RDM Embedded 9.1

SQL Language Guide

Trademarks

Contents

1. Examining ODBC

1.1 General Introduction

This manual provides information on how to use Birdstep's RDM Embedded SQL. This chapter describes how to use the ODBC C Application Programming Interface (API) and presents other ODBC-related information.

The SQL API added to RDM Embedded exists in a separate module such that programmers can use the native d_ API or the SQL API in separate applications. The SQL API conforms to ODBC 3.5. The list of implemented functions is shown later in this chapter (followed by the list of ODBC 3.5 functions not implemented).

The chapter provides the following:

1. An overview of the subset of ODBC version 3.5 that is available in RDM Embedded SQL ODBC
2. A discussion of database open modes
3. Programming guidelines for RDM Embedded SQL

1.2 Calling RDM Embedded ODBC functions

1.2.1 Overview of the RDM Embedded ODBC API

ODBC is by far the most widely used programming access interface to SQL databases. Thus a subset of ODBC is the standard of choice for access to a database residing on an ODBC data source. The tables below list the ODBC functions available in the RDM Embedded implementation, and the ODBC 3.5 functions that are not available. (Full descriptions of the implemented functions are contained in the Microsoft ODBC Programmer's Guide.)

The ODBC 3.5 functions supported in RDM Embedded are listed and defined in the following table.

Function	Description
SQLAllocHandle	Allocate connection or statement handle.
SQLBindCol	Bind a select statement result column value to a variable.
SQLBindParameter	Bind a variable to a SQL statement parameter value.
SQLCloseCursor	Close an open cursor.
SQLConnect	Connect to a data source.
SQLDescribeCol	Get a description of a select statement result column.
SQLDescribeParam	Get a description of a SQL statement parameter marker.
SQLDisconnect	Disconnect from a data source.
SQLEndTran	Commit or rollback the current transaction.
SQLExecDirect	Prepare and execute a SQL statement.
SQLExecute	Execute a previously prepared SQL statement.

Function	Description
SQLFetch	Fetch the next result row from an executed select statement.
SQLFreeHandle	Free connection or statement handle.
SQLGetCursorName	Get cursor name associated with the specified statement handle.
SQLGetDiagField	Get the requested current diagnostic value for the specified field.
SQLGetDiagRec	Get all the current diagnostic information.
SQLNumParams	Get the number of parameters in a prepared SQL statement.
SQLNumResultCols	Get the number of result columns in a prepared select statement.
SQLPrepare	Prepare (compile) a SQL statement.
SQLRowCount	Get the number of rows updated by the update or delete statement.
SQLSetConnectAttr	Sets attributes that govern aspects of connection.
SQLSetCursorName	Set the cursor name to be associated with the specified statement handle.
SQLSetStmtAttr	Sets attributes related to a statement.

Note that many of the non-supported functions in ODBC are needed to communicate driver-specific implementation details to the ODBC driver manager. Since no driver manager is used for RDM Embedded SQL, these functions are unnecessary.

ODBC 3.5 functions not supported in RDM Embedded are listed in the table below.

SQLBrowseConnect	SQLBulkOperations	SQLCancel
SQLColAttribute	SQLColumnPrivileges	SQLColumns
SQLCopyDesc	SQLDataSources	SQLDriverConnect
SQLDrivers	SQLExtendedFetch	SQLFetchScroll
SQLForeignKeys	SQLFreeStmt	SQLGetConnectAttr
SQLGetData	SQLGetDescField	SQLGetDescRec
SQLGetEnvAttr	SQLGetFunctions	SQLGetInfo
SQLGetStmtAttr	SQLGetTypeInfo	SQLMoreResults
SQLNativeSql	SQLParamData	SQLPrimaryKeys
SQLProcedureColumns	SQLProcedures	SQLPutData
SQLSetDescField	SQLSetDescRec	SQLSetEnvAttr
SQLSetPos	SQLSpecialColumns	SQLStatistics
SQLTablePrivileges	SQLTables	

1.2.2 Setting the database open mode

The SQLSetConnectAttr function is used to set database open modes. If the open mode is not specified, or if this function is not called, the default mode is Shared. The other mode options allow databases to be opened in One User or Exclusive modes. See the table below for a description of the modes. Go to Chapter 3 of the RDM Embedded Reference Manual and refer to the d_open function for further details.

Type	Description

"s"	Shared access mode. Multiple users can be accessing database at the same time.
"x"	Exclusive access mode. Only one user can access the database. All others will be locked out.
"o"	One-user-only mode. Only one user will be using specified database. The application will attempt to lock out all other users.

For multi-user operation of the database and use of the lock manager, the lock manager name and type must be set before the database is opened. The settings are made via the name and type entries in the rdm.ini file.

1.3 RDM Embedded SQL programming guidelines for ODBC

Use of the RDM Embedded SQL C functions is illustrated in this section through programming examples. The programming functionalities presented include:

1. Data source connection
2. Basic statement processing
3. Date/time data retrieval
4. Parameter marker usage
5. Premature statement termination
6. Cursors
7. Positioned updates and deletes

1.3.1 Connecting to a data source

To establish connections to a data source from an application program, perform the following steps:

1. Call SQLAllocHandle to allocate a connection handle.
2. Call SQLConnect to connect to a data source.

To set AutoCommit mode, call SQLSetConnectAttr any time after allocating the connection handle.

To set database access mode, SQLSetConnectAttr must be called prior to SQLConnect.

To end your database session, perform the following basic steps:

1. Call SQLDisconnect to terminate the client connection to RDM Embedded.
2. Call SQLFreeHandle to free the connection or statement handle.

The example below illustrates the use of these function calls. The SQLSetConnectAttr function sets the number of seconds to wait for an SQL statement to execute before returning to the application..

Note that the type definitions for the connection handle is declared in the standard header file **rdm.h**. Also note the use of the constant **SQL_NTS** for the length argument in the call to SQLConnect to indicate that each of the char arguments is a standard C null-terminated string.

```
#include "rdm.h"
HDBC ch;   /* connection handle */
HSTMT sh;  /* statement handle */

       ...

/* connect to RDM Embedded SQL */
SQLAllocHandle(SQL_HANDLE_LOCAL_DBC, SQL_NULL_HANDLE, &ch);



stat = SQLSetConnectAttr(ch, SQL_DBMS_ACCESS_MODE, (SQLPOINTER) "x", SQL_NTS);



stat = SQLConnect(ch, "sales;invntory", SQL_NTS, "user1", SQL_NTS, NULL, 0);


if (stat != SQL_SUCCESS ) return ( ErrHandler() );


...           /* run sales & invntory db application */



SQLDisconnect(ch);
SQLFreehandle(ch);
```

You can establish more than one connection to RDM Embedded SQL if needed; each connection is maintained in a separate task context.

1.3.2 Basic statement processing with ODBC

The basic steps for processing SQL statements are as follows:

1. Call SQLAllocHandle to allocate a statement handle.
2. Call SQLPrepare to compile the statement.
3. Call SQLExecute to execute the statement.

Steps 2 and 3 can be combined into a single call to SQLExecDirect if appropriate. If the statement is a select statement, include the following steps:

1. Call SQLBindCol to bind column results to host program variables.
2. Call SQLFetch to retrieve each row of the result set.

When completed, the allocated statement handle can be one of the following:

1. Re-executed (without having to recompile the statement)
2. Reused with another statement
3. Freed

The example below shows a simple command execution sequence that opens the sales and inventory databases. It consists of a call to SQLAllocHandle to allocate a statement handle, a call to SQLExecDirect to compile and execute the (non-select) SQL statement string passed in through stmt, and a call to SQLFreeHandle to free the used statement handle. Function ExecStmt assumes the connection handle is valid (if not valid, SQLAllocHandle will return the error code SQL_INVALID_HANDLE).

```
#include "rdm.h"


SQLRETURN ExecStmt(HDBC hdbc, char *stmt)
{
        SQLRETURN stat;
        HSTMT hstmt;


stat = SQLAllocHandle(SQL_HANDLE_STMT , hdbc, &hstmt);
        if ( stat == SQL_SUCCESS ) {
                stat = SQLSetStmtAttr(hstmt, SQL_ATTR_QUERY_TIMEOUT, (SQLPOINTER) 30,
                SQL_IS_NOT_POINTER);
                        stat = SQLExecDirect(hstmt, stmt, SQL_NTS);
                        SQLFreeHandle(SQL_HANDLE_STMT, hstmt);
        }
        return stat;
}
```

The next example shows these basic steps in the processing of a select statement that summarizes the year-to-date total sales for each salesperson. Function SalesSummary is called with the open connection handle to the sales database. This function allocates its own statement handle and calls SQLExecDirect to compile and execute the select statement.

Calls to SQLBindCol bind the two column results to character buffers (sale_name and amount) declared in the function. The sizes of these buffers are passed (note that SQL_NTS would be invalid as these are output and not input buffers) as well as SQL_C_CHAR, indicating that the result is to be converted to a character string. The last argument to SQLBindCol is the address of a long (SQLINTEGER) variable to contain the output result length. In this example this final argument is NULL, indicating that the program does not need the result length.

Each row of the result set is retrieved through calls to SQLFetch. Each SQLFetch call fetches the next row and stores the column results in the program locations specified in the SQLBindCol calls.

```
 #include "rdm.h"
```

```
char stmt[] =
       "select sale_name, sum(amount) from salesperson, customer,
                        sales_order "
       "where salesperson.sale_id = customer.sale_id "
                        "and customer.cust_id = sales_order.cust_id "
       "group by sale_name";
 SQLRETURN SalesSummary(
       HDBC hdbc) /*connection handle to sales database Server*/
{
       char sale_name[31]; /* salesperson name */
       char amount[20]; /* formatted sales order amount */
       HSTMT sh; /* statement handle */
               SQLRETURN stat; /* SQL status code */


       stat = SQLAllocHandle(SQL_HANDLE_STMT, hdbc, &sh);
       if ( stat != SQL_SUCCESS ) return(stat);
       if ( (stat = SQLExecDirect(sh, stmt, SQL_NTS)) == SQL_SUCCESS ) {
               SQLBindCol(sh, 1, SQL_C_CHAR, sale_name, 31, NULL);
               SQLBindCol(sh, 2, SQL_C_CHAR, amount, 20, NULL);
               while ( (stat = SQLFetch(sh)) == SQL_SUCCESS ) {
                       printf("Acct manager %s has a total of $%s in orders\n",
                       sale_name, amount);
               }
       }
       return stat;
}
```

The next example uses calls to SQLNumResultCols and SQLDescribeCol to dynamically allocate the memory needed to contain the column result data. Function PrintTable outputs all of the columns and rows contained in the table identified by tabname, which is contained on the connection associated with hdbc. Unlike the prior example, which had a fixed number of columns in the result set, this example can have a varying number of result columns. As such, the program calls SQLNumResultCols to get the number of columns in the result set. An array of column result descriptors (COL_RESULT *cols) is allocated to contain the definition and result information for each column. Function SQLDescribeCol returns the name, type, and display size for each column. The result value buffer is dynamically allocated and bound to its result column through the call to SQLBindCol.

As each row is returned from SQLFetch, each column value and its result length is stored in the COL_RESULT container for that column. A column value that is null will have its length value set to SQL_NULL_DATA. As you can see in the example, the program displays "NULL" in this case.

```
#include "rdm.h"


/* Result container */
```

```
typedef static struct {
       SQLCHAR name[33]; /* column name */
       void *value; /* column value */
       SQLSMALLINT type; /* column type */
       SQLINTEGER len; /* result value length */
} COL_RESULT;


/* Print all rows of a table */
SQLRETURN PrintTable(
       HDBC hdbc, /* connection handle */
       char *tabname) /* name of table whose rows will be printed */
{
       char stmt[80];
       HSTMT sh;
       SQLSMALLINT tot_cols;
       COL_RESULT *cols;
       SQLINTEGER size;
       long row;


/* Set up and compile select statement */
   sprintf(stmt, "select * from %s", tabname);
   SQLAllocHandle(SQL_HANDLE_STMT, hdbc, &sh);
   if ( SQLExecDirect(sh, stmt, SQL_NTS) != SQL_SUCCESS )
       return ErrHandler();


/* Allocate column results container */
   SQLNumResultCols(sh, &tot_cols);
   cols = (COL_RESULT *)calloc(tot_cols, sizeof(COL_RESULT));


/* Fetch column names and bind column results */
   for ( i = 0; i < tot_cols; ++i ) {
       SQLDescribeCol(sh, i+1, cols[i].name, 33, NULL,
                      &cols[i].type, &size, NULL, NULL);
       cols[i].value = malloc(size+6);
       SQLBindCol(sh, i+1, SQL_C_CHAR, cols[i].value, size+1,
                      &cols[i].len);
   }

/* Print all rows in record-oriented format */
   printf("========== %s ==========", stmt);
   for (row = 1; SQLFetch(sh) == SQL_SUCCESS; ++row ) {
       printf("**** row %ld:\n", row);
       for (i = 0; i < tot_cols; ++i)
                      printf(" %32.32s: %s\n", cols[i].name,
```

```
                                cols[i].len == SQL_NULL_DATA ? "NULL" :
                                               cols[i].value);
    }

  /* Drop statement handle and free allocated memory */
     SQLFreeHandle(SQL_HANDLE_STMT, sh);
     for (i = 0; i < tot_cols; ++i)
         free(cols[i].value);
     free((void *)cols);  return SQL_SUCCESS;
    }
```

1.3.3 Retrieving date/time data

RDM Embedded provides support for the ODBC date, time, and timestamp data types. Database columns of those types can be returned in struct variables of type DATE_STRUCT, TIME_STRUCT, or TIMESTAMP_STRUCT. These structure types are declared in rdm.h as shown below.

```
struct tagDATE_STRUCT {
       SQLSMALLINT year; /*year (>= 1 A.D., e.g., 1993) */
       SQLUSMALLINT month; /*month number: 1 to 12 */
       SQLUSMALLINT day; /*day of month: 1 to 31 */
} DATE_STRUCT;


struct tagTIME_STRUCT {
       SQLUSMALLINT hour; /* hour of day: 0 to 23 */
       SQLUSMALLINT minute; /* minute of hour: 0 to 59 */
       SQLUSMALLINT second; /* second of minute: 0 to 59 */
} TIME_STRUCT;


struct tagTIMESTAMP_STRUCT {
       SQLSMALLINT year; /* year (>= 1 A.D., e.g., 1993) */
       SQLUSMALLINT month; /* month number: 1 to 12 */
       SQLUSMALLINT day /* day of month: 1 to 31 */;
       SQLUSMALLINT hour; /* hour of day: 0 to 23 */
       SQLUSMALLINT minute; /* minute of hour: 0 to 59 */
       SQLUSMALLINT second; /* second of minute: 0 to 59 */
       SQLUINTEGER fraction; /*billionths of a second:  0 to 999,900,000
                          (RDM Embedded SQL accurate to 4 places only) */
} TIMESTAMP_STRUCT;
```

Use of date and time data is shown in the example below, which prints the year-to-date sales orders for a particular customer. In this example, SQLBindCol is called to request the column values in their native data type.

```
#include "rdm.h" /*Needed for ODBC date/time extensions*/


static char stmt[] =
       "select ord_num, ord_date, ord_time, amount from sales_order "
       "where cust_id = ";


SQLRETURN CustomerOrders(
       HDBC hdbc,     /* Connection handle to sales database */
       char *cust_id) /* Cust_id whose orders are to be printed */
{
       char orders[80];
       short ord_num;
       DATE_STRUCT ord_date;
       TIME_STRUCT ord_time;
       double amount;
       SQLRETURN stat;
       HSTMT hstmt;


       stat = SQLAllocHandle(SQL_HANDLE_STMT, hdbc, &hstmt);
       if ( stat != SQL_SUCCESS ) return(stat);
       /* Construct select statement extracting this customer's orders */
       strcpy(orders, stmt);
       strncat(orders, cust_id, 3);


       /* Compile, execute and bind result columns */
       SQLExecDirect(hstmt, orders, SQL_NTS);
       SQLBindCol(hstmt, 1, SQL_C_SHORT, &ord_num, 0, NULL.);
       SQLBindCol(hstmt, 2, SQL_C_DATE, &ord_date, 0, NULL.);
       SQLBindCol(hstmt, 3, SQL_C_TIME, &ord_time, 0, NULL.);
       SQLBindCol(hstmt, 4, SQL_C_DOUBLE, &amount, 0, NULL.);


printf("ORDERS FOR CUSTOMER ID: %s\n", cust_id);
   while ( (stat = SQLFetch(hstmt)) == SQL_SUCCESS ) {
              printf("order number : %d\n", ord_num);
              printf("order date : %02d/%02d/%02d\n",
                             ord_date.month, ord_date.day, ord_date.year-1900);
              printf("order time : %02%02 hours\n", ord_time.hour,
                             ord_time.minute);
              printf("order amount : $%.2f\n\n", amount);
       }
       SQLFreeHandle(SQL_HANDLE_STMT, hstmt);
       return stat;
}
```

1.3.3.1 Using parameter markers

Statements can be compiled once (by calling SQLPrepare) and executed multiple times (calling SQLExecute). To make this more practical, you can embed parameter markers in your statements in the place of literal constants. The SQLBindParameter function is called to bind a host program variable to a parameter marker. Each time you call SQLExecute for that statement, the values from those host program variables are substituted for the parameter markers.

The program in the next example uses parameter markers with an insert statement to insert rows into the product table from data contained in a text file. The values are extracted from each line of the text file using a convenient, user-created function called GetValues. This function extracts the data from a comma-delimited ASCII file format, converts the data to its native representation based on a print format string, and stores the results in the locations specified in the subsequent arguments.

The SQLExecute call executes the insert statement using the new parameter values.

```
#include <stdio.h>
#include <stdlib.h>
#include "rdm.h"


char insert[] =
       "insert into product(prod_id, prod_desc, price, cost)
               values(?,?,?,?)";

HDBC hdbc;
HSTMT hstmt;


short prod_id;
char prod_desc[40];
FLOAT price, cost;


void main()
{
       FILE *txtFile;
       char sqlstate[6], emsg[80];
       char user[15], pw[8];
       int lineno;
       UWORD txtype;


       txtFile = fopen("product.asc", "r");
       if ( ! txtFile ) abort("unable to open file\n");
       SQLAllocHandle(SQL_HANDLE_LOCAL_DBC, SQL_NULL_HANDLE, &hdbc);
```

```
        /* Connect to invntory database */
        stat = SQLConnect(hdbc, "invntory", SQL_NTS, "user2",
                SQL_NTS, NULL, 0);
        if (stat != SQL_SUCCESS ) goto quit;


        SQLAllocHandle(SQL_HANDLE_STMT, hdbc, &hstmt);
        SQLPrepare(hstmt, insert, SQL_NTS);
        SQLBindParameter(hstmt, 1, SQL_PARAM_INPUT, SQL_C_SHORT,
                SQL_SMALLINT, 0, 0, &prod_id, 0, NULL);
        SQLBindParameter(hstmt, 2, SQL_PARAM_INPUT, SQL_C_CHAR,
                        SQL_CHAR, 39, 0, prod_desc, 40, NULL);
        SQLBindParameter(hstmt, 3, SQL_PARAM_INPUT, SQL_C_FLOAT,
                SQL_FLOAT, 0, 0, &price, 0, NULL);
        SQLBindParameter(hstmt, 4, SQL_PARAM_INPUT, SQL_C_FLOAT,
                SQL_FLOAT, 0, 0, &cost, 0, NULL);


        for (lineno = 1;
                GetValues(txtFile, "%d,%s,%lf,%lf", &prod_id, prod_desc,
                                &price, &cost); ++lineno ) {
                stat = SQLExecute(hstmt);
                if ( stat != SQL_SUCCESS ) break;
        }
quit:
        if ( stat == SQL_SUCCESS )
                txtype = SQL_COMMIT;
        else {
TMT, hstmt, 1, sqlstate,
                SQLGetDiagRec(SQL_HANDLE_S
                                NULL, emsg, 80, NULL); ERROR(%s): %s\n", lineno,
                        printf( "***Line %d -
                        sqlstate, errmsg);       txtype = SQL_ROLLBACK;
        }
        /* Commit or rollback transaction */
        SQLEndTrans(SQL_HANDLE_DBC, hdbc, txtype);
}
```

Note in the example above the call to function SQLGetDiagRec to retrieve the sqlstate code and error message, in the event that SQLExecute unsuccessfully returns.

This example also illustrates the use of SQLEndTran to commit or roll back the changes based upon whether or not an error occurred.

1.3.3.2 Premature statement termination

Often a user will browse through a large result set until the desired rows have been found, at which point the remaining results are no longer needed. To terminate processing of a select statement before the last row has been fetched, you can call SQLCloseCursor as shown in the example below.

```
#include "rdm.h"
                . . .
/* print all rows of a table */
int PrintTable(
       HDBC hdbc, /* connection handle */
       char *tabname) /* name of table whose rows are to be printed */
{
       char stmt[80];
       HSTMT sh;
       . . .
       /* set up and compile select statement */
       sprintf(stmt, "select * from %s", tabname);
       SQLAllocHandle(SQL_HANDLE_STMT, hdbc, &sh);
       if ( SQLExecDirect(sh, stmt, SQL_NTS) != SQL_SUCCESS )
              return ErrHandler();
       . . .
       /* print all rows in table */
       while ( SQLFetch(sh) == SQL_SUCCESS ) {
                       . . .
       if ( cancelled_by_user )
                      SQLCloseCursor(sh);
                      . . .
       }
       SQLFreeHandle(SQL_HANDLE_STMT, &sh);
       return SQL_SUCCESS;
}
```

1.3.3.3 Cursors and positioned updates and deletes

A cursor is a named, updateable select statement in which the cursor position is the current row (that is, the row returned from the most recent call to SQLFetch). An updateable select statement is one that does not include a group by or order by clause, and only refers to a single table in the from clause. Cursors are used in conjunction with positioned updates and deletes to enable the current row from a select statement to be updated or deleted.

The general procedure for performing a positioned update or delete is as follows:

1. Call SQLAllocHandle to allocate a statement handle for the select statement (that is, the cursor).
2. Call SQLAllocHandle to allocate a statement handle for the update or delete statement.
3. Call SQLPrepare with the first statement handle to compile the select statement.

4. If you want to specify your own cursor name, call SQLSetCursorName using the select statement handle.
5. With the select statement handle, call SQLBindCol and SQLBindParameter as necessary and then call SQLExecute. If you want to use a system-generated cursor name, call SQLGetCursorName.
6. Call SQLPrepare with the second statement handle to compile the update or delete statement. Call SQLBindParameter, as necessary, to bind host variables to parameter markers embedded in the statement that represent the changes to be made.
7. Repeatedly call SQLFetch with the select statement handle until a row that you want to modify is returned.
8. Assign the parameter variables to the desired values and call SQLExecute using the second statement handle to perform the modification. Repeat steps 7 and 8 until completed.
9. Call SQLEndTran to commit the changes.
10. Drop the statement handles.

This procedure is demonstrated in the following example. The RaiseComm function fetches and displays each row of the salesperson table allowing the user (for example, the sales manager) the option of raising the salesperson's commission rate by 1%. This edition of the function uses SQLSetCursorName to give the select statement the cursor name, "comm_raise". The SQLSetCursorName can only be called prior to the call to SQLExecute. It can even be called before the SQLPrepare. If the statement associated with the cursor is not an updateable select statement, SQLSetCursorName will return an error.

```
#include "rdm.h"


static char SaleSel[ ] =
       "select sale_id, sale_name, commission from salesperson";
static char SaleUpd[ ] =
       "update salesperson set commission=commission+0.01 "
       "where current of comm_raise";
/* raise commission for selected salepersons */
       SQLRETURN RaiseComm(HDBC hdbc)
{ char sale_id[4], sale_name[31];
       float comm;
       long csize;
       char sqlstate[6], errmsg[80];
       HSTMT sHdl, uHdl;
              SQLRETURN stat;
       /* step 1: allocate select statement handle */
       stat = SQLAllocHandle(SQL_HANDLE_STMT, hdbc, &sHdl);
       if ( stat != SQL_SUCCESS ) {
                     /* this will catch connection handle problems */
                     return( stat );
       }
       /* step 2: allocate update statement handle */
       SQLAllocHandle(SQL_HANDLE_STMT, hdbc, &uHdl);
       /* step 3: compile the select statement */
       SQLPrepare(sHdl, SaleSel, SQL_NTS);
       /* step 4: specify cursor name */
       SQLSetCursorName( sHdl, "comm_raise", SQL_NTS);
```

```
        /* step 5: bind select stmt columns and execute select statement */
        SQLBindCol(sHdl, 1, SQL_C_DEFAULT, sale_id, 4, NULL);
        SQLBindCol(sHdl, 2, SQL_C_DEFAULT, sale_name, 31, NULL);
        SQLBindCol(sHdl, 3, SQL_C_DEFAULT, &comm, sizeof(float), &csize);
        stat = SQLExecute(sHdl);
        if ( stat == SQL_SUCCESS ) {
         /* step 6: compile positioned update statement */
                        SQLPrepare(uHdl, SaleUpd, SQL_NTS);


        /* step 7: fetch/display each row, allow user to update if desired */
        while ( (stat = SQLFetch(sHdl)) == SQL_SUCCESS ) {
                if ( DisplaySalesperson(sale_id, sale_name, comm, csize) == UPDATED )
        {
         /* step 8: this salesperson gets the raise */
                if ( (stat = SQLExecute(uHdl)) != SQL_SUCCESS )
                   break;
                                }
                }
        }
        if ( stat == SQL_ERROR ) {
           SQLDiagRec(SQL_HANDLE_DBC, hdbc, 1, NULL, sqlstate, NULL, emsg, 80, NULL);
           printf( "***ERROR(%s): %s\n", sqlstate, errmsg);
           SQLEndTrans(SQL_HANDLE_DBC, hdbc, SQL_ROLLBACK);
        }
        else {
        /* step 9: commit the changes */  SQLEndTrans(SQL_HANDLE_DBC, hdbc,
        SQL_COMMIT);
                }

    /* step 10: drop the statement handles */
    SQLFreeHandle(SQL_HANDLE_STMT, sHdl);
    SQLFreeHandle(SQL_HANDLE_STMT, uHdl);
    return stat;
}
```

The next example is the edition of the RaiseComm function that uses a system-generated cursor name retrieved through a call to SQLGetCursorName.

```
#include <string.h>
#include "rdm.h"


static char SaleSel[ ] =
       "select sale_id, sale_name, commission from salesperson";
static char SaleUpd[ ] =
```

```
        "update salesperson set commission=commission+0.01 "
        "where current of SQL_CUR_DDD_DDD";
/* raise commission for selected salepersons */
 SQLRETURN RaiseComm(HDBC hdbc)
{  char sale_id[4], sale_name[31];
        float comm;
        long csize;
        char *cp, cursor[16], sqlstate[6], errmsg[80];
        HSTMT sHdl, uHdl;
                SQLRETURN stat;

        /* step 1: allocate select statement handle */
        stat = SQLAllocHandle(SQL_HANDLE_STMT, hdbc, &sHdl);
        if ( stat != SQL_SUCCESS ) {
                        /* this will catch connection handle problems */
                        return( stat );
        }

        /* step 2: allocate update statement handle */
        SQLAllocHandle(SQL_HANDLE_STMT, hdbc, &uHdl);

        /* step 3: compile the select statement */
        SQLPrepare(sHdl, SaleSel, SQL_NTS);

        /* step 4: bind select stmt columns */
        SQLBindCol(sHdl, 1, SQL_C_DEFAULT, sale_id, 4, NULL);
        SQLBindCol(sHdl, 2, SQL_C_DEFAULT, sale_name, 31, NULL);
        SQLBindCol(sHdl, 3, SQL_C_DEFAULT, comm, sizeof(float),
                &csize);
        stat = SQLExecute(sHdl) ;
        if ( stat == SQL_SUCCESS ) {
                /* copy cursor name into positioned update statement */
                SQLGetCursorName(sHdl, cursor, 16, NULL);
                cp = strtok(SaleUpd, "SQL_CUR");
                strcpy(cp, cname);


        /* step 5: compile positioned update statement */
        SQLPrepare(uHdl, SaleUpd, SQL_NTS);

        /* step 6: fetch/display each row, allow user to update
                if desired */
       while ( (stat = SQLFetch(sHdl)) == SQL_SUCCESS ) {
                if (DisplaySalesperson(sale_id, sale_name, comm, csize) == UPDATED )
{

        /* step 7: this salesperson gets the raise */
                        if ( (stat = SQLExecute(uHdl)) != SQL_SUCCESS )
```

```
                                        break;
                                    }
                                }
            }
            if ( stat == SQL_ERROR ) {
               SQLDiagRec(SQL_HANDLE_DBC, hdbc, 1, NULL, sqlstate, NULL, emsg, 80, NULL);
               printf( "***ERROR(%s): %s\n", sqlstate, errmsg);
               SQLEndTrans(SQL_HANDLE_DBC, hdbc, SQL_ROLLBACK);

    }    else {
            /* step 8: commit the changes */
            SQLEndTrans(SQL_HANDLE_DBC, hdbc, SQL_COMMIT);
            }

            /* step 9: drop the statement handles */
            SQLFreeHandle(SQL_HANDLE_STMT, sHdl);
            SQLFreeHandle(SQL_HANDLE_STMT, uHdl);
            return stat;
  }
```

2. Using SQL DDL Statements

2.1 Introduction

This chapter describes how to use the DDL portion of the RDM Embedded SQL database language and presents other DDL-related information. The chapter provides the following:

- Definition of the RDM Embedded SQL DDL specification
- Overview of RDM Embedded SQL DDL
- Discussion of the sddlp processor
- Usage of RDM Embedded SQL DDL statements
- A complete DDL example

2.2 Defining an RDM Embedded SQL DDL specification

The Data Definition Language (DDL) for RDM Embedded SQL is based on, but not strictly compliant with standard ANSI SQL. (Refer to the section Using RDM Embedded SQL DDL statements later in this chapter for a definition of the language.)

An RDM Embedded SQL DDL (database definition language) specification is contained in a text file and is processed by the SQL DDL processor, sddlp, to create the database dictionary file, dbname.dbd.

2.3 Using RDM Embedded SQL DDL statements

A summary of the RDM Embedded SQL DDL statements appears in the following table.

SQL DDL Statement	Description
create database	Create new database definition.
create constant	Create a name for an integer constant. Only used for specifying char column lengths.
create domain	Create a name for a new data type and attributes (length and nullability).
create file	Create a data or index file.
create table	Create a table definition including its columns and indexes.

The required order of statements in an RDM Embedded SQL DDL specification file is shown by the following syntax.

```
create_database [create_constant | create_domain]…
       [create_file]… create_table …
```

The specification begins with a single create database statement followed by zero or more create constant and create domain statements followed by zero or more create file statements followed by one or more create table statements. Both C++ and C-style comments can be included in the specification. Thus the two styles listed below are recognized by sddlp:

```
/* This is a comment which can span multiple lines
*/

// This comment ends at the end-of-line
```

2.3.1 create database statement

The database schema for your SQL application starts with a create database statement specifying the name of the database you are creating. The syntax for this statement is given below.

```
create database dbname [pagesize = number] ;
```

This statement initiates the creation of a database named dbname. The pagesize clause is optional and, if specified, indicates the default page size for all files contained in the database.

The name of the database dbname is an identifier and is used by sddlp to form the names of the database dictionary file and the C or C++ header file. The dictionary file is named dbname.dbd and the header file is named dbname.h. Note that the length of this name can be up to 31 characters.

The RDM Embedded SQL data and key files are blocked or divided into fixed-length pages, each containing as many record or key occurrences as will fit on a page. The pagesize parameter determines the default database page size in bytes. This value should, for performance reasons, always be a multiple of the basic block size for your operating system (a multiple of 512 will work for most systems). If not specified, the default database page size is at least 1024 bytes, but will be the first number divisible by 512 that is large enough to contain the largest record.

The following example shows the use of create database in its simplest form, to create the example sales database.

```
create database sales;
```

The create database statement must appear in the DDL specification file before any of the create file, create table, and create set statements defining the tables that will be included in the database.

2.3.2 create constant and domain statements

The RDM Embedded SQL DDL allows you to declare user-defined constants and data types using the create constant and create domain statements. If you choose to use these statements, then you must include any constant and domain declarations after the create database but before any create file (or create table if create file is not used) statements.

The syntax for create constant is as follows.

```
create constant name = number;
```

The name is an identifier that can be used to specify the length of a character type or column. The number must be an integer value. The example below shows how constants can be used.

```
create database sales;
create constant NAMELEN = 30;
create table customer(
               cust_id char(3) primary key,
               company char(NAMELEN),
               contact char(NAMELEN),
               street char(30),
               city char(17),
               state char(2),
               zip char(5),
               phone char(12)
);
```

The create domain statement is used to declare a user-defined data type that can be used with any column declaration in create table. The syntax is given below.

```
create domain name as datatype [ ( length ) ] [not null] ;
```

The datatype is one of the standard base SQL data types as shown in the following table.

RDM Embedded SQL data types

SQL Data Type	Corresponding C Type
char	char
varchar	char
wchar	wchar_t
varwchar	wchar_t
smallint	int16_t
integer	int32_t

bigint	int64_t
real	float
float	double
date	int32_t
time	int32_t
timestamp	int32_t

The domain name can be used in any column declaration in place of the usual data type specification. The following example illustrates the use of domains.

```
create database sales;
create domain NAME as char(30) not null;
create table customer(
               cust_id char(3) primary key,
               company NAME,
               contact NAME,
               street char(30),
               city char(17),
               state char(2),
               zip char(5),
               phone char(12)
);
```

The previous example is identical to the following DDL without the use of create domain.

```
create table customer(
               cust_id char(3) primary key,
               company char(30) not null,
               contact char(30) not null,
               street char(30),
               city char(17),
               state char(2),
               zip char(5),
               phone char(12)
);
```

Constants can be used in domain declarations as in the example shown below, which combines the two previous examples.

```
create database sales;
create constant NAMELEN = 30;
create domain NAME as char(NAMELEN) not null;
create table customer(
               cust_id char(3) primary key,
```

```
                company NAME,
                contact NAME,
                street char(30),
                city char(17),
                state char(2),
                zip char(5),
                phone char(12)
);
```

2.3.3 create file statement

Three statements can be used to specify the files that will contain the table records, index keys and variable length data. The create data file statement specifies the name of a file that will contain the record occurrences for each of the tables it contains. The create index file specifies the name of a file that will contain the key occurrences for each of the indexes or key fields contained within it. The create vardata file specifies the name of a file that will contain the variable length text data associated with the fields contained within it.

The use of create file statements is not required by RDM Embedded SQL. If you choose not to use them, then the data associated with each table and index/key will be stored in separate files that will be automatically generated by the SQL DDL processor. Therefore, if you do choose to use create file, you must declare files for all of your tables and indexes/keys.

The create file must appear in the DDL specification file before any create table statements that are contained in the file.

The syntax for the create file statements are as follows.

```
create data file
"filename" [pagesize = number]
       containing TableName[,
       TableName]… ;

create index file
"filename" [pagesize = number]
       containing {TableName(KeyName) | KeyName}[,
       TableName(KeyName) |
KeyName]… ;

create index file
"filename" [pagesize = number]
       [inmemory [volatile | read | persistent]
       [initial = pages] [next = pages] [maxpgs = pages]
       containing {TableName(KeyName) | KeyName}[,
               TableName(KeyName) |
```

```
KeyName]… ;

create vardata file "filename" [pagesize = number]
       [inmemory [volatile | read | persistent]
       [initial = pages] [next = pages] [maxpgs = pages]
       containing {TableName(FldName) | FldName}[,
               TableName(FldName) | FldName]… ;
```

File declarations identify the physical files to contain the data stored in the database. The data file statement defines a file that will contain the rows of one or more tables. The index file statement defines a file that will contain the key occurrences for one or more indexes or primary/unique key columns.

The data, key and vardata files are blocked or divided into fixed-length pages, each containing as many record, key or text block occurrences as will fit on a page. The optional pagesize parameter specifies the page size for the file. If not specified, the page size for the file will be the default database page size. This value should always be a multiple of the basic block size for your operating system (a multiple of 512 will work for most systems). Otherwise, the operating system's file access performance will be impaired.

The name of the file, filename, is a string enclosed in quotation marks containing the physical (operating system) name of the file. It may be a fully qualified path name but must not exceed 47 characters in length. If the name of the file is not qualified (that is, it does not include a directory name), applications using the database must be executed from within the directory containing the database files.

All tables defined in the DDL must be contained in a data file. All rows of the tables listed in the containing clause will be stored in that file. The rows of a given table can only be stored in a single file. Each page in the data file consists of one or more fixed-length record slots. The size of the record slot is based upon the size of the largest-sized table contained in the file. Rows from smaller-sized tables will occupy the same record slots, thus leaving unused space. You can define all tables to be contained in one file or you can have a separate data file for each individual table. The choice is yours.

All indexes and primary/unique key columns defined in the DDL must be contained in an index file. All occurrences of the indexes and primary/unique key columns listed in the containing clause will be stored in the file. The "TableName(KeyName)" format must be used if there are two or more indexes/key columns with the same name. The TableName is the name of the table on which the index is created or the key column is declared.

Occurrences of a given index type can only be stored in a single file. Each page in the index file consists of one or more fixed-length key slots. The size of the key slot is based upon the size of the largest index or primary/unique key column contained in the file. Smaller keys will occupy the same slots, thus leaving unused space. As with tables, you can define all indexes and primary/unique key columns to be contained in one file or you can have a separate index file for each.

All varchar and varwchar columns defined in the DDL must be contained in a vardata file. The text data associated with the columns listed in the containing clause will be stored in the vardata file, while pointers to the vardata slots, and possibly a part of the text itself, will be stored in the data file that contains the associated table.

Examples:

```
create data file datfile = "chkg.dat" containing budget, check;
create index file keyfile1 = "chkg.k01" pagesize 1024
               containing code;
create data file keyfile2 = "chkg.k02" pagesize 2048
               containing check_no;
```

The checking account database consists of one data file and two key files. The data file identified as datfile contains the rows of both the budget and check tables in the physical file named chkg.dat. The default page size for datfile is 1024 bytes. The index file identified as keyfile1 has a page size of 1024 bytes and contains the index for the primary key column code in file chkg.k01. Index file keyfile2 has a page size of 2048 bytes and contains the index for the primary key column check_no in file chkg.k02.

```
create data file "/client/master.dat" containing master
```

The above file name specified in this data file statement includes the fully qualified path for file master.dat located in the client directory.

```
create index file "invnt.k01" containing stock(id_code);
create index file "invnt.k02" containing bkorder(id_code)
```

The above example shows duplicate key names that are qualified by the name of the table in which they are declared.

2.3.4 create table statement

The create table statement is used to declare the columns that will be stored in each individual row of the table. You can also define one or more indexes or keys that can be used to access individual rows or groups of related rows from the table. The syntax for the create table statement is shown below.

```
create [circular] table
TableName(  ColumnName datatype[(length)] [not null] [[primary | unique] key]
[,   ColumnName datatype[(length)] [not null] [[primary | unique] key] ]… )
[create [unique] index
IndexName(ColumnName [asc | desc] [,ColumnName [asc | desc]]…)]… ;
```

The name of the table is TableName, which is formed as an identifier. In circular tables rows are automatically re-used when the table becomes full. A circular table must be the only table in its file. The definition of this file must include a maxslots value specifying the maximum number of slots or rows in the file, and therefore the size

of the circular table. Once all the rows in the table are full, RDM Embedded will overwrite the oldest row whenever a new one is created.

The datatype of each column must be either, 1) one of the base types listed earlier in the RDM Embedded SQL data types table, or 2) the name of a user-declared domain. If the datatype is char or wchar then a length can be specified giving the maximum number of characters that will be stored in the column. Arrays of non-character types are not supported in SQL and also are not supported in RDM Embedded SQL.

The not null clause can be specified to indicate that values must always be specified for the column in all insert statements that store new rows into the table. An update statement that attempts to set a not null column to null will return an error.

The "[primary | unique] key" clause identifies a single column that can be used to locate a single row (with the primary or unique attribute) or all rows from the table with a given column value. Note that primary and unique are interchangeable. Typically a table only has one primary key but can have more than one unique key. The sddlp utility creates a separate index for each declared key. The primary/unique attribute requires that the values of the column be unique for each row of the table.

The create index is used to create a multi-column index on the table. The unique attribute indicates that the combination of column values be unique for each row of the table.

Note carefully the syntactical structure, particularly with regard to the create index component and the placement of the ending semicolon. Technically speaking, the create index is an optional clause of the create table statement rather than a separate DDL statement.

The indexes/keys are stored by RDM Embedded SQL using a standard B-tree technique. Some create table examples are shown below.

```
create table customer(
               cust_id char(3) not null primary key,
               company char(30) not null,
               contact char(30) not null,
               street char(30),
               city char(17),
               state char(2),
               zip char(5),
               phone char(12) not null,
               email char(64) key
)
```

The above statement creates a table named "customer" that contains information about a company's customers. The cust_id column is declared at the primary key so that individual rows are uniquely identified and accessed by the 3-character customer id code. Along with the cust_id column, the company, contact, and phone columns must always contain a value because of the not null attribute. The email column is declared as a key so that customer rows can be accessed by email address. Even though customers will have unique email addresses, the key is not given the unique attribute because some customers may not have an email address, hence the absence of the not null attribute.

```
create table sales_order(
               amount float,
               tax real,
               ord_date date,
               ord_num smallint primary key
)
create index order_key(ord_date desc, ord_num);
```

The statement above gives the declaration of a table named sales_order. The amount column is declared as float (which maps to a C double) and the tax column as real (which maps to a C float). The ord_date contains the date of the order. Dates are stored as long integers containing the number of elapsed days since January 1, 1 AD. (Values of type time are stored as long integers containing the time of day in milliseconds. Timestamps are stored as a two element long integer array in which the first element is the date and the second is the time.) The sales_order table has two indexes. The first is the index on the ord_num primary key (key columns are automatically indexed). The second is the index named order_key whose multiple columns are stored in descending order by ord_date, and within a given ord_date, are in increasing order by ord_num.

The next example illustrates the use of multiple create index clauses to specify indexes on single columns. You do not have to use the key attribute; you can use create index as an alternative.

```
create table salesperson(
               commission real,
               region smallint,
               office char(5),
               sale_id char(3),
               sale_name char(30)
)
create unique index sid(sale_id)
create index sname(sale_name);
```

2.3.5 Complete DDL example

The DDL example given below is for a hypothetical sales tracking database. The example first gives the RDM Embedded SQL DDL specification followed by the sales.h header file.

```
/* Example RDM Embedded SQL Sales Database */

create database sales;


create constant IDLEN = 3;
create domain NAME as char(30) not null;
```

```
/* If the following 3 create file statements were not specified
               then each table and index/key would be stored in 7 separate
               files created automatically by sddlp and named "sales.000",
               "sales.001", ... "sales.006". */
create data file "sales.d00" containing customer, sales_order,
               salesperson;
create index file "sales.k00" containing cust_id, sale_id,
               ord_num;
create index file "sales.k01" containing order_key;
create table customer(
               cust_id char(IDLEN) not null primary key,
               company NAME,
               contact NAME,
               street char(30),
               city char(17),
               state char(2),
               zip char(5),
               phone char(12)
);
create table salesperson(
               sale_id char(IDLEN) not null primary key,
               sale_name NAME,
               dob date,
               commission real,
               region smallint,
               office char(15),
               mgr_id char(IDLEN)
);
create table sales_order(
               ord_num smallint primary key,
               ord_date date,
               amount float,
               tax real
)
create index order_key(ord_date desc, ord_num asc);
```

3. Using SQL DML Statements

3.1 Introduction

This chapter describes how to use the DML (database manipulation language) portion of the RDM Embedded SQL database language and presents other DML-related information.

The chapter provides the following:

- Overview of RDM Embedded SQL DML
- How to perform and sort queries using select statements
- How to enter and modify data using transactions and the insert, update and delete statements
- Usage of the lsql utility

3.1.1 Overview of the RDM Embedded SQL DML

The SQL database manipulation language (DML) features provided in RDM Embedded SQL are a subset of the 1992 ANSI SQL standard. Because of the limited resources available on certain target platforms, we have minimized the SQL capabilities in RDM Embedded SQL to those that are most necessary for efficient data access of local database information.

Note that the information here is presented and explained in a hierarchical manner, that is, going from simple queries and building to more complex use of the DML statements. A more dictionary-like format of all the statements can be found in chapter 6, the SQL DML Statement Reference.

3.1.1.1 SQL Enhancement to Fetch Database Addresses

The RDM Embedded SQL library has been enhanced with the ability to retrieve the database address associated with any record in the result set. To support this feature, the SQL language has been extended to recognize two virtual columns in each table, called tablename.db_addr and tablename.rowid, which evaluate to the database address and the rowid of the current row in the specified table. The rowid is derived from the database address by masking out the file number, to leave the slot number only. The ability to retrieve database addresses through the SQL API makes it practical for the first time to use the SQL and d_ APIs in a single application.

As well as the change to the SQL language described above, a minor extension to the ODBC API has been made. A new SQL data type SQL_DB_ADDR has been defined, for columns consisting of the new db_addr keyword in a SQL select statement. This SQL data type will be returned by SQLDescribeCol for any db_addr column. Similarly, a new C data type SQL_C_DB_ADDR has been defined, for specifying C variables bound to database address columns. This C data type must be passed to SQLBindCol for database address columns, along with the address of a variable of type DB_ADDR.

These data types will only be used for columns consisting of the new db_addr keyword in select statements. Database fields of type db_addr, as defined in the database schema, are not currently supported by the SQL library, and will still not be supported.

Columns that contain the new rowid keyword in select statements will have SQL data type SQL_INTEGER, and C data type SQL_C_LONG.

3.1.2 Performing SQL queries with select

3.1.2.1 Basic queries

The select statement is used to retrieve rows from a table. The syntax for a basic select statement is as follows.

```
select {ColumnName [,ColumnName]…} | *} from TableName ;
```

The ColumnName identifies a column declared in TableName. You can specify "*" instead of the column list to indicate that all columns declared in TableName are to be returned. For example, you can enter the following statement to see all the data in the salesperson table.

```
select * from salesperson;


SALE_ID SALE_NAME       DOB        COMMISSION REGION OFFICE MGR_ID

BNF     Flores, Bob     1943-07-17     0.1        0      SEA    *null*
GAP     Porter, Greg    1949-03-03     0.08       1      SEA    *null*
BPS     Stouffer, Bill  1952-11-21     0.08       2      SEA    *null*
CMB     Blades, Chris   1958-09-08     0.08       3      SEA    *null*
SWR     Roberts, Sue    1968-10-11     0.07       0      LAX    BNF
BCK     Kennedy, Bob    1956-10-29     0.075      0      DEN    BNF
ERW     Wyman, Eliska   1959-05-18     0.075      1      NYC    GAP
SKM     McGuire, Sidney 1947-12-02     0.07       1      WDC    GAP
WAJ     Jones, Walter   1960-06-15     0.07       2      CHI    BPS
WWW     Warren, Wayne   1953-04-29     0.075      2      MIN    BPS
GSN     Nash, Gail      1954-10-20     0.07       3      DAL    CMB
SSW     Williams, Steve 1944-08-30     0.075      3      ATL    CMB
JTK     Kirk, James     2100-08-30     0.075      3      ATL    *null*
DLL     Lister, Dave    1999-08-30     0.075      3      ATL    *null*
```

The result columns are listed in the order in which they are declared in the salesperson table. Each salesperson has a unique id code (the primary key). The mgr_id column is null for the regional managers. The salespersons who work for each regional manager have their manager's sale_id code in the mgr_id column. Our fictitious computer supply company has four sales regions. Region 0 is west, region 1 is east, region 2 is central, and region 3

is south. The company headquarters is located in Seattle with sales offices located in Los Angeles (LAX), Denver (DEN), New York City (NYC), Washington D.C. (WDC), Chicago (CHI), Minneapolis (MIN), Dallas (DAL), and Atlanta (ATL). The office each salesperson works from is shown in the table. Also shown is each salesperson's birth date and commission rate.

If you are interested in seeing only certain columns, you can specify only those columns in the select statement as shown below.

```
select sale_name, commission from salesperson;


SALE_NAME           COMMISSION

Flores, Bob         0.1
Porter, Greg        0.08
Stouffer, Bill      0.08
Blades, Chris       0.08
Robinson, Stephanie 0.07
Kennedy, Bob        0.075
Wyman, Eliska       0.075
McGuire, Sidney     0.07
Jones, Walter       0.07
Warren, Wayne       0.075
Nash, Gail          0.07
Williams, Steve     0.075
Kirk, James         0.075
Lister, Dave        0.075
```

You can select any of the columns from the specified table in any order. See below.

```
select mgr_id, dob, sale_name from salesperson;
select commission, sale_id, office, sale_name,
       region from salesperson;
```

The output table produced from the selected columns is called a projection in relational database terminology.

3.1.3 Sorting queries

You can choose to sort the output table of a select using the order by clause as specified in the following syntax.

```
select {ColumnName [, ColumnName]…} | *} from TableName
 order by {number | ColumnName} {asc | desc}
[,{number | ColumnName} {asc | desc}]… ;
```

The resultant rows of the select statement are sorted using the order by clause to specify the columns on which the sort is to be performed, and to specify if those columns are sorted in ascending (default) or descending order. The column references are specified by either the ColumnName or by its ordinal position (number) in the select list (that is, the first column is 1, the second 2, etc.). The first column listed will be the major sort column, the values in the second column will be sorted within each of the major columns values, and so on.

For example, the following statement sorts the salesperson table in alphabetical sale_name order.

```
select * from salesperson order by sale_name;

SALE_ID SALE_NAME      DOB        COMMISSION REGION OFFICE MGR_ID

CMB     Blades, Chris  1958-09-08      0.08       3      SEA    *null*
BNF     Flores, Bob    1943-07-17      0.1        0      SEA    *null*
WAJ     Jones, Walter  1960-06-15      0.07       2      CHI    BPS
BCK     Kennedy, Bob   1956-10-29      0.075      0      DEN    BNF
JTK     Kirk, James    2100-08-30      0.075      3      ATL    *null*
DLL     Lister, Dave   1999-08-30      0.075      3      ATL    *null*
SKM     McGuire, Sid   1947-12-02      0.07       1      WDC    GAP
GSN     Nash, Gail     1954-10-20      0.07       3      DAL    CMB
```

Columns listed in the order by clause can be specified by name or by number. The first column in the select list is 1, the second column is 2, and so on. The following lists the salesperson names and birth dates in date of birth (DOB) order.

```
select sale_name, dob from salesperson order by 2;


SALE_NAME           DOB

Flores, Bob         1943-07-17
Williams, Steve     1944-08-30
McGuire, Sidney     1947-12-02
Porter, Greg        1949-03-03
Stouffer, Bill      1952-11-21
Warren, Wayne       1953-04-29
Nash, Gail          1954-10-20
Kennedy, Bob        1956-10-29
Blades, Chris       1958-09-08
Wyman, Eliska       1959-05-18
```

```
Jones, Walter       1960-06-15
Robinson, Stephanie 1968-10-11
Lister, Dave        1999-08-30
Kirk, James         2100-08-30
```

You can also sort on more than one column as well as being able to specify whether each column is in ascending (the default) or descending order, as in the next example.

```
select commission, sale_name from salesperson order by 1 desc, 2 asc;

COMMISSION SALE_NAME

0.1        Flores, Bob
0.08       Blades, Chris
0.08       Porter, Greg
0.08       Stouffer, Bill
0.075      Kennedy, Bob
0.075      Kirk, James
0.075      Lister, Dave
0.075      Warren, Wayne
0.075      Williams, Steve
0.075      Wyman, Eliska
0.07       Jones, Walter
0.07       McGuire, Sidney
0.07       Nash, Gail
0.07       Robinson, Stephanie
```

Column 1 is the primary sort column. The values in column 2 that have identical column 1 values are sorted within that group.

3.1.3.1 Conditional row selection

The previous examples selected all of the rows in the salesperson table. However, you often want to select data that meets a specific set of selection criteria. You can use the where clause of the select statement to specify which particular rows to select as specified in the syntax below.

```
select {ColumnName [, ColumnName]…} | *} from TableName where cond_expr ;
cond_expr:
 rel_expr [{and | or} rel_expr]…
rel_expr:
 ColumnName rel_oper value
       | ColumnName [not] like "pattern"
       | ColumnName [not] between value and value
```

```
        | ColumnName [not] in (value[, value]…)
        | ColumnName is [not] null
        | (cond_expr)
        | not rel_expr
```

The where clause contains a conditional expression consisting of one or more relational expressions separated by the Boolean operators and and or. Relational expressions are comparisons of two expressions that evaluate to true or false. A relational expression can also be a parenthesized conditional expression. The relational operators in RDM Embedded SQL are shown in the following table.

Relational Operator	Definition
=	Equal
<>	Not equal
<	Less than<
<=	Less than or equal
>	Greater than
>=	Greater than or equal
like "pattern"	String match with pattern
between lowVal andhighVal	>= lowVal and <= highVal
in (val1, val2, ...)	Equal to one of the specified values

The usual relational comparison operators (for example, "=" , "<" , ">") compare the contents of a column with a literal value.

For example, the following query will select only those customers who have a sale_id equal to "SKM" (that is, the customer accounts that are serviced by Sidney McGuire).

```
select sale_id, cust_id, company, city, state from customer
 where sale_id = "SKM";


SALE_ID   CUST_ID   COMPANY                    CITY          STATE
SKM       PIT       Steelers National Bank     Pittsburgh    PA
SKM       WAS       Redskins Outdoor Equip Co. Arlington     VA
SKM       PHI       Eagles Electronics Corp.   Philadelphia  PA
```

The between operation checks that the specified expression returns values inclusively within a given range of values. The following query returns the January sales orders.

```
select cust_id, ord_num, ord_date from sales_order
    where ord_date between date '1997-01-01' and date '1997-01-31';
```

```
CUST_ID ORD_NUM ORD_DATE
CHI     2201    1997-01-02
MIN     2202    1997-01-02
KCC     2203    1997-01-02
CIN     2204    1997-01-02
BUF     2205    1997-01-03
LAN     2206    1997-01-02
DEN     2207    1997-01-06
PHI     2208    1997-01-07
PHO     2209    1997-01-07
IND     2210    1997-01-09
GBP     2211    1997-01-10
ATL     2212    1997-01-15
NYG     2213    1997-01-16
LAA     2214    1997-01-16
SEA     2215    1997-01-17
KCC     2216    1997-01-21
SDC     2217    1997-01-24
NOS     2218    1997-01-24
DET     2219    1997-01-27
DEN     2220    1997-01-27
NEP     2221    1997-01-27
CLE     2222    1997-01-28
MIN     2223    1997-01-28
TBB     2224    1997-01-28
SEA     2225    1997-01-29
HOU     2226    1997-01-30
IND     2227    1997-01-31
```

Notice that date constants are encoded according to the following format:

```
date 'YYYY-MM-DD'
```

The date constant format conforms to the 1992 ANSI SQL and ODBC standards. In this format all four digits of the year must be specified. Date '97-01-02' would be January 2, 97 A.D not January 2, 1997.

The in operation evaluates to true when the value of the expression is contained in a specified list of values. All customers located in Pacific coast states are selected with the following statement.

```
select cust_id, company, city, state from customer
    where state in ("CA", "OR", "WA");


CUST_ID COMPANY                 CITY          STATE
```

```
SEA     Seahawks Data Srvcs     Seattle       WA
SFF     Forty-niners Group      San Francisco CA
LAA     Raiders Dev. Co.        Los Angeles   CA
LAN     Rams Data Process, Inc. Los Angeles   CA

SDC     Chargers Credit Corp.   San Diego     CA
```

The like operation allows wild-card checking of string expressions: a _ character in the check string matches any single character, and a % in the check string matches zero or more characters. The comparison expression must return a string result. The next query returns the customers that have "Data" as part of their company name.

```
select cust_id, company, city, state from customer
    where company like "%Data%";

CUST_ID COMPANY                 CITY         STATE

SEA     Seahawks Data Srvcs     Seattle      WA
LAN     Rams Data Process, Inc. Los Angeles  CA
DAL     Cowboys Data Services   Dallas       TX
TBB     Bucks Data Services     Tampa        FL
```

3.1.3.2 Performing multiple table joins

All of the example queries so far have selected data from a single table. It is often necessary to retrieve data from several related tables. The tables from which the data is to be retrieved are specified in a comma-separated list in the from clause. Each table must also have a column that matches a column of one of the other listed tables and this must be specified in the where clause as shown in the following syntax.

```
select {ColumnName [, ColumnName]…} | *} from TableName [, TableName]…
 where [TableName.]ColumnName = [TableName.]ColumnName
 [and [TableName.]ColumnName = [TableName.]ColumnName]…
```

Each of the relational expressions that equate the matching columns in a pair of tables is called the join predicate. A properly formed multiple-table, join select statement will have one less join predicate than the number of tables listed in the from clause.

To select data from two tables, specify each table name in the from clause and include in the where clause a comparison that equates the associated columns from the two tables. This is shown in the following example that lists each salesperson's customers.

```
select sale_name, company, city, state from salesperson, customer
       where salesperson.sale_id = customer.sale_id;

SALE_NAME COMPANY CITY STATE . . .
```

Notice that in order to differentiate between the two sale_id columns in the salesperson and customer tables it was necessary to prefix the table names to the column names in the comparison (for example, salesperson.sale_id). The join predicate used to combine the two tables is: salesperson.sale_id = customer.sale_id.

Any number of tables can be joined with a select statement. The next example illustrates a three-table join that shows the January sales orders booked by Sue Roberts.

```
select sale_name, cust_id, ord_date, ord_num, amount
    from salesperson, customer, sales_order
    where salesperson.sale_id = "SWR" and
               salesperson.sale_id = customer.sale_id and
               customer.cust_id = sales_order.cust_id and
               ord_date between date   '1997-01-01' and date '1997-01-31';

SALE_NAME    CUST_ID ORD_DATE   ORD_NUM AMOUNT
Roberts, Sue LAN     1997-01-02 2206    15753.19
Roberts, Sue LAA     1997-01-16 2214    12614.34
Roberts, Sue SDC     1997-01-24 2217    705.98
```

3.1.3.3 Computational queries

Basic expression evaluation

In all of the previous examples only simple columns were specified in the select list. You can also specify arithmetic expressions involving columns and values as defined in the syntax below.

```
select expression [, expression]… from …expression:
 arith_operand [ { + | - | * | / } arith_operand]…
arith_operand:
 [TableName.]ColumnName
 | [+ | -]value
 | ( expression )
```

RDM Embedded SQL allows you to add, subtract, multiply, and divide columns and constants. Arithmetic expressions can be parenthesized in the usual manner. The following table shows the evaluation precedence of the arithmetic expression operators in order of highest to lowest. Operators at the same precedence level evaluate from left to right.

Arithmetic Operator	Definition (highest to lowest)

( )	parenthetical expressions
+, -	unary plus or minus (sign)
*, /	multiply and divide
+, -	Add and subtract

The sales order table contains a column called amount that contains the total amount of the order, and a column called tax containing the sales tax, if any. The following query will display the invoice amount that is the sum of the two columns.

```
select ord_date, amount, tax, amount+tax from sales_order;

ORD_DATE   AMOUNT    TAX      AMOUNT+TAX

1997-01-02 46740     4113.12  50853.1201171875
1997-01-02 25915.86  1840.03  27755.8900292969
1997-01-02 19567.08  978.35   20545.4299755859
1997-01-02 2733.28   0        2733.28
1997-01-03 150871.2  12069.7  162940.900195313
1997-01-02 15753.19  1417.79  17170.9800390625
1997-01-06 274375    17011.25 291386.25
1997-01-07 3437.5    0        3437.5
1997-01-07 3715.83   0        3715.83
1997-01-09 8780      0        8780
1997-01-10 53634.12  0        53634.12
1997-01-15 12569.75  817.03   13386.7800292969
1997-01-16 7895      0        7895
1997-01-16 12614.34  1135.29  13749.6300390625
1997-01-17 16892     1435.82  18327.8199462891
1997-01-21 7847.51   392.38   8239.89000488281
1997-01-24 705.98    63.54    769.520000915527
1997-01-24 81375     0        81375
1997-01-27 74034.9   0        74034.9
1997-01-27 49980     3098.76  53078.7600097656
1997-01-27 13235.34  992.65   14227.9900244141
1997-01-28 1877.03   0        1877.03
```

The next query shows the salespersons' orders with the largest earned commissions. The select statement computes the commission earned by multiplying the commission rate by the amount of the order. The salesperson's name is accessed by using a three-table join between the salesperson, customer, and sales_order tables. Finally, the result is sorted in descending order by the earned commission.

```
select sale_name, ord_num, amount*commission
               from salesperson, customer, sales_order
               where salesperson.sale_id = customer.sale_id
               and customer.cust_id = sales_order.cust_id
   order by 3 desc;
```

```
SALE_NAME           ORD_NUM  AMOUNT*COMMISSION
Kennedy, Bob        2207     20578.1258177012
Porter, Greg        2288     20193.9995486289
Wyman, Eliska       2205     11315.3404496312
Kennedy, Bob        2253     10753.1254272908
Robinson, Stephanie 2234     8726.20003715158
Kennedy, Bob        2237     7790.61030957103
Flores, Bob         2284     7431.51611073822
Nash, Gail          2324     7281.36503100023
Porter, Greg        2250     6594.46785260215
Porter, Greg        2219     5922.79186761528
Warren, Wayne       2292     5793.56273021549
Jones, Walter       2241     5762.05002453178
Nash, Gail          2218     5696.25002425164
Wyman, Eliska       2281     4975.61269771308
Williams, Steve     2230     4675.50018578768
Jones, Walter       2257     4363.80001857877
Wyman, Eliska       2270     4115.62516354024
Warren, Wayne       2211     4022.55915984213
Nash, Gail          2226     3841.25001635402
```

The previous query showed how much the salesperson gets from each order. The next query shows how much the company gets from each order. The amount to the company is simply the order amount minus the commission. The following query produces the desired results.

```
select sale_name, ord_num, amount-amount*commission
               from salesperson, customer, sales_order
               where salesperson.sale_id = customer.sale_id
                       and customer.cust_id = sales_order.cust_id
               order by 3 desc;



SALE_NAME     ORD_NUM AMOUNT-AMOUNT*COMMISSION

Kennedy, Bob  2207     253796.874182299
Porter, Greg  2288     232231.000451371
Wyman, Eliska 2205     139555.859550369
Kennedy, Bob  2253     132621.874572709
Roberts, Sue  2234     115933.799962848
Nash, Gail    2324     96738.1349689998
Kennedy, Bob  2237     96084.189690429
Jones, Walter 2241     76552.9499754682
```

```
Porter, Greg  2250     75836.3821473978
```

3.1.3.4 Grouped calculations

All select statements in this chapter have generated detail rows, that is, there is one line of output for each row selected. Although this is usually adequate for the information you are selecting, you often want to summarize a number of similar records into a single row. RDM Embedded SQL uses the select statement group by clause to summarize the selected information. The data is arranged based on the columns specified in the group by clause; one row is returned each time one of the listed columns changes value.

The set of rows of similar records on which the group by summary is performed is known as an aggregate. The functions described below perform their calculations on aggregates.

In addition to the arithmetic operators you saw earlier in this chapter, RDM Embedded SQL has calculation functions that perform computations on aggregates. The select statement syntax given below shows how to formulate grouped select statements.

```
select expression [, expression]… from TableName [, TableName]…
 [where cond_expr ]
 [group by [TableName.]ColumnName [,[TableName.]ColumnName]…
 [having cond_expr]
expression:
 arith_operand [ { + | - | * | / } arith_operand]…
arith_operand:
 [TableName.]ColumnName
 | value
 | calc_function
 | (expression)
calc_function:
 {sum | avg | max | min}(expression)
 | count ( {* | [TableName.]ColumnName} )
cond_expr:
 rel_expr [{and | or} rel_expr]…
rel_expr:
 expression {= | <> | < | > | <= | >=} expression
 | expression [not] like "pattern"
 | expression [not] between expression and expression
 | expression [not] in (value[, value]…)
 | (cond_expr)
 | not rel_expr
```

The following table lists the calculation functions.

Function	Description
sum (expr)	Computes the sum of the results of the specified expression for each row of the aggregate.
avg (expr)	Computes the average of all rows of the aggregate.
count ({colname | *})	Counts the number of rows in the aggregate.
max (expr)	Computes the maximum of the results of the specified expression for all rows of the aggregate.
min (expr)	Computes the minimum of the results of the specified expression for all rows of the aggregate.

An aggregate is specified with the group by clause of the select statement. Suppose you wanted to see the year-to-date earnings for each salesperson. The following example shows how grouped calculations are used to formulate a select statement that will produce the desired information. All orders for each salesperson are summarized and the total amount of all orders is computed as is the total commissions.

```
select sale_name, sum(amount), sum(amount*commission)
               from salesperson, customer, sales_order
               where salesperson.sale_id = customer.sale_id
                              and customer.cust_id = sales_order.cust_id
               group by sale_name;


SALE_NAME SUM(AMOUNT) SUM(AMOUNT*COMMISSION)
.
.
.
```

If you do not specify a group by clause, all of the resulting rows comprise the aggregate. The next query computes the total year-to-date sales.

```
select sum(amount) from sales_order;


SUM(AMOUNT)
3717187.03
```

The count function is used to calculate the number of detail rows from which the aggregate is comprised. The next query shows the number of orders placed by each customer.

```
select company, count(ord_num) from customer, sales_order
               where customer.cust_id = sales_order.cust_id
                              group by company;
```

```
COMPANY                         COUNT(ORD_NUM)
"Bills We Pay" Financial Corp. 5
Bears Market Trends, Inc.      5
Bengels Imports                5
Broncos Air Express            7
Browns Kennels                 7
.
.
.
```

The argument to count can be any of the column names or, since it really doesn't matter which column you choose to count (they all give the same result), you can simply write, count(*).

The following example shows the total number of orders in the system along with the minimum, maximum, and average order amounts.

```
select count(*), min(amount), max(amount), avg(amount)
from sales_order;


COUNT(*) MIN(AMOUNT) MAX(AMOUNT) AVG(AMOUNT)
127      68.75       274375      29269.1892125984
```

You cannot use a where clause to restrict the output rows based on calculated results. RDM Embedded SQL provides an additional clause, called having, for restricting rows based on aggregate functions. In the example below, the having clause selects only the results from those companies that have more than five orders for the year.

```
select company, count(ord_num), sum(amount)
               from customer, sales_order
               where customer.cust_id = sales_order.cust_id
               group by company having count(ord_num) > 5;

COMPANY COUNT(ORD_NUM) SUM(AMOUNT)
.
.
.
```

If you had put count(ord_num) > 5 in the where clause, the system would have returned an error. Calculation functions cannot be specified in the where clause. The where clause is used to restrict detail rows before they affect the summary calculations. The having clause is used to restrict aggregate result rows after the calculations have been performed.

The query in the next example shows the salespersons who had more than $50,000 in sales during the month of June.

```
select sale_name, sum(amount) from salesperson,
                              customer, sales_order
               where salesperson.sale_id = customer.sale_id and
                              customer.cust_id = sales_order.cust_id and
               ord_date
                              between date '1997-06-01' and date '1997-06-30'
               group by sale_name having sum(amount) > 50000.00;


SALE_NAME         SUM(AMOUNT)

Flores, Bob       55026.50
McGuire, Sidney   72541.92
Nash, Gail        104019.50
Roberts, Sue      58970.50
Williams, Steve   52569.49
Wyman, Eliska     103076.79
```

3.1.4 Entering and modifying data

3.1.4.1 Transactions

Transactions are used to maintain the logical consistency of a database by allowing multiple, related, database-modification statements to be grouped together and committed to the database as a single unit. Transactions guarantee that either all of the associated changes are stored in the database or that none of them are stored (for example, if the database RDM Embedded Server were to go down due to a power failure, uncommitted changes would not have been stored).

A transaction is started either explicitly by executing a begin transaction statement, or implicitly by executing an insert, update, or delete statement. The changes affected by all insert, update, and delete statements after a transaction begins are made permanent by executing a commit statement. Alternatively, you can undo (that is, discard) the changes by executing a rollback statement. However, once the changes have been committed, they cannot be rolled back.

You must use transactions in RDM Embedded SQL. If you do not commit your changes to the database before you end your session, all of the changes you have made will be lost.

Descriptions for each of the transaction processing statements are given below.

```
begin [work] ;
```

Begins a transaction. If not specified, the transaction begins when the first database modification statement is executed. Note that you cannot begin a transaction when there is already an active transaction.

```
commit [work];
```

Commits the changes made during the transaction to the database and ends the transaction.

```
rollback [work] ;
```

Rolls back (discards) all changes made during the transaction.

Example transactions are given in the following sections.

3.1.4.2 Inserting rows into a table

You insert rows into a table using the insert statement. The syntax for the insert statement is given below.

```
insert into TableName[(ColumnName[, ColumnName]...)] values(constant[, constant]...)
```

The values to be inserted are explicitly specified using the values clause.

If a ColumnName list is specified, then it must name columns defined in TableName. The values must be specified in the same order. If no columns are specified, the values must be listed in the same order as the columns defined in the create table for TableName.

An insert must contain a non-null value for each column that is defined to contain only non-null data.

For example, the following statement inserts a new salesperson into the salesperson table.

```
insert into salesperson(sale_id, sale_name, dob, commission,
                                region, office, mgr_id)
            values("MMB","Bryant, Mike",date
                                '1960-11-14',0.05,0,"SEA","BNF");
commit;
```

Each column of the salesperson table is specified in the column list (they can be specified in any order). The values for each column are specified in the values list and must be specified in the same order as their associated columns. You do not have to specify all of the table's columns. Only those columns that have been declared as not-null need to be specified. The unspecified columns will be assigned null.

If you do not include a column list, then you must specify the values in the order in which the columns have been declared in the create table, and values must be provided for all of the table's columns.

The next example shows the insert statements needed to store a complete sales order in the database.

```
begin work;
insert into sales_order
               values("SEA",2328,date '1993-06-30',time '13:17:00',
                               30036.50,2553.10, timestamp '1993-07-06 14:10:00'+);


insert into item values(2311,16311,"SEA",30);
insert into item values(2311,18061,"SEA",200);
insert into item values(2311,18121,"SEA",1000);
commit work;
```

The columns in the sales_order table are, respectively, cust_id, ord_num, ord_date, ord_time, amount, and tax. The columns in the item table are ord_num, prod_id, and quantity. Note that this is a single transaction, which contains four insert statements. Hence, there is a single commit statement.

Time constants are entered similarly to date constants. The format available for time values is as follows: time 'HH:MM:SS'

3.1.4.3 Updating columns in a table

The update statement is used to modify the values of one or more columns of one or more rows in a table.

```
update TableName
       set ColumnName = {expression | null} [, ColumnName = {expression | null}]...
       where cond_expr
```

The values to which the named columns in the set clause are assigned, are the evaluated results of the specified columns. The values of the columns in TableName that are referenced in the expressions are the pre-updated column values. The rows that are updated are those for which the conditional expression is true. If the where clause is not specified, all of the rows in TableName are updated.

The following example shows a basic update statement that sets the commission for salesperson with sale_id "SWR" (Stephanie Robinson) to 8 percent. This update modifies only a single row of a table.

```
begin transaction;
update salesperson
       set commission = 0.08
       where sale_id = "SWR";
commit transaction;
```

The next example gives each non-manager salesperson a 10% increase in their commission rate.

```
update salesperson
       set commission = commission + 0.10*commission
       where mgr_id is null;
commit work;
```

Rams Data Process, Inc. has moved to a new address. The next statement modifies the relevant columns in the customer table.

```
update customer
 set street = "17512 SW 123rd St.", city = "Tustin",
              zip = "90121"
       where cust_id = "LAN";
commit trans;
```

Eliska Wyman has left the company. Until her replacement is hired, her customers in New York and New Jersey are to be serviced by Greg Porter and her other customers by Sidney McGuire. The following statements make the appropriate changes.

```
begin trans;
update customer
       set sale_id = "GAP"
       where sale_id = "ERW" and state in ("NY", "NJ");
update customer
       set sale_id = "SKM"
       where sale_id = "ERW" and state not in ("NY", NJ");
commit trans;
```

3.1.4.4 Deleting rows from a table

After reassigning ERW's customers, it would be necessary to delete her from the salesperson table. The delete statement is used to delete rows from a table. The following statement shows how it is used to delete the sale id ERW.

```
delete from salesperson where sale_id = "ERW";
```

The syntax for the delete statement is shown below.

```
delete from TableName [where cond_expr]
```

The statement deletes all of the rows from TableName for which the conditional expression in the where clause is true. If the where clause is not specified, all of the rows in TableName are deleted.

3.2 Using the interactive RDM Embedded SQL utility

A utility program is provided that allows a user to interactively execute RDM Embedded SQL DDL statements. This program, called lsql, is designed primarily to aid in the testing of RDM Embedded SQL DDL and to help you learn RDM Embedded SQL.

The RDM Embedded SQL statements are submitted one at a time. The program keeps track of the most recently entered statements, allowing you to select a previously entered statement, edit it, and re-execute it as desired. The program can also process RDM Embedded SQL statements stored in text files.

3.2.1 Invoking the lsql utility

The RDM Embedded SQL utility lsql is a program for controlling the compilation and execution of RDM Embedded SQL DDL statements. Execute lsql as shown below.

```
lsql [-s num] [-l num] [-f script] dbname[;dbname]…
```

Each of the listed command line arguments is described below.

-s num

This option specifies the maximum number of statement handles per connection that can be used. The default is 5.

-l num

This option specifies the number of display lines per output page. The default is 25.

-f script

Process the lsql statements contained in the text file named script.

dbname

This is the name of a database to be opened and accessed. Each dbname.dbd file must exist in the current directory.

Once you have started lsql, you enter utility commands or RDM Embedded SQL statements at the prompt ("lsql>"). Each statement is terminated by a semi-colon (";") at the end of the line. Multi-line statements can be entered simply by pressing <Return> at the end of each line. A statement is not considered complete until the program sees a semi-colon at the end of a line. An example of a multi-line statement is shown below.

```
lsql> select sale_name, sum(amount), max(amount), min(amount), avg(amount)
lsql> from path salesperson to customer to sales_order
lsql> group by sale_name;
lsql utility commands
```

The lsql program contains a set of commands to control the compilation and execution of RDM Embedded SQL statements and the retrieval of results. Each lsql utility command begins with a special character (".") in the first column of the entered statement. A summary of each command is given in the following table.

3.2.1.1 lsql Utility Commands

Command	Description
?	Display a summary of all utility commands.
.b [num | -]	Turn on bracket mode (set to num) or turn off bracket mode (set to -). Bracket mode ...
.e	Toggle the statement echo flag. When turned on, each command or statement processed by an .r command is displayed.
.f getcursor	Call SQLGetCursorName XE "SQLGetCursorName" to get the system-generated cursor name associated with the current statement handle.
.f setcursor name	Call SQLSetCursorName to set the cursor name associated with the current statement handle to name.
.h [num]	Switch to statement handle number num. Statement handles are numbered between 1 and the limit specified by the command line option -h (default is 5 per connection). The .h command by itself displays the most recently compiled and executed statement for each statement handle in the current connection.
.n	When table mode is off, this command fetches the next select statement result row.
.p par [par]...	Specify value for 1st parameter marker, 2nd parameter marker, and so on.
.q	Exits program. This closes all statement handles and connections, then terminates the program.
.r file	Process all of the commands and statements contained in file. Both utility commands and RDM Embedded SQL statements can be contained in file, including other .r commands.
.t	Toggle the select statement output display mode between table mode and row-at-a-time mode. Row-at-a-time mode is used for cursor-based retrieval and positioned updates and deletes.
.x	Re-execute the compiled statement associated with the current statement handle. Calls SQLExecute directly; the statement was compiled (through a call to SQLPrepare when it was first submitted.
.y	Toggle the compile only flag. When turned on, each SQL statement is compiled but not executed.
.z	When table mode is off, this command calls SQLCloseCursor to close the active cursor.

Some of these commands have been implemented specifically to test positioned updates and deletes. The following example illustrates how this works.

```
lsql> .t
*** table mode is off
lsql> select * from salesperson;
SALE_ID: BNF
SALE_NAME: Flores, Bob
COMMISSION: 0.135000
REGION: 0
```

```
OFFICE: 1
MGR_ID: **NULL**
lsql> .n
SALE_ID: JBW
SALE_NAME: Warner, John
COMMISSION: 0.115000
REGION: 1
OFFICE: 1
MGR_ID: **NULL**
lsql> .f getcursor
*** cursor = sql_cur_0001
lsql> .h 2
*** using statement handle #2
lsql> update salesperson set commission = 0.10
lsql>where current of sql_cur_0001;
*** 1 rows affected
lsql> commit;
```

When you are finished, enter the .q command to log out of the connected database RDM Embedded servers and terminate execution.

4. Function Reference

4.1 ODBC data access API

The RDM Embedded data access functions are a subset of the standard ODBC functions. Access to databases and tables is managed using handles that you allocate and free by using the API calls.

The data access function calls are divided into the following categories:

- Connection processing
- SQL operation
- Result set processing
- Transaction processing
- Error processing

> This chapter does not provide a complete reference to all of the parameters and uses of these functions. For complete information, refer to a current ODBC reference manual.

4.1.1 Unicode Function Arguments

Unicode functions that always return or take strings or length arguments are passed as count-of-characters. For functions that return length information for server data, the display size and precision are described in number of characters. When a length (transfer size of the data) could refer to string or non-string data, the length is described in byte lengths. For example, SQLDescribeParam will still take the length as count-of-bytes, but SQLExecDirect will use count-of-characters.

4.1.2 Connection processing

Use the following calls to connect and disconnect to a database, and to create and destroy ODBC handles

SQLSetConnectAttr	Sets RDM Embedded SQL attributes that govern aspects of connections.
SQLConnect	Used to establish a connection to a specified driver and its underlying data source. Requires a username and password.
SQLDisconnect	Used to close the data source connection associated with a specific connection handle.
SQLAllocHandle	Allocates a connection or SQL statement handle.
SQLFreeHandle	Releases a connection or SQL statement handle, freeing the handle's associated memory.
SQLSetStmtAttr	Sets RDM Embedded SQL attributes related to a statement

4.1.3 SQL operation

Use the following calls to create and execute an SQL statement:

SQLBindParameter	Binds application data buffers to columns in the result set.
SQLCloseCursor	Closes a cursor that has been opened on a SQL statement handle. If a cursor is closed while it contains results that are pending, those results are discarded.
SQLDescribeParam	Gets a description of a SQL statement parameter marker.
SQLExecDirect	Prepares and executes a SQL statement, using the current values of any parameter marker variables that have been bound to the SQL statement.
SQLExecute	Executes a SQL statement that has been successfully prepared (by the SQLPrepare function) using the current values of any parameter marker variables that have been bound to the SQL statement.
SQLGetCursorName	Retrieves the name of the cursor associated with a specific SQL statement handle.
SQLNumParams	Gets the number of parameters in a prepared SQL statement.
SQLPrepare	Sends a SQL statement to a data source so it can be prepared (compiled) for execution.
SQLRowCount	Obtains a count of the number of rows in a table that were affected by an insert, update, or delete operation.
SQLSetCursorName	Associates a cursor name with an active statement.

4.1.4 Result set processing

Use the following calls to retrieve the data resulting from a SQL statement:

SQLBindCol	Associates (binds) parameter markers in a SQL statement with application variables.
SQLDescribeCol	Retrieves the basic result data set meta-data (specifically, column name, SQL data type, column size, decimal precision, and nullability) for a specified column in a result data set.
SQLFetch	Advances a cursor to the next row of data in a result data set and retrieves data from any bound columns that exist for that row into their associated application variables.
SQLNumResultCols	Determines the number of columns that exist in a result data set.

4.1.5 Transaction processing

Use the following call to commit or rollback the current transaction:

SQLEndTran	Requests a commit or rollback operation for all active transactions associated with a specific connection handle.

4.1.6 Error processing

Use the following calls to retrieve additional information on system errors:

SQLGetDiagField	Retrieves the current value of a field in a diagnostic header or status record (associated with a specific connection or statement handle).
SQLGetDiagRec	Retrieves the current values of several commonly used fields of a diagnostic status record. This record contains error, warning, and/or status information generated by the last ODBC API function executed.

Summary Listing of SQL ODBC API Functions

SQLAllocHandle

Allocates an environment, connection, or SQL handle

Syntax

```
SQLRETURN SQLAllocHandle (
       SQLSMALLINT HandleType,
       SQLHANDLE InputHandle,
       SQLHANDLE *OutputHandle)
```

Description

This function allocates an environment, connection, or SQL statement handle and its associated resources.

RDM Embedded does not require you to allocate an environment handle before allocating connection or statement handles, and doing so currently has no effect. The environment handle type is included only for ODBC compatibility.

Parameters

HandleType

[Input] The type of handle to allocate memory for. Types are listed below:

SQL_HANDLE_DBC

SQL_HANDLE_STMT

InputHandle

[Input] The existing handle in whose context the environment, connection, or statement handle is allocated. Use NULL to allocate DBC handles.

OutputHandle

[Output] A pointer to a location in memory where this function is to store the new handle.

Return Values

SQL_ERROR

SQL_INVALID_HANDLE

SQL_SUCCESS

Example

```
// This demonstrates SQLAllocHandle and SQLFreeHandle
// with DBC handles
void AllocAndFreeDBC(void)
{
        SQLHANDLE hDBC;
        SQLRETURN rc;



        // Allocate the DBC handle
        rc = SQLAllocHandle(SQL_HANDLE_DBC, NULL, &hDBC);
        if (!SQL_SUCCEEDED(rc))
        {
                // ERROR !!
        }
        // Free the DBC handle
        rc = SQLFreeHandle(SQL_HANDLE_DBC, hDBC);
        if (!SQL_SUCCEEDED(rc))
        {
                // ERROR !!
        }
}
```

SQLBindCol

Binds SQL parameter markers with application variables

Syntax

```
SQLRETURN SQLBindCol (
       SQLHSTMT StatementHandle,
       SQLUSMALLINT ColumnNumber,
       SQLSMALLINT TargetType,
       SQLPOINTER TargetValue,
       SQLINTEGER BufferLength,
       SQLINTEGER *StrLen_or_Ind)
```

Description

This function binds application data buffers to columns in the result set.

Parameters

StatementHandle

[Input] A SQL statement handle, previously allocated by calling SQLAllocHandle.

ColumnNumber

[Input] The column´s location in the result data set. Columns are numbered sequentially from left to right, starting with 1, as they appear in the result data set.

TargetType

[Input] The data type of the value buffer that the column data being retrieved is to be stored in.

TargetValue

[Output] A pointer to a location in memory where the driver is to store column data when it is retrieved (fetched) from the result data set or where the application is to store column data that is to be written to a data source with a positioned UPDATE or DELETE operation.

BufferLength

[Input] The size of the buffer.

StrLen_or_Ind

[Output] A pointer to a location in memory where this function is to store either the size of the data value associated with the column or a special indicator value associated with the column data.

Return Values

SQL_ERROR

SQL_INVALID_HANDLE

SQL_SUCCESS

Example

```
// This demonstrates SQLBindCol,
// SQLFetch, and SQLNumResultCols
void UseBindColFetchAndNumResultCols(void)
{
        SQLHANDLE hDBC;
        SQLHANDLE hStmt;
        SQLRETURN rc;
        SQLCHAR szCField[20];
        SQLINTEGER iCFieldInd = 0;
        SQLINTEGER lIField;
        SQLINTEGER lIFieldInd = 0;
        SQLSMALLINT sColumnCount;



        // Allocate the DBC handle
        rc = SQLAllocHandle(SQL_HANDLE_DBC, NULL, &hDBC);
        if (!SQL_SUCCEEDED(rc))
        {
                // ERROR !!
        }


        // Connect to the data source
        rc = SQLConnect(hDBC, "YourDataSource", SQL_NTS, "YourUser", SQL_NTS,
                                "YourPassword", SQL_NTS);
        if (!SQL_SUCCEEDED(rc))
        {
                // ERROR !!
        }
```

```
        // Allocate the statement handle
        rc = SQLAllocHandle(SQL_HANDLE_STMT, hDBC, &hStmt);
        if (!SQL_SUCCEEDED(rc))
        {
                // ERROR !!
        }



        // Prepare a SQL statement, using the previously allocated
        // statement handle
        rc = SQLPrepare(hStmt, "SELECT CHAR_FIELD, INT_FIELD FROM YOUR_TABLE"),
                                        SQL_NTS;
        if (!SQL_SUCCEEDED(rc))
        {
                // ERROR !!
        }

        // Execute the prepared SQL statement
        rc = SQLExecute(hStmt);
        if (!SQL_SUCCEEDED(rc))
        {
                // ERROR !!
        }

        // Get the number of result columns. In this example
        // it should be 2 (CHAR_FIELD and INT_FIELD) tmt, &sColumnCount);
        rc = SQLNumResultCols(hS (!SQL_SUCCEEDED(rc))
        if
        {
                // ERROR !!
        }


        // Bind a string to the character field
        rc = SQLBindCol(hStmt, 1, SQL_C_CHAR, szCField,
                        sizeof(szCField), &iCFieldInd);
        if (!SQL_SUCCEEDED(rc))
        {
                // ERROR !!
        }


        // Bind a long integer (SQLINTEGER) to the integer field
        rc = SQLBindCol(hStmt, 2, SQL_C_SLONG, &lIField,
```

```
                        sizeof(lIField), &lIFieldInd);
        if (!SQL_SUCCEEDED(rc))
        {
                // ERROR !!
        }


        // Now fetch the first row. The row's character field will
        // end up in szCField, and the integer field in lIField
        rc = SQLFetch(hStmt);
        if (!SQL_SUCCEEDED(rc))
        {
                // ERROR !!
        }


        // Free the statement handle
        rc = SQLFreeHandle(SQL_HANDLE_STMT, hStmt);
        if (!SQL_SUCCEEDED(rc))
        {
                // ERROR !!
        }


        // Disconnect from the data source
        rc = SQLDisconnect(hDBC);
        if (!SQL_SUCCEEDED(rc))
        {
                // ERROR !!
        }


        // Free the DBC handle
        rc = SQLFreeHandle(SQL_HANDLE_DBC, hDBC);
        if (!SQL_SUCCEEDED(rc))
        {
                // ERROR !!
        }
}
```

SQLBindParameter

Binds SQL parameter markers with application variables

Syntax

```
SQLRETURN SQLBindParameter (
        SQLHSTMT StatementHandle,
        SQLUSMALLINT ParameterNumber,
        SQLSMALLINT InputOutputType,
        SQLSMALLINT ValueType,
        SQLSMALLINT ParameterType,
        SQLUINTEGER ColumnSize,
        SQLSMALLINT DecimalDigits,
        SQLPOINTER ParameterValue,
        SQLINTEGER BufferLength,
        SQLINTEGER *StrLen_or_Ind)
```

Description

This function associates (binds) parameter markers in an SQL statement with application variables.

Parameters

StatementHandle

[Input] A SQL statement handle, previously allocated by calling SQLAllocHandle.

ParameterNumber

[Input] The parameter number in the SQL statement text. Parameter numbers are numbered sequentially from left to right, starting with 1, as they appear in the SQL statement.

InuptOutputType

[Input] The type of parameter being bound.

ValueType

[Input] The C language data type of the parameter being bound.

ParameterType

[Input] The SQL data type of the parameter being bound.

ColumnSize

[Input] The size of the column or expression of the corresponding parameter marker.

DecimalDigits

[Input] The decimal digits of the column or expression in the corresponding parameter marker.

ParameterValue

[Output] A pointer to a location in memory where the value associated with the parameter marker is stored.

BufferLength

[Input] The size of the buffer.

StrLen_or_Ind

[Output] A pointer to a buffer for the parameter length.

Return Values

SQL_ERROR

SQL_INVALID_HANDLE

SQL_SUCCESS

Example

See also the example for SQLDescribeParam.

```
// This demonstrates SQLBindParameter
// and SQLCloseCursor

void BindParamAndCloseCursor(void)
{
        SQLHANDLE hDBC;
        SQLHANDLE hStmt;
        SQLRETURN rc;
        SQLINTEGER lParam1;
        SQLINTEGER lParam1Ind = 0;
```

```
        // Allocate the DBC handle
        rc = SQLAllocHandle(SQL_HANDLE_DBC, NULL, &hDBC);
        if (!SQL_SUCCEEDED(rc))
        {
                // ERROR !!
        }



        // Connect to the data source
        rc = SQLConnect(hDBC, "YourDataSource", SQL_NTS,
                                        "YourUser", SQL_NTS,
                                        "YourPassword", SQL_NTS);
        if (!SQL_SUCCEEDED(rc))
        {
                // ERROR !!
        }



        // Allocate the statement handle
        rc = SQLAllocHandle(SQL_HANDLE_STMT, hDBC, &hStmt);
        if (!SQL_SUCCEEDED(rc))
        {
                // ERROR !!
        }



        // Prepare a SQL statement that uses a bound parameter,
        // using the previously allocated statement handle
        rc = SQLPrepare(hStmt, "SELECT * FROM YOUR_TABLE
                                                        WHERE INT_FIELD = ?",
                                                        SQL_NTS);
        if (!SQL_SUCCEEDED(rc))
        {
                // ERROR !!
        }



        // Bind an integer to the parameter
        rc = SQLBindParameter(hStmt, 1, SQL_PARAM_INPUT,
                                                        SQL_C_SLONG, SQL_INTEGER, 0, 0,
                                                        &lParam1, sizeof(lParam1),
                                                        &lParam1Ind);
        if (!SQL_SUCCEEDED(rc))
```

```
        {
                // ERROR !!
        }


        ue for the parameter

        // Set the vallParam1 = 27;



        // Execute the prepared SQL statement   // The value, 27, will replace the '?' in
        // the query.
        rc = SQLExecute(hStmt);
        if (!SQL_SUCCEEDED(rc))
        {
                // ERROR !!
        }



        // Fetch the first record
        rc = SQLFetch(hStmt);
        if (!SQL_SUCCEEDED(rc))
        {
                // ERROR !!
        }



        // Now close the cursor. This makes the statement
        // handle usable for a different query
        rc = SQLCloseCursor(hStmt);
        if (!SQL_SUCCEEDED(rc))
        {
                // ERROR !!
        }


        // Free the statement handle
        rc = SQLFreeHandle(SQL_HANDLE_STMT, hStmt);
        if (!SQL_SUCCEEDED(rc))
        {
                // ERROR !!
        }
```

```
        // Disconnect from the data source
        rc = SQLDisconnect(hDBC);
        if (!SQL_SUCCEEDED(rc))
        {
                // ERROR !!
        }



        // Free the DBC handle
        rc = SQLFreeHandle(SQL_HANDLE_DBC, hDBC);
        if (!SQL_SUCCEEDED(rc))
        {
                // ERROR !!
        }
}
```

SQLCloseCursor

Closes a cursor opened on a SQL statement handle

Syntax

```
SQLRETURN SQLCloseCursor (
       SQLHSTMT StatementHandle)
```

Description

This function closes a cursor that was opened on a SQL statement handle. If a cursor is closed while it contains pending results, those results are discarded.

Parameters

StatementHandle

[Input] A SQL statement handle, previously allocated by calling SQLAllocHandle.

Return Values

SQL_ERROR

SQL_INVALID_HANDLE

SQL_SUCCESS

Example

See the SQLBindParameter example.

SQLConnect

Establishes a connection to a driver

Syntax

```
SQLRETURN SQLConnect (
       SQLHDBC ConnectionHandle,
       SQLCHAR *ServerName,
       SQLSMALLINT NameLength1,
       SQLCHAR *UserName,
       SQLSMALLINT NameLength2,
       SQLCHAR *Authentication,
       SQLSMALLINT NameLength3)
```

Description

This function is used to establish a connection to a specified driver and its underlying data source.

Parameters

ConnectionHandle

[Input] A data source connection handle, previously allocated by calling SQLAllocHandle with a SQL_HANDLE_DBC handle type.

ServerName

[Input] A pointer to a memory location where the name of the server to connect to is stored.

NameLength1

[Input] The length of the server name value, in bytes, stored in the ServerName parameter or SQL_NTS for a NULL terminated string.

UserName

[Input] A pointer to a memory location where the user´s authorization name is stored.

NameLength2

[Input] The length of the user authorization name value, in bytes, stored in the

UserName parameter or SQL_NTS for a NULL terminated string.

Authentication

[Input] A pointer to a location in memory where the authentication for the specified user-name is stored.

NameLength3

[Input] The length of the authorization value in bytes stored in the Authentication parameter or SQL_NTS for a NULL terminated string.

Return Values

SQL_ERROR

SQL_INVALID_HANDLE

SQL_SUCCESS

Example

```
// This demonstrates SQLConnect and SQLDisconnect


void ConnectDisconnect(void)
{
       SQLHANDLE hDBC;
       SQLRETURN rc;



       // Allocate the DBC handle
       rc = SQLAllocHandle(SQL_HANDLE_DBC, NULL, &hDBC);
       if (!SQL_SUCCEEDED(rc))
       {
              // ERROR !!
       }


       // Connect to the data source
       rc = SQLConnect(hDBC, "YourDataSource", SQL_NTS,
                                               "YourUser", SQL_NTS,
                                               "YourPassword", SQL_NTS);
       if (!SQL_SUCCEEDED(rc))
       {
              // ERROR !!
```

```
        }



        // Disconnect from the data source
        rc = SQLDisconnect(hDBC);
        if (!SQL_SUCCEEDED(rc))
        {
                // ERROR !!
        }


        // Free the DBC handle
        rc = SQLFreeHandle(SQL_HANDLE_DBC, hDBC);
        if (!SQL_SUCCEEDED(rc))
        {
                // ERROR !!
        }
}
```

SQLDescribeCol

Retrieves the basic result data set for a column

Syntax

```
SQLRETURN SQLDescribeCol (
       SQLHSTMT StatementHandle,
       SQLUSMALLINT ColumnNumber,
       SQLCHAR *ColumnName,
       SQLSMALLINT BufferLength,
       SQLSMALLINT *NameLength,
       SQLSMALLIINT *DataType,
       SQLUINTEGER *ColumnSize,
       SQLSMALLINT *DecimalDigits,
       SQLSMALLINT *Nullable)
```

Description

This function retrieves the basic meta-data result set (specifically, column name, SQL data type, column size, decimal precision, and nullability) for a specified column.

Parameters

StatementHandle

[Input] A SQL statement handle, previously allocated by calling SQLAllocHandle.

ColumnNumber

[Input] The column´s location in the result data set. Columns are numbered sequentially from left to right, starting with 1, as they appear in the result data set.

ColumnName

[Input] A pointer to a location in memory where this function is to store the name of the specified column.

BufferLength

[Input] The size of the buffer.

NameLength

[Output] Pointer to a buffer in which to return the length of the string available to return in ColumnName

DataType

[Output] A pointer to a location in memory where this function is to store the data type of the specified column.

ColumnSize

[Output] A pointer to a location in memory where this function is to store the maximum length, in bytes, of the column as it is defined in the data source.

DecimalDigits

[Output] A pointer to a location in memory where this function is to store the number of digits of the column on the data source.

Nullable

[Output] A pointer to a location in memory where this function is to store information about whether the column accepts/allows NULL values.

Return Values

SQL_ERROR

SQL_HANDLE_STMT

SQL_INVALID_HANDLE

SQL_SUCCESS

SQL_SUCCESS_WITH_INFO

Example

```
// This demonstrates SQLDescribeCol
void DescribeCol(void)
{
       SQLHANDLE hDBC;
       SQLHANDLE hStmt;
       SQLRETURN rc;
       SQLWCHAR szColName[20];
       SQLSMALLINT sColNameBytes;
       SQLSMALLINT sDataType;
       SQLUINTEGER uiColumnSize;
       SQLSMALLINT sDecimalDigits;
```

```
        SQLSMALLINT sNullable;


        // Allocate the DBC handle
        rc = SQLAllocHandle(SQL_HANDLE_DBC, NULL, &hDBC);
        if (!SQL_SUCCEEDED(rc))
        {
                // ERROR !!
        }



        // Connect to the data source
        rc = SQLConnect(hDBC, "YourDataSource", SQL_NTS,
                                        "YourUser", SQL_NTS,
                                        "YourPassword", SQL_NTS);
        if (!SQL_SUCCEEDED(rc))
        {
                // ERROR !!
        }



        // Allocate the statement handle
        rc = SQLAllocHandle(SQL_HANDLE_STMT, hDBC, &hStmt);
        if (!SQL_SUCCEEDED(rc))
        {
                // ERROR !!
        }


        // Prepare a SQL statement, using the previously allocated
        // statement handle
        rc = SQLPrepare(hStmt, "SELECT * FROM YOUR_TABLE",
                                        SQL_NTS);
        if (!SQL_SUCCEEDED(rc))
        {
                // ERROR !!
        }


        // Execute the prepared SQL statement
        rc = SQLExecute(hStmt);
        if (!SQL_SUCCEEDED(rc))
        {
                // ERROR !!
        }
```

```
        // Get the meta-data information about the first column
        // l call,
        // in the result set. After a successfu
        // szColName contains the column name, ull
        // sColNameBytes contains the count of bytes in the f  // column name, sDataType cont
        // of the column (SQL_WCHAR, etc), uiColumnSize contains
        // the column size, sDecimalDigits contains the number of
        // decimal digits, and sNullable will be SQL_NO_NULLS,
        // SQL_NULLABLE, or SQL_NULLABLE_UNKNOWN


        rc = SQLDescribeCol(hStmt, 1, szColName,
                                                    sizeof(szColName), &sColNameBytes,
                                                    &sDataType, &uiColumnSize,
                                                    &sDecimalDigits, &sNullable);
        if (!SQL_SUCCEEDED(rc))
        {
                // ERROR !!
        }



        // Free the statement handle
        rc = SQLFreeHandle(SQL_HANDLE_STMT, hStmt);
        if (!SQL_SUCCEEDED(rc))
        {
                // ERROR !!
        }


        // Disconnect from the data source
        rc = SQLDisconnect(hDBC);
        if (!SQL_SUCCEEDED(rc))
        {
                // ERROR !!
        }



        // Free the DBC handle
        rc = SQLFreeHandle(SQL_HANDLE_DBC, hDBC);
        if (!SQL_SUCCEEDED(rc))
        {
                // ERROR !!
        }
}
```

SQLDescribeParam

Retrieves description of a parameter marker

Syntax

```
SQLRETURN SQLDescribeParam (
       SQLHSTMT StatementHandle,
       SQLUSMALLINT ParamNumber,
       SQLSMALLINT *ParamType,
       SQLUINTEGER *ParamSize,
       SQLSMALLINT *DecimalDigits,
       SQLSMALLINT *Nullable)
```

Description

This function returns the description of a parameter marker associated with a prepared SQL statement.

Parameters

StatementHandle

[Input] A SQL statement handle, previously allocated by calling SQLAllocHandle.

ParamNumber

[Input] Specifies the parameter number in the SQL statement text. Parameter numbers are numbered sequentially from left to right, starting with 1, as they appear in the SQL statement.

ParamType

[Output] The SQL data type of the parameter being requested.

ParamSize

[Output] Pointer to a buffer in which to return the size of the column or expression of the corresponding parameter marker.

DecimalDigits

[Output] Pointer to a buffer in which to return the number of decimal digits of the column or expression of the corresponding parameter marker.

Nullable

[Output] A pointer to a location in memory where this function is to store information about whether the parameter accepts/allows NULL values.

Return Values

SQL_SUCCESS

SQL_ERROR

SQL_INVALID_HANDLE

Example

```
SQLHANDLE hdbc;
SQLHSTMT hstmt;



SQLSMALLINT num, type, scale, nullable, rgb;
SQLINTEGER nts = SQL_NTS;
SQLUINTEGER prec;



if (SQL_SUCCEEDED(ret))
               ret = SQLAllocHandle(SQL_HANDLE_DBC, NULL, &hdbc);



// should continue checking return values in a normal program
SQLConnect(hdbc,       (SQLCHAR *)"SALES", SQL_NTS,
                                    (SQLCHAR*)"admin", SQL_NTS, (SQLCHAR *)"admin",
                                    SQL_NTS);



SQLAllocHandle(SQL_HANDLE_STMT, hdbc, &hstmt);


SQLPrepare(hstmt, (SQLCHAR *)
                                "insert into salesperson(sale_id, sale_name, dob
                                                        values('jld', 'Jane Doe', ?)"), SQL_N



SQLNumParams(hstmt, &num);
```

```
// describe and bind first bound parameter
if (num == 1)
               SQLDescribeParam(hstmt, num, &type, &prec, &scale,
                                &nullable);



SQLBindParameter(hstmt, 1, SQL_PARAM_INPUT, SQL_C_CHAR, type,
                                prec, scale, "1974-02-21", 11, &nts);



SQLExecute(hstmt);


SQLEndTran(SQL_HANDLE_DBC, hdbc, SQL_COMMIT);
```

SQLDisconnect

Closes data source connection of a specified handle

Syntax

```
SQLRETURN SQLDisconnect (
       SQLHDBC ConnectionHandle)
```

Description

This function is used to close the data source connection associated with a specific connection handle.

Parameters

ConnectionHandle

[Input] A data source connection handle, previously allocated by calling SQLAllocHandle.

Return Values

SQL_ERROR

SQL_INVALID_HANDLE

SQL_SUCCESS

SQL_SUCCESS_WITH_INFO

Example

See the SQLConnect example.

SQLEndTran

Requests a commit or rollback for active transactions

Syntax

```
SQLRETURN SQLEndTran (
       SQLSMALLINT HandleType,
       SQLHANDLE Handle,
       SQLSMALLINT CompletionType)
```

Description

This function requests a commit or a rollback operation for all active transactions associated with a specific connection handle.

Parameters

HandleType

Specifies which type of handle to request a commit or rollback operation for. The handle type must be SQL_HANDLE_DBC.

Handle

The connection handle whose transaction(s) are to be terminated.

CompletionType

Specifies which type of action to use to terminate the current transaction (SQL_COMMIT or SQL_ROLLBACK).

Return Values

SQL_ERROR

SQL_INVALID_HANDLE

SQL_SUCCESS

SQL_SUCCESS_WITH_INFO

Example

```
// This demonstrates SQLEndTran
void EndTran(void)
{
       SQLHANDLE hDBC;
       SQLHANDLE hStmt;
       SQLRETURN rc;


       // Allocate the DBC handle
       rc = SQLAllocHandle(SQL_HANDLE_DBC, NULL, &hDBC);
       if (!SQL_SUCCEEDED(rc))
       {
              // ERROR !!
       }


       // Connect to the data source
       rc = SQLConnect(hDBC,   "YourDataSource", SQL_NTS,
                                                 "YourUser", SQL_NTS,
                                                 "YourPassword", SQL_NTS);
       if (!SQL_SUCCEEDED(rc))
       {
              // ERROR !!
       }


       // Allocate the statement handle
       rc = SQLAllocHandle(SQL_HANDLE_STMT, hDBC, &hStmt);
       if (!SQL_SUCCEEDED(rc))
       {
              // ERROR !!
       }



       // Execute a SQL insert statement, using the
       // previously allocated statement handle

       rc = SQLExecDirect(hStmt, "INSERT INTO YOUR_TABLE
                                                          (CHAR_FIELD, INT_FIELDVALUES('ABC', 1
                                                 SQL_NTS);

       if (!SQL_SUCCEEDED(rc))
       {
              // ERROR !!
```

```
        }


        // Free the statement handle
        rc = SQLFreeHandle(SQL_HANDLE_STMT, hStmt);
        if (!SQL_SUCCEEDED(rc))
        {
                // ERROR !!
        }



        // Now roll back the insertion
        rc = SQLEndTran(SQL_HANDLE_DBC, hDBC, SQL_ROLLBACK);
        if (!SQL_SUCCEEDED(rc))
        {
                // ERROR !!
        }
  a source
        // Disconnect from the dat;
        rc = SQLDisconnect(hDBC)  if (!SQL_SUCCEEDED(rc))
        {
                // ERROR !!
        }



        // Free the DBC handle
        rc = SQLFreeHandle(SQL_HANDLE_DBC, hDBC);
        if (!SQL_SUCCEEDED(rc))
        {
                // ERROR !!
        }
}
```

SQLExecDirect

Prepares and executes an SQL statement

Syntax

```
SQLRETURN SQLExecDirect (
       SQLHSTMT StatementHandle,
       SQLCHAR *StatementText,
       SQLINTEGER TextLength)
```

Description

Prepares and executes an SQL statement, using the current values of any parameter marker variables that are bound to the SQL statement.

Parameters

StatementHandle

A SQL statement handle, previously allocated by calling SQLAllocHandle.

StatementText

A pointer to a location in memory where the SQL statement to be prepared and executed is stored.

TextLength

The length of the SQL statement in bytes stored in the StatementText parameter or SQL_NTS for a NULL terminated string.

Return Values

SQL_ERROR

SQL_INVALID_HANDLE

SQL_SUCCESS

SQL_SUCCESS_WITH_INFO

Example

```
// This demonstrates SQLExecDirect
void ExecDirect(void)
{
       SQLHANDLE hDBC;
       SQLHANDLE hStmt;
       SQLRETURN rc;


       // Allocate the DBC handle
       rc = SQLAllocHandle(SQL_HANDLE_DBC, NULL, &hDBC);
       if (!SQL_SUCCEEDED(rc))
       {
              // ERROR !!
       }


       // Connect to the data source
       rc = SQLConnect(hDBC,   "YourDataSource", SQL_NTS,
                                                 "YourUser", SQL_NTS,
                                                 "YourPassword", SQL_NTS);
       if (!SQL_SUCCEEDED(rc))
       {
              // ERROR !!
       }


       // Allocate the statement handle
       rc = SQLAllocHandle(SQL_HANDLE_STMT, hDBC, &hStmt);
       if (!SQL_SUCCEEDED(rc))
       {
              // ERROR !!
       }


       // Execute a SQL statement, using the previously allocated
       // statement handle
       rc = SQLExecDirect(hStmt,       "SELECT * FROM YOUR_TABLE",
                                                       SQL_NTS);
       if (!SQL_SUCCEEDED(rc))
       {
              // ERROR !!
       }


       // Free the statement handle
```

```
        rc = SQLFreeHandle(SQL_HANDLE_STMT, hStmt);
        if (!SQL_SUCCEEDED(rc))
        {
                // ERROR !!
        }

        // Disconnect from the data source
        rc = SQLDisconnect(hDBC);
        if (!SQL_SUCCEEDED(rc))
        {
                // ERROR !!
        }


        // Free the DBC handle ANDLE_DBC, hDBC);
        rc = SQLFreeHandle(SQL_H (!SQL_SUCCEEDED(rc))
        if  {
                // ERROR !!
        }
}
```

SQLExecute

Executes a previously prepared SQL statement

Syntax

```
SQLRETURN SQLExecute (
       SQLHSTMT StatementHandle)
```

Description

This function executes an SQL statement that has been successfully prepared (by the SQLPrepare function) using the current values of any parameter marker variables bound to the SQL statement.

Parameter

StatementHandle

[Input] An SQL statement handle, previously allocated by calling SQLAllocHandle.

Return Values

SQL_ERROR

SQL_INVALID_HANDLE

SQL_SUCCESS

SQL_SUCCESS_WITH_INFO

Example

```
// This demonstrates SQLPrepare and SQLExecute
void PrepareAndExecute(void)
{
       SQLHANDLE hDBC;
       SQLHANDLE hStmt;
       SQLRETURN rc;


       // Allocate the DBC handle
```

```
        rc = SQLAllocHandle(SQL_HANDLE_DBC, NULL, &hDBC);
        if (!SQL_SUCCEEDED(rc))
        {
                // ERROR !!
        }


        // Connect to the data source
        rc = SQLConnect(hDBC,   "YourDataSource", SQL_NTS,
                                                  "YourUser", SQL_NTS,
                                                  "YourPassword", SQL_NTS);
        if (!SQL_SUCCEEDED(rc))
        {
                // ERROR !!
        }


        // Allocate the statement handle
        rc = SQLAllocHandle(SQL_HANDLE_STMT, hDBC, &hStmt);
        if (!SQL_SUCCEEDED(rc))
        {
                // ERROR !!
        }


        // Prepare a SQL statement, using the previously allocated
        // statement handle
        rc = SQLPrepare(hStmt, "SELECT * FROM YOUR_TABLE",
                                                  SQL_NTS);
        if (!SQL_SUCCEEDED(rc))
        {
                // ERROR !!
        }


        // Execute the prepared SQL statement
        rc = SQLExecute(hStmt);
        if (!SQL_SUCCEEDED(rc))
        {
                // ERROR !!
        }


        // Free the statement handle
        rc = SQLFreeHandle(SQL_HANDLE_STMT, hStmt);
        if (!SQL_SUCCEEDED(rc))
        {
                // ERROR !!
```

```
        }

        a source
        // Disconnect from the dat;
        rc = SQLDisconnect(hDBC)  if (!SQL_SUCCEEDED(rc))
        {
                // ERROR !!
        }


        // Free the DBC handle
        rc = SQLFreeHandle(SQL_HANDLE_DBC, hDBC);
        if (!SQL_SUCCEEDED(rc))
        {
                // ERROR !!
        }
}
```

SQLFetch

Fetches data from a bound column to an application variable

Syntax

```
SQLRETURN SQLFetch (
       SQLHSTMT StatementHandle)
```

Description

This function advances a cursor to the next row of data in a result data set, and retrieves data from any bound columns that exist for that row into the associated application variables.

Parameters

StatementHandle

[Input] An SQL statement handle, previously allocated by calling SQLAllocHandle.

Return Values

SQL_ERROR

SQL_INVALID_HANDLE

SQL_NO_DATA

SQL_NO_DATA_FOUND

SQL_SUCCESS

SQL_SUCCESS_WITH_INFO

Example

See the SQLBindCol example.

SQLFreeHandle

Frees a statement handle

Syntax

```
SQLRETURN SQLFreeHandle (
       SQLSMALLINT HandleType,
       SQLHANDLE Handle)
```

Description

This function releases a connection or statement handle and frees any memory associated with it.

Parameters

HandleType

[Input] Specifies the type of handle that the memory to be freed is associated with (SQL_HANDLE_DBC or SQL_HANDLE_STMT).

Return Values

SQL_ERROR

SQL_INVALID_HANDLE

SQL_SUCCESS

Example

See the example for SQLAllocHandle.

SQLGetCursorName

Retrieves the cursor name of an SQL statement handle

Syntax

```
SQLRETURN SQLGetCursorName (
       SQLHSTMT StatementHandle,
       SQLCHAR *CursorName,
       SQLSMALLINT BufferLength,
       SQLSMALLINT *NameLength)
```

Description

This function retrieves the name of the cursor associated with a specific SQL statement handle.

Parameters

StatementHandle

[Input] A SQL statement handle, previously allocated by calling SQLAllocHandle.

CursorName

[Output] A pointer to a location in memory where this function is to store the cursor name retrieved.

BufferLength

[Input] The size of the buffer where this function is to store the cursor name retrieved.

NameLength

[Output] A pointer to a location in memory where this function is to store the actual length of the string available to return in CursorName.

Return Values

SQL_ERROR

SQL_INVALID_HANDLE

SQL_SUCCESS

SQL_SUCCESS_WITH_INFO

Example

See the example for SQLSetCursorName.

SQLGetDiagField

Retrieves current field value of a status record

Syntax

```
SQLRETURN SQLGetDiagField (
       SQLSMALLINT HandleType,
       SQLHANDLE Handle,
       SQLSMALLINT RecNumber,
       SQLSMALLINT DiagIdentifier,
       SQLPOINTER DiagInfo,
       SQLSMALLINT BufferLength,
       SQLSMALLINT *StringLength)
```

Description

This function retrieves the current value of a field associated with a specific connection or statement handle, in a diagnostic header or status record.

Parameters

HandleType

[Input] Specifies which type of handle to retrieve diagnostic information for (SQL_HANDLE_DBC or SQL_HANDLE_STMT).

Handle

[Input] A connection or SQL statement handle, previously allocated by calling SQLAllocHandle.

RecNumber

[Input] Specifies the diagnostic status record from which SQLGetDiagField retrieves information.

DiagIdentifier

[Input] The field of the diagnostic header record or status record whose value is to be retrieved.

DiagInfo

[Output] A pointer to a location in memory where this function is to store the data source- specific error code retrieved.

BufferLength

[Input] The size of the buffer.

StringLength

[Output] A pointer to a location in memory where SQLGetDiagField stores the length of the string available to return in DiagInfo.

Return Values

SQL_ERROR

SQL_INVALID_HANDLE

SQL_NO_DATA

SQL_SUCCESS

SQL_SUCCESS_WITH_INFO

Example

```
// This demonstrates SQLGetDiagField and SQLGetDiagRec
// void GetDiagFieldAndRec(void)
{
       SQLHANDLE hDBC;
       SQLHANDLE hStmt;
       SQLRETURN rc;
       SQLWCHAR szState[6];
       SQLSMALLINT sBytes;
       SQLINTEGER lNative;
       SQLWCHAR szMsg[200];


// Allocate the DBC handle
  rc = SQLAllocHandle(SQL_HANDLE_DBC, NULL, &hDBC);
  if (!SQL_SUCCEEDED(rc))
  {
       // ERROR !!
  }


// Connect to the data source
```

```
        rc = SQLConnect(hDBC, "YourDataSource", SQL_NTS,
                                        "YourUser", SQL_NTS,
                                        "YourPassword", SQL_NTS);
    if (!SQL_SUCCEEDED(rc))
    {
        // ERROR !!
    }


  // Allocate the statement handle
    rc = SQLAllocHandle(SQL_HANDLE_STMT, hDBC, &hStmt);
    if (!SQL_SUCCEEDED(rc))
    {
        // ERROR !!
    }


  // Execute a SQL select statement that should fail,
    // using the previously allocated statement handle
    rc = SQLExecDirect(hStmt, "SELECT * FROM INVALID_TABLE",
                        SQL_NTS);


  // Get just the SQLSTATE error code
    rc = SQLGetDiagField(SQL_HANDLE_STMT, hStmt, 1,
                                        SQL_DIAG_SQLSTATE, szState,
                                        sizeof(szState), &sBytes);
    if (!SQL_SUCCEEDED(rc))
    {
        // ERROR !!
    }

    // Get the entire diagnostic record. This will get the
    // SQLSTATE, Native error code, and error text
    rc = SQLGetDiagRec(SQL_HANDLE_STMT, hStmt, 1, szState,
                                                &lNative, szMsg, sizeof(szMsg),
                        bytes);
                        (!SQL_SUCCEEDED(rc))
    if
    {     // ERROR !!
    }


  // Free the statement handle
    rc = SQLFreeHandle(SQL_HANDLE_STMT, hStmt);
    if (!SQL_SUCCEEDED(rc))
    {
        // ERROR !!
```

```
    }


  // Disconnect from the data source
    rc = SQLDisconnect(hDBC);
    if (!SQL_SUCCEEDED(rc))
    {
         // ERROR !!
    }


  // Free the DBC handle
    rc = SQLFreeHandle(SQL_HANDLE_DBC, hDBC);
    if (!SQL_SUCCEEDED(rc))
    {
         // ERROR !!
    }
  }
```

SQLGetDiagRec

Retrieves current values of several common fields of a status record

Syntax

```
SQLRETURN SQLGetDiagRec (
       SQLSMALLINT HandleType,
       SQLHANDLE Handle,
       SQLSMALLINT RecNumber,
       SQLCHAR *Sqlstate,
       SQLINTEGER *NativeError,
       SQLCHAR *MessageText,
       SQLSMALLINT BufferLength,
       SQLSMALLINT *TextLength)
```

Description

This function retrieves the current values of several commonly used fields of a diagnostic status record. This record contains error, warning, and/or status information generated by the last ODBC API function executed.

Parameters

HandleType

[Input] Specifies which type of handle to retrieve diagnostic information for.

Handle

[Input] A connection or SQL statement handle, previously allocated by calling **SQLAllocHandle**.

RecNumber

[Input] Specifies the diagnostic status record from which this function is to retrieve information.

Sqlstate

[Output] A pointer to a location in memory where **SQLGetDiagRec** stores the SQL state value retrieved.

NativeError

[Output] A pointer to a location in memory where this function is to store the data source-

specific error code retrieved.

MessageText

[Output] A pointer to a location in memory where **SQLGetDiagRec** stores the message text retrieved.

BufferLength

[Input] The length of the storage buffer where this function is to store the message text retrieved.

TextLength

[Output] A pointer to a location in memory where **SQLGetDiagRec** stores the length to be written into the MessageText buffer.

Return Values

SQL_ERROR

SQL_INVALID_HANDLE

SQL_SUCCESS

SQL_SUCCESS_WITH_INFO

Example

See the example for SQLGetDiagField.

SQLNumParams

Determines the number of parameters in a prepared statement

Syntax

```
SQLRETURN SQL_API SQLNumParams (
       SQLHSTMT StatementHandle,
       SQLSMALLINT *ParamCount)
```

Description

This function returns the number of parameters in a prepared SQL statement.

Parameters

StatementHandle

[Input] A SQL statement handle, previously allocated by calling SQLAllocHandle.

ParamCount

[Output] A pointer to a location in memory where is SQLNumParams stores the number of parameters found in the statement text associated with StatementHandle.

Return Values

SQL_SUCCESS

SQL_ERROR

SQL_INVALID_HANDLE

Example

See the example for SQLDescribeParam.

SQLNumResultCols

Determines number of columns in a result set

Syntax

```
SQLRETURN SQLNumResultCols (
       SQLHSTMT StatementHandle,
       SQLSMALLINT *ColumnCount)
```

Description

This function determines the number of columns that exist in a result data set.

Parameters

StatementHandle

[Input] A SQL statement handle, previously allocated by calling SQLAllocHandle.

ColumnCount

[Output] A pointer to a location in memory where SQLNumResultCols stores the number of columns found in the result data set associated with StatementHandle.

Return Values

SQL_ERROR

SQL_INVALID_HANDLE

SQL_SUCCESS

SQL_SUCCESS_WITH_INFO

Example

See the example for SQLBindCol.

SQLPrepare

Sends an SQL statement to a data source for compiling

Syntax

```
SQLRETURN SQLPrepare (
       SQLHSTMT StatementHandle,
       SQLCHAR *StatementText,
       SQLINTEGER TextLength)
```

Description

This function sends an SQL statement to a data source so it can be prepared (compiled) for execution.

Parameters

StatementHandle

[Input] A SQL statement handle, previously allocated by calling SQLAllocHandle.

StatementText

[Input] A pointer to a location in memory where the SQL statement to be prepared is stored.

TextLength

[Input] The length of the SQL statement in bytes stored in the StatementText parameter or SQL_NTS for a NULL terminated string.

Return Values

SQL_ERROR

SQL_INVALID_HANDLE

SQL_SUCCESS

SQL_SUCCESS_WITH_INFO

Example

See the examples for SQLExecute, SQLBindCol and SQLRowCount.

SQLRowCount

Gets row count in a table following an INSERT, UPDATE, or DELETE

Syntax

```
SQLRETURN SQLRowCount (
       SQLHSTMT StatementHandle,
       SQLINTEGER *RowCount)
```

Description

This function obtains a count of the number of rows in a table that were affected by an INSERT, UPDATE, or DELETE operation.

Parameters

StatementHandle

[Input] A SQL statement handle, previously allocated by calling SQLAllocHandle.

RowCount

[Output] A pointer to a location in memory where this function is to store a count of the actual number of rows in a table that were affected by an INSERT, UPDATE, or DELETE operation.

Return Values

SQL_ERROR

SQL_INVALID_HANDLE

SQL_SUCCESS

Example

```
// This demonstrates SQLRowCount
void RowCount(void)
{
       SQLHANDLE hDBC;
```

```
        SQLHANDLE hStmt;
        SQLRETURN rc;
        SQLINTEGER iRows;

        // Allocate the DBC handle
        rc = SQLAllocHandle(SQL_HANDLE_DBC, NULL, &hDBC);
        if (!SQL_SUCCEEDED(rc))
        {
                // ERROR !!
        }

        // Connect to the data source
        rc = SQLConnect(hDBC, "YourDataSource", SQL_NTS,
                                        "YourUser", SQL_NTS,
                                        "YourPassword", SQL_NTS);
        if (!SQL_SUCCEEDED(rc))
        {
                // ERROR !!
        }

        // Allocate the statement handle
        rc = SQLAllocHandle(SQL_HANDLE_STMT, hDBC, &hStmt);
        if (!SQL_SUCCEEDED(rc))
        {
                // ERROR !!
        }

        // Prepare a SQL insert statement, using the previously
        // allocated statement handle
        rc = SQLPrepare(hStmt,
                "INSERT INTO "
                "YOUR_TABLE(CHAR_FIELD, INT_FIELD) "
                "VALUES('ABC', 1)",
                SQL_NTS);
        if (!SQL_SUCCEEDED(rc))
        {
                // ERROR !!
        }

        // Execute the prepared SQL statement
        rc = SQLExecute(hStmt);
        if (!SQL_SUCCEEDED(rc))
        {
                // ERROR !!
        }

        // Retrieve the count of rows affected by the SQLExecute
        // command
```

```
        rc = SQLRowCount(hStmt, (!SQL_SUCCEEDED(rc))
        &iRows);
        if
        {
                // ERROR !!

        }

        // Free the statement handle
        rc = SQLFreeHandle(SQL_HANDLE_STMT, hStmt);
        if (!SQL_SUCCEEDED(rc))
        {
                // ERROR !!
        }

        // Disconnect from the data source
        rc = SQLDisconnect(hDBC);
        if (!SQL_SUCCEEDED(rc))
        {
                // ERROR !!
        }

        // Free the DBC handle
        rc = SQLFreeHandle(SQL_HANDLE_DBC, hDBC);
        if (!SQL_SUCCEEDED(rc))
        {
                // ERROR !!
        }
}
```

SQLSetConnectAttr

Sets RDM Embedded SQL attributes that govern aspects of connections

Syntax

```
SQLRETURN SQLSetConnectAttr(
        SQLHDBC ConnectionHandle,
        SQLINTEGER Attribute,
        SQLPOINTER ValuePtr,
        SQLINTEGER StringLength);
```

Parameters

ConnectionHandle

[Input] Connection handle.

Attribute

[Input] Specifies the connection attribute type being set. Possible values are described below.

Description

The SQLSetConnectAttr function is used to set the following modes:

SQL_AUTOCOMMIT

Autocommit mode effects locking and transaction behavior when no explicit transaction has been started. If autocommit is off, SQL will begin a transaction upon receiving the first select or update statement, will obtain and retain locks, and will keep the transaction active until an explicit transaction abort or commit. Locks obtained outside a transaction will be retained if an explicit transaction is started. If autocommit mode is on, a commit is performed following each statement execution.

A 32-bit integer indicating the transaction commit state. Possible values are:

SQL_AUTOCOMMIT_OFF	Specifies that the application issues commits. This is the default value. Note that it is different from ODBC-specific default.
SQL_AUTOCOMMIT_ON	Specifies that RDM Embedded automatically issues a transaction commit after each statement.

SQL_ATTR_RDM_INITDB

A 32-bit integer indicating the whether the database needs to be initialized on connection using SQLConnect. Possible values are:

SQL_INITDB_OFF	Specifies that the database should not be initialized. This is the default value.
SQL_INITDB_ON	Specifies that the database should be initialized.

SQL_DBMS_ACCESS_MODE

If the open mode is not specified, or if this function is not called, the default mode is "Shared". The other open mode options allow databases to be opened in One User ("o") or Exclusive ("x") modes.

A null-terminated string identifying the RDM Embedded database open mode. Note that this attribute must be set prior to an SQLConnect call. Possible values are:

"s"	Shared access mode. Multiple users can be accessing database at the same time.
"x"	Exclusive access mode. Only one user can access the database. All others will be locked out.
"o"	One-user-only mode. Only one user will be using specified database. The application will attempt to lock out all other users.

SQL_ATTR_RDM_COMM_TYPE

A null-terminated string specifying the lock manager communication type.

" Internal "	Use Internal Lock manager.
" TCP "	Use TCP Lock manager
" IP "	Use IP lock manager.

SQL_ATTR_RDM_LOCKMGR_NAME

A null-terminated string specifying the RDM Embedded lock manager name.

ValuePtr

[Input] Pointer to the value to be associated with Attribute. Depending on the value of Attribute, ValuePtr will be a 32-bit unsigned integer value, or will point to a null-terminated character string.

StringLength

[Input] The size of the buffer in bytes or SQL_NTS. If ValuePtr is an integer, StringLength is ignored.

SQL_ATTR_RDM_DBD

A binary DBD definition generated by sddlp.

ValuePtr

[Input] Pointer to the DBD definition structure generated by sddlp.

StringLength

[Input] The size in bytes of the DBD definition structure.

SQL_ATTR_RDM_MAX_BLOB_SIZE

The maximum blob size that can be read can be set in the configuration settings (rdm.ini) or through this attribute setting. The attribute setting takes precedence if both options are used. If neither option is used, the original value of 132 characters is used. The values specified are in characters, so if the column is a long wvarchar, the actual number of bytes used is sizeof(wchar_t) times the value specified.

ValuePtr

Maximum size of blob data

StringLength

Size of character

Return Codes

SQL_ERROR

SQL_INVALID_HANDLE

SQL_SUCCESS

SQL_SUCCESS_WITH_INFO

SQLSetCursorName

Closes a cursor

Syntax

```
SQLRETURN SQLSetCursorName (
       SQLHSTMT StatementHandle,
       SQLCHAR *CursorName,
       SQLSMALLINT BufferLength)
```

Description

This function closes a cursor that has been opened on a SQL statement handle.

Parameters

StatementHandle

[Input] A SQL statement handle, previously allocated by calling SQLAllocHandle.

CursorName

[Output] A pointer to a location in memory where the user-defined cursor name is stored.

BufferLength

[Input] The size of the buffer in bytes or SQL_NTS.

Return Values

SQL_ERROR

SQL_INVALID_HANDLE

SQL_SUCCESS

Example

```
// This demonstrates SQLSetCursorName and SQLGetCursorName

void SetAndGetCursorName(void)
{
       SQLHANDLE hDBC;
       SQLHANDLE hStmt;
       SQLRETURN rc;
       SQLINTEGER iRows;
       SQLWCHAR szNameBuffer[15];
       SQLSMALLINT sActualBytes;

       // Allocate the DBC handle
       rc = SQLAllocHandle(SQL_HANDLE_DBC, NULL, &hDBC);
       if (!SQL_SUCCEEDED(rc))
       {
              // ERROR !!
       }

       // Connect to the data source
       rc = SQLConnect(hDBC, "YourDataSource", SQL_NTS,
                       "YourUser", SQL_NTS,
                       "YourPassword", SQL_NTS);
       if (!SQL_SUCCEEDED(rc))
       {
              // ERROR !!
       }

       // Allocate the statement handle
       rc = SQLAllocHandle(SQL_HANDLE_STMT, hDBC, &hStmt);
       if (!SQL_SUCCEEDED(rc))
       {
              // ERROR !!
       }


       // Set the name of the cursor to something
       rc = SQLSetCursorName(hStmt, "INS_CURS", SQL_NTS);
       if (!SQL_SUCCEEDED(rc))
       {
              // ERROR !!
       }


       // Prepare a SQL statement, using the previously allocated
       // statement handle
```

```
        rc = SQLPrepare(hStmt, "SELECT * FROM YOUR_TABLE",
                        SQL_NTS);
        if (!SQL_SUCCEEDED(rc))
        {
                // ERROR !!
        }


        // Execute the prepared SQL statement
        rc = SQLExecute(hStmt);
        if
        (!SQL_SUCCEEDED(rc))
        {
                // ERROR !!
        }


        // Retrieve the count of rows affected by the SQLExevute
        // command
        rc = SQLRowCount(hStmt, &iRows);
        if (!SQL_SUCCEEDED(rc))
        {
                // ERROR !!
        }

        // Clear the buffer
        memset(szNameBuffer, 0, sizeof(szNameBuffer));

        // Get the name of the cursor, and the number of bytes
        // returned
        rc = SQLGetCursorName(hStmt, szNameBuffer,
                        sizeof(szNameBuffer),
                        &sActualBytes);
        if (!SQL_SUCCEEDED(rc))
        {
                // ERROR !!
        }

        // Free the statement handle
        rc = SQLFreeHandle(SQL_HANDLE_STMT, hStmt);
        if (!SQL_SUCCEEDED(rc))
        {
                // ERROR !!
        }

        // Disconnect from the data source
        rc = SQLDisconnect(hDBC);
        if (!SQL_SUCCEEDED(rc))
```

```
        {
                // ERROR !!
        }

        // Free the DBC handle
        rc = SQLFreeHandle(SQL_HANDLE_DBC, hDBC);
        if (!SQL_SUCCEEDED(rc))
        {
                // ERROR !!
        }
}
```

SQLSetStmtAttr

Sets RDM Embedded SQL attributes related to a statement

Syntax

```
SQLRETURN SQLSetStmtAttr(
       SQLHSTMT StatementHandle,
       SQLINTEGER Attribute,
       SQLPOINTER ValuePtr,
       SQLINTEGER StringLength);
```

Description

The SQLSetStmtAttr function is used to set the query timeout for the number of seconds to wait for a SQL statement to execute before returning to the application.

Parameters

StatementHandle

[Input] Statement handle.

Attribute

[Input] Specifies the statement option type. Possible values are described below.

SQL_ATTR_QUERY_TIMEOUT

A 32-bit integer value that specifies the number of seconds to wait for an SQL statement to execute before timeing out. The default timeout is 10 seconds. A value of zero indicates that no timeout is set.

ValuePtr

[Input] Pointer to the value to be associated with Attribute. Depending on the value of Attribute, ValuePtr will be a 32-bit unsigned integer value, or will point to a null-terminated character string.

StringLength

[Input] The size of the buffer in bytes or SQL_NTS. If ValuePtr is an integer,

StringLength is ignored.

Return Codes

SQL_ERROR

SQL_INVALID_HANDLE

SQL_SUCCESS

SQL_SUCCESS_WITH_INFO

5. SQL DDL Statement Reference

5.1 Overview of the RDM Embedded SQL DDL

The Database Definition Language (DDL) for RDM Embedded SQL is based on standard ANSI SQL. The syntax for each RDM Embedded SQL DDL statement is provided in this chapter for ease of reference. Details of how to use each statement can be found in Using RDM Embedded SQL DDL.

A summary of the RDM Embedded SQL DDL statements appears in the following table.

SQL DDL Statement	Description
create database	Create new database definition.
create constant	Create a name for an integer constant. Only used for specifying char column lengths.
create domain	Create a name for a new data type and attributes (length and nullability).
create data file	Create a data file
create blob file	Create a BLOB file (containing long varchar field data)
create index file	Create a index file
create vardata file	Create a vardata file (containing varchar field data)

The required order of statements in an RDM Embedded SQL DDL specification file is shown by the following syntax.

```
create_database
[create_constant | create_domain].
[create_file].
create_table .
```

5.2 Command-line Utilities

The following lists the command-line utilities available in RDM Embedded:

Utility	Description
sddlp	Compiles the DDL specification
lsql	Allows a user to interactively execute RDM Embedded SQL statements

5.2.1 RDM Embedded SQL DDL syntax summary

```
RDM Embedded SQL_DDL
       : create_database
                [create_constant | create_domain].
                [create_file].
                create_table
                [create_table | create_set].

create_database
       : create database dbname [pagesize = number]
       [inmemory [persistent | read | volatile]];

create_constant
       : create constant constname = number;

create_domain
       : create domain domname as data_type[(number)] [not null];

data_type
       : char | wchar | smallint | integer | real | float
       | date | time | timestamp | db_addr | domname

create_file
       : create_data_file | create_index_file | create_vardata_file |
          create_blobdata_file

create_data_file
       : create data file [fileid =] "filename" [pagesize = number]
       [initial = number] [next = number] [maxpgs = number]
       [pctincrease = number]
       [maxslots = slots]
       [inmemory [persistent | read | volatile]]

       containing tablename [, tablename]. ;

create_index_file
       : create index file [fileid =] "filename" [pagesize = number]
       [initial = number] [next = number] [maxpgs = number]
       [pctincrease = number]
       [inmemory [persistent | read | volatile]]

                containing {tablename(keyname) | keyame}
                        [, {tablename(keyname) | keyname} ]. ;
create_vardata_file
       : create vardata file [fileid =] "filename"
```

```
        [pagesize = number]
        [initial = number] [next = number] [maxpgs = number]
        [pctincrease = number]
        [inmemory [persistent | read | volatile]]
        containing {tablename(fldname) | fldname}
                        [, {tablename(fldname) | fldname} ]. ;

create_blobdata_file
        : create blobdata file [fileid =] "filename"
        [pagesize = number]
        [initial = number] [next = number] [maxpgs = number]
        [pctincrease = number]
        [inmemory [persistent | read | volatile]]
        containing {tablename(fldname) | fldname}
                        [, {tablename(fldname) | fldname} ]. ;

create_table
        : create [circular] table tabname(
                column_spec
                [, column_spec]
                …
        )
        [create_index]. ;

column_spec
        : colname data_type[({number | constname})] [not null] [
           [primary | unique] key]

create_index
        : create [unique] index indexname (colname [asc | desc]
           [, colname [asc | desc]].) ;

create_set
        : create set setname order {ascending | descending | first | last | next}
                from tabname to tabname [by colname [, colname].]
                        [or tabname [by colname [, colname].]]. ;
```

Notes:

1. Reserved words are in boldface text.
2. The italicized items are user-supplied values. All of the italicized items correspond to identifiers except **number**, which must be a positive integer value. An identifier is any sequence of letters, digits, or '_' beginning with a letter.
3. Optional items are enclosed in brackets ("[ ]").
4. Alternative choices are shown separated by "|". Alternatives for the item that must be selected are specified within braces ("{ }")

6. SQL DML Statement Reference

6.1 Overview of the RDM Embedded SQL DML

The SQL database manipulation language (DML) features provided in RDM Embedded SQL are a subset of the 1989 ANSI SQL standard. Because of the limited resources available on certain target platforms, we have minimized the SQL capabilities in RDM Embedded SQL to those that are most necessary for efficient data access of local database information.

The following is a list of the DML statements used in RDM Embedded SQL.

SQL DML Statement	Description
begin	Begins a transaction
commit	Commits changes made during a transaction
delete	Deletes a row or rows from a table
insert	Inserts rows into a table
rollback	Discards all changes made since a transaction start
select	Retrieves columns and rows
update	Updates rows in a table

6.2 RDM Embedded SQL DML syntax summary

```
RDM Embedded SQL_DML
        : select
        | transaction
        | insert
        | update
        | delete

select
        : select expression [, expression]
                from table_name[, table_name]
                [where conditional_expression]
                [group by [table_name.]colname
                        [, [table_name.]colname]
                        [having conditional_expression]]
                [order by {number | [table_name.]colname}
                        [, {number | [table_name.]colname}];

table_name
        : [dbname.]tabname
```

```
expression
       : arith_operand [ {+ | - | * | /} arith_operand]
       | string_operand ^ string_operand
string_operand
       : [tabname.]colname
       | "string"
       | min( expression ) | max( expression )
       | ( expression )arith_operand
       | value
       | calc_function
       | ( expression )
       | {+ | -} arith_operand
value
       : [+|-]n[.n][e[+|-]nn]
       | "string"
       | date 'nnnn-nn-nn'
       | time 'nn:nn:nn'
       | timestamp 'nnnn-nn-nn nn:nn:nn'
n
       : 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
calc_function
       : {sum | avg | max | min}( expression )
       | count ( {* | [tabname.]colname} )
conditional_expression
       : relational_expression [{and | or} relational_expression]
relational_expression
       : expression {= | <> | < | > | <= | >=} expression
       | expression [not] like "pattern"
       | expression [not] between expression and expression
       | expression [not] in (value [, value])
       | ( conditional_expression )
       | not relational_expression
transaction
       : begin [work];
       | commit [work];
       | rollback [work];
insert
       : insert into table_name[( colname [, colname])]
              values (value [, value]);
update
       : update table_name
              set colname = {expression | null}
              [, colname = {expression | null}]
              [where {current of cursorname | conditional_expression}];
delete
       : delete from table_name
              [where {current of cursorname | conditional_expression}];
```

Notes:

1. Reserved words are in boldface text.
2. The italicized items are user-supplied values. All of the italicized items correspond to identifiers except number, which must be a positive integer value. An identifier is any sequence of letters, digits, or '_' beginning with a letter.
3. Optional items are enclosed in brackets ("[ ]").
4. Alternative choices are shown separated by "|". Alternatives for the item that must be selected are specified within braces ("{ }")

Return Codes

7.1 About return codes

Return codes can be divided into the categories of status and error codes. Status codes have a value of zero or greater, and they represent non-error information. Error codes are negative numbers. Note that the names of the return codes partially describe the status or error that has occurred.

Note that for SQL errors, the code return is typically, SQL_ERROR. If you need more details, it is necessary to call SQLGetDiagRec and/or SQLGetDiagField to get possible reasons for the errors.

The two tables below list the status codes and error codes, respectively, in numerical order. In the following section, the codes are described in alphabetical order.

Status Number	Status Name	Status Description
0	SQL_SUCCESS	The function has executed successfully.
1	SQL_SUCCESS_WITH_INFO	The function has executed successfully, but a warning or additional information is available. To retrieve more detailed failure information, call the SQLGetDiagRec and/or SQLGetDiagField functions.
99	SQL_NEED_DATA	One or more parameters have not been set.
100	SQL_NO_DATA_FOUND	A data exception has occurred.
-1	SQL_ERROR	The application has generated an SQL error. To retrieve more detailed failure information, call the SQLGetDiagRec and/or SQLGetDiagField functions.
-2	SQL_INVALID_HANDLE	A null statement handle or connection handle was encountered. This return code results from a programming error only.