of 64kb in size. On database implementations that don't support this partial insertion or extraction of column data, Rexx/SQL has no choice but to insert the file or extract the BLOB in one piece. This requires the allocation of memory of the size of the file or BLOB. So if your file or BLOB is 2gb in size I hope you have a **lot** of memory!!!

Appendix A - Rexx/SQL for Oracle

This section describes features of Rexx/SQL specific to the Oracle implementation.

General:

- All arguments to SQLCONNECT are optional.
- Examples of different connections with SQLCONNECT:

  The following connects to the database running on the local machine as *SCOTT* with password *TIGER*.

  ```
  rc = sqlconnect(,"scott","tiger")
  ```

  The following connects to the database identifed by the SQL*Net V2 entry; *XYZ.WORLD* as *SCOTT* with password *TIGER* with a connection name of *MYCON*.

  ```
  rc = sqlconnect("MYCON","scott","tiger",,"XYZ.WORLD")
  ```

  The following connects to the database running on the local machine as an externally identified user.

  ```
  rc = sqlconnect()
  ```

- If, when the first call to SQLCOMMAND or SQLPREPARE is made, the user is not connected to a database, an implicit SQLCONNECT is made. ie. via the OPS$ feature of Oracle.

Bind Variables:

Rexx/SQL for Oracle can use Oracle's two proprietary forms of placemarkers for bind variables; numbers and names, as well as the *standard* placemarker; **?**, if SQLVARIABLE**('STANDARDPLACEMARKERS',1)** is called before calling SQLCOMMAND or SQLOPEN. See 7. Standard placemarkers and bind variables for further details.

The following describes the Oracle bind mechanisms:

Bind by number:

You cannot bind a LONG or RAW column value with this mechanism. The reason is that there is no mechanism to specify that the input value is a LONG or RAW. You need to either use Standard placemarkers and bind variables or *Bind by name* and specify the bind value type.

The placemarkers in the *sql statement* are numeric; :1, :2 etc. The arguments passed to the SQLCOMMAND and SQLOPEN functions for bind values consist of a '#' followed by the bind values. eg.

```
query1 = "select name from emp where id = :1 and deptno = :2"
rc = sqlcommand(q1,query1,"#",345,10)
```

Bind by name:

The arguments passed to the SQLCOMMAND and SQLOPEN functions are pairs of placemarker name and bind variable value.

The format of the bind variable names is similar to those provided in 7. Standard placemarkers and bind variables, but the bind variable name is appended. Examples of valid bind variable names are:

- FILE:LONG RAW:IMAGE - bind value is LONG RAW column type from a file bound to name :image
- LONG:IMAGE - bind value is LONG column type from Rexx variable bound to name :image
- :IMAGE - bind value not specified from Rexx variable bound to name :image

Some example program would look like:

```
query1 = "select name from my_table where image_name = :name and image = :image"
rc = sqlcommand( q1, query1, "VARCHAR2:name", "mypicture.jpg", "FILE:LONG RAW:image", "/tmp/mypi
```

If a bind variable type is not specified it is assumed to be "VARCHAR2". So in the following example the placemarkers in the *sql statement* named; :PHONE, :ADD are VARCHAR2 values.

```
insert1 = "insert into person values( :PHONE , :ADDRESS )"
rc = sqlcommand(i1,insert1,":PHONE",'+613456123',":ADD",'10 somewhere')
```

Overcoming limitations:

The placemarkers in the *sql statement* are named; :ID, :DEP. The Some Rexx implementations limit the number of parameters that can be passed to a Rexx external function; (OS/2 Rexx has a limit of 20). To work around this limitation, Rexx/SQL allows Oracle bind values and/or placemarkers to be specified as *arrays*. This capability is similar to the method of passing bind values when using *standard* placemarkers but is not as flexible.
To specify that *arrays* are being used to pass bind values and/or placemarkers, the first bind parameter must be a '.' followed by either one or two stem variables. These stem variables must conform to Rexx's convention of *arrays*; ie. the compound variable with a tail of 0 contains the number of elements in the *array*.

The following example demonstrates the use of *arrays* when using numbered placemarkers (only 1 *array* required):

```
query1 = "select name from emp where id = :1 and deptno = :2"
bv.0 = 2
bv.1 = 345
bv.2 = 10
rc = sqlcommand(q1,query1,".","bv.")
```

The following example demonstrates the use of *arrays* when using named placemarkers (two *arrays* required):

```
query1 = "select name from emp where id = :ID and deptno = :DEP"
bn.0 = 2
bn.1 = ":ID"
bn.2 = ":DEP"
bv.0 = 2
bv.1 = 345
bv.2 = 10
```

```
rc = sqlcommand(q1,query1,".","bn.","bv.")
```

Column names:

If a column specification in a SQL statement passed to SQLCOMMAND or SQLPREPARE contains a function or is a constant, the column specifier *must* be aliased so that a valid Rexx variable can be generated for that column.

SQLDESCRIBE variables:

The Oracle implementation does not include any extra variable components.

Appendix B - Rexx/SQL for mSQL

This section describes features of Rexx/SQL specific to the mSQL implementation.

General:

- The *database name* argument in SQLCONNECT is mandatory.
- Examples of different connections with SQLCONNECT:

  The following connects to the *TEST* database running on the local machine with a connection name of *MYCON*.

  ```
  rc = sqlconnect("MYCON",,,"TEST")
  ```

  The following connects to the *MINERVA* database running on the machine *xyz.my.org*.

  ```
  rc = sqlconnect(,,,"MINERVA","xyz.my.org")
  ```

- As mSQL has no concept of a transaction, the functions, SQLCOMMIT and SQLROLLBACK don't do anything. They are included for consistency.
- All statements are actually executed by SQLPREPARE. This obviates the SQLEXECUTE function, but it still should be used for portability.
- Under mSQL 1.x implementations, the variable **SQLCA.ROWCOUNT** always returns 0 for non **SELECT DML** statements. Thus, there is no way of determining how many rows were deleted, updated, or inserted. For mSQL 2.x implementations, **SQLCA.ROWCOUNT** returns the number of rows affected correctly.

Bind Variables:

mSQL has no provision for bind variables in SQL statements. The return value from SQLVARIABLE**('SUPPORTSPLACEMARKERS')** is always **0**, so any references to bind variables in this document should be ignored for mSQL.

Column names:

If a column specification in a SQL statement passed to SQLCOMMAND or SQLPREPARE contains a table alias, eg. a.emp_id, the Rexx variables created corresponding to this column DO NOT contain the "a." prefix.

SQLDESCRIBE variables:

The mSQL implementation includes the extra variable component; **PRIMARYKEY**.

Appendix C - Rexx/SQL for DB2

This section describes features of Rexx/SQL specific to the DB2 implementation.

General:

The DB2 port uses the Call Level Interface (CLI) provided by DB2. This CLI is based on the X/Open CLI and is very similar to the ODBC API.

- The *database name* argument in SQLCONNECT is mandatory.
- Examples of different connections with SQLCONNECT:

  The following connects to the *SAMPLE* database running on the local machine with a connection name of *MYCON*. It is assumed that the user has previously logged on, or will be prompted for a userid and password.

  ```
  rc = sqlconnect("MYCON",,,"SAMPLE")
  ```

  The following connects to the *SAMPLE* database running on the local machine. The Rexx/SQL session will logon as the user *FRED* with a password of *WILMA*.

  ```
  rc = sqlconnect(,'FRED','WILMA',"SAMPLE")
  ```

Bind Variables:

DB2 uses the *standard* placemarker; **?**. See 7. Standard placemarkers and bind variables for further details.

Data types:

The datatypes supported by DB2 can be determined by a call to SQLGETINFO with the *DATATYPES* option.

Appendix D - Rexx/SQL for Sybase System 10/11

This section describes features of Rexx/SQL specific to the Sybase System 10/11 implementation.

This port would not have been possible without the support of Pieter Leemeijer, Sybase, Brisbane and Chuck Moore, DIS, Washington State.

The documentation for this port is incomplete; as is the code!

General:

The Sybase port uses the Sybase Open Client Client-Library/C API, which is unique to Sybase.

- The *username*, *password* and *database name* arguments in SQLCONNECT are mandatory.
- Example of a connection with SQLCONNECT:

  The following connects to the *SADEMO* database running on the local machine with a connection name of *MYCON* and a username of *dba* and password of *sql*.

  ```
  rc = sqlconnect("MYCON","dba","sql","SADEMO")
  ```

Bind Variables:

Sybase System 10/11 uses the *standard* placemarker; **?**. See 7. Standard placemarkers and bind variables for further details.

Data types:

The datatypes supported by Sybase System 10/11 can be determined by a call to SQLGETINFO with the *DATATYPES* option.

Appendix E - Rexx/SQL for Sybase SQLAnyWhere

This section describes features of Rexx/SQL specific to the Sybase SQLAnyWhere implementation.

This port would not have been possible without the support of Pieter Leemeijer, Sybase, Brisbane.

General:

The Sybase SQLAnyWhere port uses the ODBC API.

- The *username*, *password* and *database name* arguments in SQLCONNECT are mandatory.
- Example of a connection with SQLCONNECT:

  The following connects to the *SADEMO* database running on the local machine with a connection name of *MYCON* and a username of *dba* and password of *sql*.

  ```
  rc = sqlconnect("MYCON","dba","sql","SADEMO")
  ```

Bind Variables:

Sybase SQLAnyWhere uses the *standard* placemarker; **?**. See 7. Standard placemarkers and bind variables for further details.

Data types:

The datatypes supported by SQLAnyWhere can be determined by a call to SQLGETINFO with the *DATATYPES* option.

Appendix F - Rexx/SQL for ODBC and UDBC

This section describes features of Rexx/SQL specific to the generic ODBC implementation and to the Openlink UDBC implementation.

General:

- The arguments to SQLCONNECT can be in one of two different combinations:
    1. *username*, *password* and *database name* must all be supplied
    2. Only *host* supplied. *host* in an ODBC connection is actually a **connect string** which takes the form of a number of keyword/value pairs

  otherwise you will get an error -76: *<connect string> must be only argument*. The contents of a **connect string** are dependent on the ODBC driver and/or database.
  The *username* and *password* arguments are required if you have not set these values when you created your Data Source Name(DSN) using ODBC Manager.
  The *database name* argument is the DSN name you specified when you created your DSN using ODBC Manager.
- Examples of connections with SQLCONNECT:

  The following connects to the ODBC/UDBC DSN *REXXSQL* with a username of *scott* and password of *tiger*.

```
rc = sqlconnect( "MYCON", "scott", "tiger", "REXXSQL" )
```

  The following connects to the ODBC/UDBC DSN *REXXSQL* with no username or password. The *REXXSQL* DSN must specify the username and password.

```
rc = sqlconnect( "MYCON", , , "REXXSQL" )
```

  The following connects to the ODBC/UDBC database with a **connect string** only

```
connect_string = "Driver={SQL Server};Server=MyServer;db=pubs;uid=scott;pwd=tiger"
rc = sqlconnect( "MYCON", , , , connect_string )
```

Bind Variables:

ODBC uses the *standard* placemarker; **?**. See 7. Standard placemarkers and bind variables for further details.

Data types:

The datatypes supported by ODBC can be determined by a call to SQLGETINFO with the *DATATYPES* option.

Known errors:

- The Win95/NT ODBC 3.0 drivers sometimes return inconsistent results for queries.

- The Win95/NT Access driver always returns 1 for the *nullable* column when a statement is described.

Appendix G - Rexx/SQL for MySQL

This section describes features of Rexx/SQL specific to the MySQL implementation.

Connection

- The *database name* argument in SQLCONNECT is mandatory.
- Examples of different connections with SQLCONNECT:

  The following connects to the *TEST* database running on the local machine with a connection name of *MYCON*.

  ```
  rc = sqlconnect("MYCON",,,"TEST")
  ```

  The following connects to the *TEST* database running on the machine *xyz.my.org*.

  ```
  rc = sqlconnect(,,,"TEST","xyz.my.org")
  ```
- If using a non-standard TCP port (not 3306) for your MySQL server, you can change the port number that Rexx/SQL uses by settingthe environment variable; MYSQL_TCP_PORT to the port number required. One way is to:

  ```
  Call Value 'MYSQL_TCP_PORT', 3307, 'ENVIRONMENT'
  ```

  before calling SQLCONNECT.

General:

- All statements are actually executed by SQLPREPARE. This obviates the SQLEXECUTE function, but it still should be used for portability.

Transactions:

MySQL has support for transactions, but only on BDB or INNOBASE tables. The return value from SQLVARIABLE**('SUPPORTSTRANSACTIONS')** is always **1**, but this doesn't necessarily mean that a transaction will be committed or rolled back when you use the SQLCOMMIT or SQLROLLBACK functions. SQLROLLBACK will result in a warning (a positive number) if you attempt to rollback a transaction that updates a table which resides in neither BDB or INNOBASE.

Bind Variables:

MySQL supports bind variables in SQL statements, however, Rexx/SQL currently doesn't. The return value from SQLVARIABLE**('SUPPORTSPLACEMARKERS')** is always **0**, so any references to bind variables in this document should be ignored for MySQL.

Column names:

If a column specification in a SQL statement passed to SQLCOMMAND or SQLPREPARE contains a table alias, eg. a.emp_id, the Rexx variables created corresponding to this column DO NOT contain the "a." prefix.

SQLDESCRIBE variables:

The MySQL implementation includes the extra variable component; **PRIMARYKEY**.

SQLTABLES behaviour:

The MySQL implementation has the following behaviour:

- The ***table qualifier*** argument is ignored. Only tables from the connected MySQL database are returned.
- The ***table owner*** argument is ignored.
- The ***table type*** argument is ignored.
- The ***table name*** argument is optional, but can have wildcards.

SQLCOLUMNS behaviour:

The MySQL implementation has the following behaviour:

- The ***table qualifier*** argument is ignored. Only tables from the connected MySQL database are returned.
- The ***table owner*** argument is ignored.
- The ***table type*** argument is ignored.
- The ***table name*** argument is mandatory and cannot have wildcards.

Appendix H - Rexx/SQL for Solid Server

This section describes features of Rexx/SQL specific to the Solid Server implementation.

General:

- The *username*, *password* and *database name* arguments in SQLCONNECT are mandatory.
- Examples of connections with SQLCONNECT:

  The following connects to the Solid Server DSN *REXXSQL* with a username of *scott* and password of *tiger*.

  ```
  rc = sqlconnect("MYCON","scott","tiger","REXXSQL")
  ```

  The following connects to the Solid Server DSN *REXXSQL* with no username or password. The *REXXSQL* DSN must specify the username and password.

  ```
  rc = sqlconnect("MYCON","","","REXXSQL")
  ```

Bind Variables:

Solid Server uses the *standard* placemarker; **?**. See 7. Standard placemarkers and bind variables for further details.

Data types:

The datatypes supported by Solid Server can be determined by a call to SQLGETINFO with the *DATATYPES* option.

Appendix I - Rexx/SQL for SQLite3

This section describes features of Rexx/SQL specific to the **SQLite** implementation. Support is no longer offered for SQLite prior to version 3.

General:

- All arguments to SQLCONNECT are optional. If no *database name* argument is supplied, then the SQLite3 feature of an in-memory database will be used. With this feature the database is created on connection and all tables need to be created and can then be manipulated as one would normally. The data in the database is lost when disconnecting.
  The *database name* argument in SQLCONNECT is the only argument that can be supplied. All other arguments should not be specified. The *database name* argument is a file name. If the database in the specified file name does not exists, it will be created.
- SQLite only allows tables to be described; not a SELECT statement. Therefore is you want to use the SQLDESCRIBE function, pass a table name as the parameter, not a select statement.

Bind Variables:

SQLite supports standard placemarkers and SQLite specific placemarkers. Need to document how these work; similar to Oracle.

Transactons:

SQLite supports transactions, but you need to explicitly execute a "BEGIN TRANSACTION" command to start a transaction. See *tester.cmd* sample program for an example.

Data types:

SQLite is a typeless database. All columns are stored as text, so SQLite3 allows any datatype you want. Rexx/SQL for SQLite3 however, supports the following datatypes explicitly:

- INTEGER PRIMARY KEY
- INTEGER
- INT
- FLOAT
- NUMERIC
- CHARACTER
- CHAR
- TEXT
- VARCHAR
- BOOLEAN
- BLOB
- CLOB
- TIMESTAMP
- DATETIME

The datatypes supported by Rexx/SQL for SQLite can be determined by a call to SQLGETINFO with the *DATATYPES* option.

Frequently Asked Questions

Q: I use a database function in the column list of a SELECT statement. How do I retrieve the value of the column ?

A: Most databases have the capability for columns in a SELECT statement to be given an alias. Suppose you try to call SQLCOMMAND as:

```
        Call SQLCommand 'q1', 'SELECT count(*) FROM MYTABLE'
```

Rexx/SQL will try and set the variables: Q1.COUNT(*).0 and Q1.COUNT(*).1. Clearly these variables will not be available in your Rexx program. Instead use a column alaias. There are generally two syntaxes for this. Both are shown below:

```
        Call SQLCommand 'q1', 'SELECT count(*) AS mycount FROM MYTABLE'
        Call SQLCommand 'q1', 'SELECT count(*) mycount FROM MYTABLE'
```

Rexx/SQL will then set the variables: Q1.MYCOUNT.0 and Q1.MYCOUNT.1.

Q: When I download Rexx/SQL binaries, or build Rexx/SQL from source I get 2 DLLs and 2 executables. Why ?

A: If you are accessing a database from a single vendor, such as Oracle, then you would usually use the external functions that start with **SQL**, rather than the external functions that start with **ORA**.
See 5. Implementation Notes for more information.
REXXSQL.DLL/librexxsql.so contains the external functions that start with **SQL**.
REXXORA.DLL/librexxora.so contains the external functions that start with **ORA**.

Q: Does Rexx/SQL support MS SQL Server or MS Access?

A: Yes, the Windows ODBC port (rxsqlxx_odbc_w32.zip) supports SQL Server and Access. You can even manipulate Excel spreadsheets with Rexx/SQL.

History of Rexx/SQL

This section provides details of changes and additions made to the Rexx/SQL interface as it evolves.

Version 2.5: 15 Oct 2006

- Added SQLDataSource(), SQLTables() and SQLColumns(). Documentation and original ODBC implementation courtesy of Lorne Sunley. Not all of these functions completed on all platforms. tester.cmd changed to test these functions.
- Rexx/SQL is now thread-safe. Thread-safety of a Rexx/SQL application is dependent now on whether the database client is thread-safe. Where it is known that the database client is thread-safe, the new Rexx/SQL variable; SUPPORTSTHREADS is set to 1.
- Fixed bug with SQLCA.ROWCOUNT when calling SQLFETCH; it now sets the value as documented.
- Updated documentation to state that SQLCA. stem is read-only; don't change values or DROP the variables. You have been warned!
- Added support for SQLite3. Support for placemarkers in SQLite3 is now supported. NB Support for version of SQLite before 3.x is not provided. If you used a beta of Rexx/SQL 2.5 that included "lite3" in the distribution filename or in the external function names, remove the "3". ie if you used LITE3GETINSERTID it is now called LITEGETINSERTID.
- Fixed bug with following sequence of calls:

  ```
  SQLPREPARE
    SQLOPEN
      SQLFETCH (in loop)
    SQLCLOSE
    SQLOPEN
      SQLFETCH (in loop)
    SQLCLOSE
  SQLDISPOSE
  ```

  Previously, the second SQLOPEN/SQLFETCH/SQLCLOSE sequence would not work on all database platforms.
  Added test in tester.cmd "fetch" to test this case.
- Added MYGETINSERTID for MySQL to return the value generated for an AUTO_INCREMENT column as a result of the last INSERT or UPDATE statement.
- Added LITEGETINSERTID for SQLite3 to return the last identifier from an INSERT of NULL into a column defined as INTEGER PRIMARY KEY.
- Added support for Open Object Rexx.
- Fixed bug with Oracle and SQLGETDATA. Now no longer crashes.
- Added extra bind variable capability to Oracle bind by name. Allows LONG data to be bound from a Rexx variable or a file; similar to bind capability with standard placemarkers.
- Support for Sybase SQL Anywhere (now called Adaptive Server Anywhere) on Un*x is not working due to major changes by Sybase.

Version 2.4: 7 Jul 2002

- Added a default subcommand handler, so that Operating System commands in a Rexx/SQL program NOT specifically addressed to the system with ADDRESS SYSTEM, will execute properly.

- MySQL port updated to work around introduced incompatibility with SELECT statements in MySQL 3.22.24.
- MySQL port now supports transactions if using BDB or INNOBASE tables. The sample program tester.cmd builds its test tables with BDB, so if your installation of MySQL does not support BDB, remove the two lines in tester.cmd like:

  ```
  If db = 'MYSQL' Then create1 = create1 'TYPE = BDB'
  ```
- DB2 port now links with **db2** library rather than **db2ar** library.
- Fixed bug with Oracle DATE datatype support for non-US installations
- Fixed bug with DB2 and ODBC datatype checking
- Added support for Object Rexx under AIX
- Added support for uni-REXX
- Added support for BirdStep (Raima) Velocis database servers
- Added support for unixODBC project (http://genix.net/unixODBC)
- Added support for SQLite; an embedded SQL database engine (no server required) (http://www.sqlite.org)
- Win32 ports now use Rexx/Trans DLL so only one Rexx/SQL required for Regina, Object Rexx, Personal Rexx and WinRexx.
- Support for Solid Server 2.3 included; now no longer need to determine operating system specific libraries.
- REXX/6000 support more robust
- Added multiple database error message capture for those databases that can return multiple lines of errors. This is handled by addition of an "array"; sqlca.sqlerrm.0 to sqlca.sqlerrm.n where n is sqlca.sqlerrm.0. There is also a sqlca.sqlstate "array"
- Database APIs that are based on ODBC or CLI, that can return a "success with info" status, are also supported. The return code from Rexx/SQL functions are now:

  ```
      0  - success
      0 - success with info (the "info" is in sqlca.sqlerrm array)
  ```

  SQLFETCH is an exception to the above rule as a positive return value indicates how many rows were fetched.
- Added PostgreSQL 6.4 support using iODBC.
- Support for mSQL (2.0+) under OS/2 using Netlabs port added.
- MySQL port under OS/2 added.
- Changes parameter passing convention for rexxsql executable. It now behaves the same way as the Rexx interpreter for multiple parameters.
- Added extra method for setting internal DEBUG variable, by setting environment variable REXXSQL_DEBUG.
- Added extra internal debugging flag. Invoke by calling SQLVARIABLE 'DEBUG', 4. Can also be set by passing -D flag to command line, or setting environment variable REXXSQL_DEBUG to 4.
- Changed method of specifying Rexx Interpreter and Database vendor. See INSTALL for more detailed expalanation.

Version 2.3: 26 Jun 1998

- Fixed major memory leaks with all ports.
- Changed the Win32 ports for Personal Rexx and Enterprise Rexx so they would work with the rexxsql.dll. They way that the interpreters handle external functions, required special initialisation and

termination code :-(
- Split this document up into pieces.

Version 2.2: 10 Dec 1997

- Addition of DB2/NT and SQL Anywhere for Unix ports.
- Support for Object Rexx for Linux.

Version 2.1: 23 Sep 1997

- Addition of UDBC and Solid Server ports.
- Added support for mSQL 2.x.

Version 2.0: 10 Jun 1997

- Addition of DB2, SQL Anywhere, MySQL and ODBC ports.
- Updated support for mSQL 1.0.16.
- Addition of new settable variables, AUTOCOMMIT, IGNORETRUNCATE, NULLSTRINGOUT,NULLSTRINGIN and STANDARDPLACEMARKERS and readonly variables DESCRIBECOLUMNS, SUPPORTSPLACEMARKERS, SUPPORTSSQLGETDATA and SUPPORTSTRANSACTIONS for use with SQLVARIABLE
- Addition of new functions use with SQLGETINFO and use with SQLGETDATA
- Changed the Oracle port to use deferred parsing. This necessitated changing the way that execution of **DDL** statements are handled. In previous versions, it was possible to call SQLPREPARE and the **DDL** statement would be executed as well. The new (and more consistent) method involves calling SQLEXECUTE after preparing the statement. This change to execution of **DDL** statements is applicable to **all** Rexx/SQL ports. Added support to the Oracle port to enable the use of '?' as a bind variable placemarker, to be more consistent with other databases.
- Added new command line switch **-i** to allow Rexx/SQL to be run interactively or to be used as a filter under Un*x.
- Added support for bind values to be passed in Rexx *arrays*.

Version 1.3: 10 March 1996

- Addition of OS/2 port for mSQL (version 1.0.13), using Dirk Ohme's OS/2 port.
- Parameters to SQLCONNECT changed.

Version 1.2: sometime in 1995

- Addition of mSQL (version 1.0.10) support under Un*x.

Version 1.1: sometime in 1995

- Addition of DLL support under OS/2 for Oracle.

Version 1.0: 10 June 1995

- First public release. This consisted of the Oracle port for Un*x and OS/2, but no DLL support under OS/2.