Open DataBase Connectivity (ODBC)

version 0.8

Typeset in LaTeX from SGML source using the DOCBUILDER 3.0 Document System.

Contents

Chapter 1

ODBC User's Guide

ODBC is written in Erlang and intended to be used from Erlang applications.
The *ODBC* application enables access to foreign SQL DBMSs from Erlang.

1.1 Introduction

Introduction

This book describes the *ODBC (Open DataBase Connectivity) Application*. It is an interface to various SQL databases to and it is a part of the Open Telecom Platform (OTP).

ODBC is written in Erlang and C and is intended to be used in conjunction with Erlang applications. To use the ODBC application it is necessary to install one or more ODBC Drivers written in C. ODBC is a 'pure ODBC 3.0 application', and it is defined in Reference 1. This implies that the application would not work with ODBC 2.x (or earlier) Drivers.

The ODBC application implements a subset of the Core level functionality as specified by the ODBC standard.

Before using the ODBC application it is necessary to install it properly. This includes compiling it with the proper include files, and dynamically linking to libraries of the ODBC Driver.

Prerequisites

Readers of this manual are assumed to be familiar with the Erlang programming language in general.

1.2 Installation

Installation

Before using the ODBC application it needs to be properly installed. The reason why it is not completely installed, with the rest of Erlang/OTP, is that it is necessary to have access to the include files, and dynamic libraries, of the ODBC software it will be working with.

To install the ODBC module you need to:

- Install your ODBC software package (ODBC Driver Manager and Driver).
- Edit the Makefile in the src directory of the odbc application. The Makefile may need to be adapted to use correct paths and compiler command and options.
- Run the Makefile.

Before you can use the ODBC application you also need to install a database.

1.3 Overview

Interfaces

The interface of the *ODBC* application is divided into three parts:

- Functions for starting and stopping ODBC servers.
- The *Basic API* which is almost identical to that defined by the ODBC standard.
- The*Utility API* which supplies a few easy-to-use functions for database access with little control over details.

An ODBC serving process can be compared to an Erlang port. It allows you to make function calls, from Erlang, to an ODBC Driver (or in reality a Driver Manager, which talks to a Driver). The Driver communicates with the database, which it is built for. The serving process also plays the role of a server and will be referred to as the server in this text.

When you start ODBC you get a new server, which handles requests by passing them on to an ODBC Driver. Requests are handled sequentially and the server cannot be connected to more than one database at any time.

> Note:
> An ODBC server can only be started on a named node with a cookie (start Erlang with one of the -name <nodename> or -sname <nodename> options and possibly with the -setcookie <cookie> option).

The server links to the process that starts it (the client). Should the client terminate, the server terminates too. The server is dedicated to the client process just like an Erlang port, but there are two important differences: an ODBC server accepts requests from any process (ports accept messages from the connected process only), and it does not send messages spontaneously (or deliver them from the ODBC Driver) – the ODBC server is passive.

Utility API

Using the Utility API is easy. Follow the steps below:

1. Start the server.
2. Initialise the ODBC environment by calling `init_env/[1, 2]`, where [1, 2] marks the possible arities of the function
3. Connect to the database with one of `connect/[3, 4, 5, 6]`
4. Submit SQL statements to the database by calling `exec_stmt/[3, 4]`.
5. Disconnect by calling `disconnect/[2, 3]`. At this point it is possible to connect to another database by calling `connect/[3, 4, 5, 6]` again, or you may wish to terminate the environment by calling `terminate_env/[2, 3]`.
6. The server process dies when `stop/[1, 2]` is called.

Basic API

To be able to make full use of the Basic API it is necessary to be familiar with the ODBC standard. Here is a typical way to use it for a SELECT statement though:

1. Allocate an environment handle: `sql_alloc_handle/[3, 4]` .
2. Set the ODBC version environment attribute: `sql_set_env_attr/[5, 6]`.
3. Allocate a database connection handle: `sql_alloc_handle/[3, 4]`.
4. Connect to the database: `sql_connect/[5, 6]` or `sql_driver_connect/[5, 6]`.
5. Set connection attributes (if you're not happy with the default values): `sql_set_connect_attr/[5, 6]`.
6. Allocate a statement handle: `sql_alloc_handle/[3, 4]`.
7. Execute an SQL statement: `sql_exec_direct/[3, 4]`
8. Check if the statement generated a result set (or table), which SELECT statements do: `sql_num_result_cols/[2, 3]` returns a value greater than zero.
9. Retrieve data about the buffer size needed for one of the returned columns of the table: `sql_describe_col/[4, 5]` and `display_size/2`.
10. Allocate a buffer for the column at hand: `alloc_buffer/[3, 4]`.
11. Bind the buffer to the column: `sql_bind_col/[4, 5]`.
12. Repeat steps 9 through 11 for each desired column (not necessarily all columns in the table).
13. Retrieve the first/next column into the buffers: `sql_fetch/[2, 3]`.
14. Read the buffer values: `read_buffer/[2, 3]`.
15. Repeat steps 13 through 14 until all rows in the table have been retrieved.
16. Close the cursor on the statement: `sql_close_cursor/[2, 3]`.
17. Start over with step 7 to execute a new statement or free the statement handle: `sql_free_handle/[3, 4]`.
18. Disconnect from the database: `sql_disconnect/[2, 3]` .
19. Start over with step 4 to connect to another database or free the database connection handle: `sql_free_handle/[3, 4]`.
20. Free the environment handle: `sql_free_handle/[3, 4]`.

Here is a typical way to use the Basic API for an INSERT statement:

- Follow steps 1 through 8 above.
- Check how many rows were affected by the statement: `sql_row_count/[2, 3]`.
- Commmit or rollback the statement: `sql_end_tran/[4, 5]`.
  Note:
  This is only necessary if the connection attribute SQL_ATTR_AUTOCOMMIT has been changed from the default value to SQL_AUTOCOMMIT_OFF. Otherwise the transaction is automatically committed.
- Follow steps 17 through 19 above.

You may use the Basic API and the Utility API intermittently, but remember that certain operations performed in the Basic API (like setting different attributes defined by the ODBC standard) may affect the behaviour of the Utility API.

Choice of API

When to use which API? The Utility API can be used when none of the following is true:

- It is necessary to set an attribute. This can only be done through the Basic API. You may however use the Basic API just for setting attributes, and the Utility API for all other tasks.
- The table resulting from a SELECT statement is very big. In this case using the Utility API would consume a huge amount of memory, since the whole table is returned in one chunk. You can get around this problem by using the Basic API by retrieving fewer rows at a time, or by using smaller buffers for large columns (which causes data to be truncated).
- You consider ODBC Driver warnings to be serious. The Utility API ignores warnings (when the ODBC Driver returns the code SQL_SUCCESS_WITH_INFO).

1.4 ODBC Examples

Example Introduction

In this and the following chapter two examples of the usage of the ODBC interface will be introduced.

The examples contain some basic operations, creating tables, writing and reading data, and dropping tables. They do the same things using the two different interface parts: the Utility API and the Basic API. The SQL used in the example is supported by Oracle8 and Intersolv's DataDirect ODBC Oracle8 Driver, but it may not work with other DBMSs/Drivers. The strings used to connect to the database depend on how your ODBC Driver is configured.

Utility API

A database has been created in the RDBMS at hand. We will now create a new table, insert data into it, select the data, and finally delete the table. The ODBC application has been started on a named node with some cookie.

```
%% ''The contents of this file are subject to the Erlang Public License,
%% Version 1.1, (the "License"); you may not use this file except in
%% compliance with the License. You should have received a copy of the
%% Erlang Public License along with this software. If not, it can be
%% retrieved via the world wide web at http://www.erlang.org/.
%%
%% Software distributed under the License is distributed on an "AS IS"
%% basis, WITHOUT WARRANTY OF ANY KIND, either express or implied. See
%% the License for the specific language governing rights and limitations
%% under the License.
%%
%% The Initial Developer of the Original Code is Ericsson Utvecklings AB.
%% Portions created by Ericsson are Copyright 1999, Ericsson Utvecklings
%% AB. All Rights Reserved.''
%%
%%     $Id$
%%
-module(utility).

-export([start/0]).


% This string depends on how your ODBC Driver is configured.
% You need to fill in your own value.
%-define(ConnectStr, "DSN=Oracle8;UID=myself;PWD=secret").


%% Note that the SQL syntax is database and ODBC Driver dependent.
%%
start() ->
    % Start a new ODBC server. The application must already be started.
```

```
    {ok, _Pid} = odbc:start_link({local, odbc1}, [], []),

    % Initialise the environment (also loads the Driver Manager).
    {ok, EnvHandle} = odbc:init_env(odbc1, infinity),

    % Load the Driver and connect to the database.
    {ok, ConnectionHandle} = odbc:connect(odbc1, EnvHandle, ?ConnectStr, infinity),

    % Create a new table.
    % By default, all transactions are automatically committed.
    CreateStmt = "CREATE TABLE TAB1 (ID number(3), DATA char(10))",
    {updated, NAffectedRows1} =
        odbc:execute_stmt(odbc1, ConnectionHandle, CreateStmt, infinity),
    ok = io:format("Create: Number of affected rows: ~p~n", [NAffectedRows1]),

    % Insert a row.
    InsertStmt = "INSERT INTO TAB1 VALUES (1, 'a1a2a3a4a5')",
    {updated, NAffectedRows2} =
        odbc:execute_stmt(odbc1, ConnectionHandle, InsertStmt, infinity),
    ok = io:format("Insert: Number of affected rows: ~p~n", [NAffectedRows2]),

    % Select all rows.
    SelectStmt = "SELECT * FROM TAB1",
    {selected, ColumnNames, Rows} =
        odbc:execute_stmt(odbc1, ConnectionHandle, SelectStmt, infinity),
    ok = io:format("Select: Column names: ~p, Rows: ~p~n", [ColumnNames, Rows]),

    % Delete the table.
    DropStmt = "DROP TABLE TAB1",
    {updated, NAffectedRows3} =
        odbc:execute_stmt(odbc1, ConnectionHandle, DropStmt, infinity),
    ok = io:format("Delete: Number of affected rows: ~p~n", [NAffectedRows3]),

    % Disconnect.
    ok = odbc:disconnect(odbc1, ConnectionHandle, infinity),

    % Terminate the environment.
    ok = odbc:terminate_env(odbc1, EnvHandle, infinity),

    % Stop the server.
    ok = odbc:stop(odbc1, infinity).
```

Basic API

A database has been created in the RDBMS at hand. We will now create a new table, insert data into it, select the data, and finally delete the table. The ODBC application has been started on a named node with some cookie. In this example we will not use the autocommit mode, meaning that we have to commit changes explicitly. The example does not include any error handling.

```
%% ''The contents of this file are subject to the Erlang Public License,
%% Version 1.1, (the "License"); you may not use this file except in
%% compliance with the License. You should have received a copy of the
```

```
%% Erlang Public License along with this software. If not, it can be
%% retrieved via the world wide web at http://www.erlang.org/.
%%
%% Software distributed under the License is distributed on an "AS IS"
%% basis, WITHOUT WARRANTY OF ANY KIND, either express or implied. See
%% the License for the specific language governing rights and limitations
%% under the License.
%%
%% The Initial Developer of the Original Code is Ericsson Utvecklings AB.
%% Portions created by Ericsson are Copyright 1999, Ericsson Utvecklings
%% AB. All Rights Reserved.''
%%
%%     $Id$
%%
-module(basic).

-export([start/0]).


% Contains macros defined by the ODBC standard.
-include("/clearcase/otp/libraries/odbc/include/odbc.hrl").


% These strings depend on how your ODBC Driver is configured.
% You need to fill in your own values.
%-define(DSN, "Oracle8").
%-define(UID, "myself").
%-define(PWD, "secret").


% The maximum length for column names + 1.
% The 1 is there to allow room for a null-termination character.
-define(BufLenColName, 64).

%% The maximum length of table data + 1.
-define(MaxDataBufLen, 1024).

%% Note that the SQL syntax is database and ODBC Driver dependent.
%% Error handling is not covered by the example.
%%
start() ->
    % Start a new ODBC server. The application must already be started.
    {ok, _Pid} = odbc:start_link({local, odbc1}, [], []),

    % Allocate an environment handle (also loads the Driver Manager).
    {?SQL_SUCCESS, EnvHandle} =
      odbc:sql_alloc_handle(odbc1, ?SQL_HANDLE_ENV, ?SQL_NULL_HANDLE, infinity),

    % Set the ODBC version attribute to tell the Driver we're a 3.0 application.
    ?SQL_SUCCESS =
      odbc:sql_set_env_attr(odbc1,
                            EnvHandle,
                            ?SQL_ATTR_ODBC_VERSION,
```

```
                          ?SQL_OV_ODBC3,
                          ?SQL_C_ULONG,
                          infinity),

  % Allocate a connection handle.
  {?SQL_SUCCESS, ConnectionHandle} =
    odbc:sql_alloc_handle(odbc1, ?SQL_HANDLE_DBC, EnvHandle, infinity),

  % Connect to the database (also loads the Driver).
  ?SQL_SUCCESS =
    odbc:sql_connect(odbc1, ConnectionHandle, ?DSN, ?UID, ?PWD, infinity),

  % Turn the autocommit mode off (if you don't want it).
  ?SQL_SUCCESS = odbc:sql_set_connect_attr(odbc1,
                                           ConnectionHandle,
                                           ?SQL_ATTR_AUTOCOMMIT,
                                           ?SQL_AUTOCOMMIT_OFF,
                                           ?SQL_C_ULONG,
                                           infinity),

  % Allocate a statement handle.
  {?SQL_SUCCESS, StmtHandle} =
    odbc:sql_alloc_handle(odbc1, ?SQL_HANDLE_STMT, ConnectionHandle,
                          infinity),

  % Create a new table.
  CreateStmt = "CREATE TABLE TAB1 (ID number(3), DATA char(10))",
  ?SQL_SUCCESS = odbc:sql_exec_direct(odbc1, StmtHandle, CreateStmt,
                                      infinity),

  % Print how many rows were affected by the statement.
  {?SQL_SUCCESS, NAffectedRows1} = odbc:sql_row_count(odbc1, StmtHandle,
                                                      infinity),
  ok = io:format("Create: Number of affected rows: ~p~n", [NAffectedRows1]),

  % Commit the transaction.
  ?SQL_SUCCESS =
    odbc:sql_end_tran(odbc1,
                      ?SQL_HANDLE_DBC,
                      ConnectionHandle,
                      ?SQL_COMMIT,
                      infinity),

  % Insert a new row.
  InsertStmt = "INSERT INTO TAB1 VALUES (1, 'a1a2a3a4a5')",
  ?SQL_SUCCESS = odbc:sql_exec_direct(odbc1, StmtHandle, InsertStmt,
                                      infinity),

  % Print how many rows were affected by the statement.
  {?SQL_SUCCESS, NAffectedRows2} =
    odbc:sql_row_count(odbc1, StmtHandle, infinity),
  ok = io:format("Insert: Number of affected rows: ~p~n", [NAffectedRows2]),
```

```
% Commit the transaction.
?SQL_SUCCESS =
  odbc:sql_end_tran(odbc1,
                    ?SQL_HANDLE_DBC,
                    ConnectionHandle,
                    ?SQL_COMMIT,
                    infinity),

% Select the DATA column from all rows.
SelectStmt = "SELECT DATA FROM TAB1",
?SQL_SUCCESS = odbc:sql_exec_direct(odbc1, StmtHandle, SelectStmt, infinity),

% Print how many columns there are in the table resulting from the
% statement.
{?SQL_SUCCESS, NSelectedCols} =
  odbc:sql_num_result_cols(odbc1, StmtHandle, infinity),
ok = io:format("Select: Number of columns: ~p~n", [NSelectedCols]),

% Describe the column(s) of the resulting table.
{?SQL_SUCCESS, {ColName, _LenColName}, SqlType, ColSize, _DecDigits,
                _Nullable} =
  odbc:sql_describe_col(odbc1, StmtHandle, 1, ?BufLenColName, infinity),

% Calculate the size of the buffer(s) we're going to use to retrieve data.
% Make sure you protect yourself from trying to allocate huge amounts of
% memory.
DispSize = odbc:display_size(SqlType, ColSize),
BufSz =
    if
        ColSize > ?MaxDataBufLen ->
            ?MaxDataBufLen;
        true ->
            DispSize
    end,

% Allocate data buffer(s).
{ok, Buf} =
  odbc:alloc_buffer(odbc1, ?SQL_C_CHAR, BufSz, infinity),

% Bind the buffer(s) to the column.
?SQL_SUCCESS = odbc:sql_bind_col(odbc1, StmtHandle, 1, Buf, infinity),

% Fetch the first row into the bound buffer(s) (only one buffer bound here).
?SQL_SUCCESS = odbc:sql_fetch(odbc1, StmtHandle, infinity),

% Read the value from the buffer(s).
{ok, {ColValue, _LenColValue}} =
  odbc:read_buffer(odbc1, Buf, infinity),
io:format("Select: Column name: ~p, Data: ~p~n", [ColName, ColValue]),

% Check that there are no more rows to fetch.
?SQL_NO_DATA = odbc:sql_fetch(odbc1, StmtHandle, infinity),
```

```
% Close the cursor on the statement.
?SQL_SUCCESS = odbc:sql_close_cursor(odbc1, StmtHandle, infinity),

% Deallocate the buffer(s).
ok = odbc:dealloc_buffer(odbc1, Buf, infinity),

% Delete the table.
DropStmt = "DROP TABLE TAB1",
?SQL_SUCCESS = odbc:sql_exec_direct(odbc1, StmtHandle, DropStmt, infinity),

% Print how many rows were affected by the statement.
{?SQL_SUCCESS, NAffectedRows3} = odbc:sql_row_count(odbc1, StmtHandle,
                                                    infinity),
ok = io:format("Delete: Number of affected rows: ~p~n", [NAffectedRows3]),

% Commit the transaction.
?SQL_SUCCESS =
  odbc:sql_end_tran(odbc1,
                    ?SQL_HANDLE_DBC,
                    ConnectionHandle,
                    ?SQL_COMMIT,
                    infinity),

% Free the statement handle.
?SQL_SUCCESS =
    odbc:sql_free_handle(odbc1, ?SQL_HANDLE_STMT, StmtHandle, infinity),

% Disconnect from the database.
?SQL_SUCCESS = odbc:sql_disconnect(odbc1, ConnectionHandle, infinity),

% Free the connection handle.
?SQL_SUCCESS =
  odbc:sql_free_handle(odbc1, ?SQL_HANDLE_DBC, ConnectionHandle, infinity),

% Free the environment handle.
?SQL_SUCCESS =
  odbc:sql_free_handle(odbc1, ?SQL_HANDLE_ENV, EnvHandle, infinity),

% Stop the server.
ok = odbc:stop(odbc1).
```

1.5 ODBC Release Notes

This document describes the changes made to the ODBC application. The intention of this document is to list all incompatibilities as well as all enhancements and bug-fixes for each and every release of ODBC. Each release of ODBC constitutes one section in this document. The title of each section is the version number of ODBC.

ODBC 0.8

ODBC is a application, which allows client applications to access SQL databases.

Improvements and new features

- Functions for starting and stopping ODBC servers.
- The *Basic API* which is almost identical to that defined by the ODBC standard.
- The *Utility API* which supplies a few easy-to-use functions for database access with little control over details.

Fixed Bugs and malfunctions

Version 0.8.2 is the source code of ODBC in the directory <OTP_INSTALLATIONPATH>/lib/odbc-<OdbcVersion>/src.

Incompatibilities

N.A.

Known bugs and problems

- The documentation is not ready.

Any comments regarding the ODBC application would be appreciated.

ODBC Reference Manual

Short Summaries

- Erlang Module odbc [page 19] – Open Data Base Connectivity

odbc

The following functions are exported:

- `start_link(Args, Options) ->`
  [page 19] Starts a new ODBC server process.
- `start_link(ServerName, Args, Options) -> Result`
  [page 19] Starts a new ODBC server process.
- `stop(Server) ->`
  [page 20]
- `stop(Server, Timeout) -> ok`
  [page 20]
- `init_env(Server) ->`
  [page 21]
- `init_env(Server, Timeout) -> {ok, RefEnvHandle} | {error, {Fcn, [Reason]}}`
  [page 21]
- `connect(Server, RefEnvHandle, ConnectStr) ->`
  [page 22]
- `connect(Server, RefEnvHandle, ConnectStr, Timeout) ->`
  [page 22]
- `connect(Server, RefEnvHandle, DSN, UID, PWD) ->`
  [page 22]
- `connect(Server, RefEnvHandle, DSN, UID, PWD, Timeout) -> {ok, RefConnHandle} | {error, {Fcn, [Reason]}}`
  [page 22]
- `execute_stmt(Server, RefConnHandle, Stmt) ->`
  [page 23]
- `execute_stmt(Server, RefConnHandle, Stmt, Timeout) -> {updated, NRows} | {selected, [ColName], [Row]} {error, {Fcn, [Reason]}}`
  [page 23]

- alloc_buffer(Server, BufCType, Size, Timeout) -> {ok, RefBuf}
[page 36]
- dealloc_buffer(Server, RefBuf) ->
[page 37]
- dealloc_buffer(Server, RefBuf, Timeout) -> ok
[page 37]
- read_buffer(Server, RefBuf) ->
[page 37]
- read_buffer(Server, RefBuf, Timeout) -> {ok, {Value, LenInd}}
[page 37]

odbc (Module)

The ODBC API is divided into three parts:

- Start and Stop
  Starts and stops the server process.
- Basic API
  Gives access to the IDL Interface functions, which are mapped on ODBC functions.
- Utility API
  Consists of functions that are easier to use than the Basic API. These functions are on a higher level, do more of the job, but allow less control to the application programmer.

All functions described are synchronous. The interface supports all ODBC defined SQL data types except binaries. They are all mapped on Erlang strings. The type `string()` is a `list()` of integers representing ASCII codes. The type `boolean()` is either the macro `?SQL_TRUE` or the macro `?SQL_FALSE`. The default Timeout for all functions is 5000 ms, unless otherwise stated.

Start and Stop

Exports

```
start_link(Args, Options) ->
start_link(ServerName, Args, Options) -> Result
```

Types:

- Args = [Arg]
- Arg = {buffer_size, integer()} | {max_len_data, integer()}| {max_len_err_msg, integer()} | {max_len_str, integer()}
  {`buffer_size, integer()`}: The initial size of the buffer through which communication with the C node is done. The value does not limit the amount of data that can pass in either direction of a function call, since the buffer will grow dynamically. The default is 32 kb. The minimum is 4 kb.

- {max_len_data, integer()}: The maximum length, including null-termination, of table data, returned from ODBC. This value must be chosen with the buffer size in mind. The default is 8 kb. The argument is used only by the Utility API. *NOTE: The data source or driver may have a lower limit for the maximum size of returned data. This limit is the value of the optional statement attribute SQL_ATTR_MAX_LENGTH (see [1]).*
- {max_len_err_msg, integer()}: The maximum length, including null-termination, of the message part of ODBC error messages. This value must be chosen with the buffer size in mind. The default is 1 kb. The argument is used only by the Utility API.
- {max_len_str, integer()}: The maximum length, including null-termination, of other strings passed from ODBC to the ODBC server (e.g. column names). The value does not limit the size of returned table values. It must be chosen with the buffer size in mind. The default is 1 kb. The argument is used only by the Utility API.
- Options = [Opt]
- Opt = {timeout, integer()} | {debug, [Dbg]}
  timeout: The time in ms allowed for initialisation (see gen_server). debug: Debug options.
- Dbg = trace | log | statistics | {log_to_file, FileName} | {install, {Func, FuncState}}
  See gen_server and sys.
- ServerName = {local, atom()} | {global, atom()}
  When supplied, causes the server to be registered locally or globally. If the server is started without a name it can only be called using the returned pid.
- Result = {ok, pid()} | {error, Reason}
  The pid of the server or an error tuple.
- Reason = {already_started, pid()} | timeout | {no_c_node, Info}
  The server was already started, a timeout has expired, or the C node could not be started (the program may not have been found or may not have been executable e.g.).
- Info = string()
  More information.

Starts a new ODBC server process, registers it with the supervisor, and links it to the calling process. Opens a unique IDL connection to a new C node on the local host, using the same cookie as is used by the node of the calling process. Links to the process on the C node.

> **Note:**
> There is no default timeout value. Not using the timeout option is equivalent to having an infinite timeout value.
> An expired timeout is reported as an error here, not an exception.
> The debug options are described in the sys module documentation.

```
stop(Server) ->
stop(Server, Timeout) -> ok
```

Types:

- Server = pid() | Name | {global, Name} | {Name, Node}
  The pid of the server process, a registered name, a globally registered name, or a registered name on a remote node.
- Timeout = integer() | infinity
  Max time (ms) for serving the request.

Stops the ODBC server process as soon as all already submitted requests have been processed. The C node is also stopped.

Utility API

The Utility API uses three maximum string length parameters: the maximum data string length (max_len_data), the maximum error message length (max_len_err_msg), and the maximum length of 'other strings' (e.g. column names) passed from ODBC (max_len_str). These can be set in the call to `start_link/[2, 3]`, but there are default values. Errors reported by the ODBC API are returned in lists. The relative order of these errors is the same as specified in [1]. Warnings are always ignored and execution proceeds. Should an error occur, execution stops.

Exports

```
init_env(Server) ->
init_env(Server, Timeout) -> {ok, RefEnvHandle} | {error, {Fcn, [Reason]}}
```

Types:

- Server = pid() | Name | {global, Name} | {Name, Node}
  The pid of the server process, a registered name, a globally registered name, or a registered name on a remote node.
- Timeout = integer() | infinity
  Max time (ms) for serving the request.
- RefEnvHandle = term()
  Reference to the initialised environment.
- Fcn = atom()
  The originating function.
- Reason = {SqlState, MoreInfo}
  An ODBC error tuple.
- SqlState = string()
  The SQL state, see [1].
- MoreInfo = {NativeCode, Msg, LenMsg}
  More error info.
- NativeCode = string()
  Data source specific error code.
- Msg = string()
  Error message.

- LenMsg = integer()
  Length of `Msg` before truncation.

Initialises the ODBC environment on the C node.

```
connect(Server, RefEnvHandle, ConnectStr) ->
connect(Server, RefEnvHandle, ConnectStr, Timeout) ->
connect(Server, RefEnvHandle, DSN, UID, PWD) ->
connect(Server, RefEnvHandle, DSN, UID, PWD, Timeout) -> {ok, RefConnHandle} |
        {error, {Fcn, [Reason]}}
```

Types:

- Server = pid() | Name | {global, Name} | {Name, Node}
  The pid of the server process, a registered name, a globally registered name, or a registered name on a remote node.
- RefEnvHandle = term()
  Reference to the environment. Returned by `init_env/[1,2]`.
- ConnectStr = string()
  Connection string. For syntax see SQLDriverConnect in [1].
- DSN = string()
  Name of the data source.
- UID = string()
  User ID.
- PWD = string()
  Password.
- Timeout = integer() | infinity
  Maximum time (ms) for serving the request.
- RefConnHandle = term()
  Reference to the opened connection.
- Fcn = atom()
  The originating function.
- Reason = {SqlState, MoreInfo}
  An ODBC error tuple.
- SqlState = string()
  The SQL state, see [1].
- MoreInfo = {NativeCode, Msg, LenMsg}
  More error info.
- NativeCode = string()
  Data source specific error code.
- Msg = string()
  Error message.
- LenMsg = integer()
  Length of `Msg` before truncation.

Opens a connection to a data source. There can be only one open data source connection per server. `connect/[3, 4]` is used when the information that can be supplied through `connect/[5, 6]` does not suffice.

> **Note:**
> The syntax to be used for `ConnectStr` is described under SQLDriverConnect in [1]. The `ConnectStr` must be complete.

```
execute_stmt(Server, RefConnHandle, Stmt) ->
execute_stmt(Server, RefConnHandle, Stmt, Timeout) -> {updated, NRows} | {selected,
             [ColName], [Row]} {error, {Fcn, [Reason]}}
```

Types:

- Server = pid() | Name | {global, Name} | {Name, Node}
  The pid of the server process, a registered name, a globally registered name, or a registered name on a remote node.
- RefConnHandle = term()
  Reference to an open connection. Returned by `connect/[3,4,5,6]`.
- Stmt = string()
  SQL statement to execute.
- Timeout = integer() | infinity
  Maximum time (ms) for serving the request.
- NRows = integer()
  The number of updated rows for UPDATE, INSERT, or DELETE statements, or -1 if the number is not available. For other statement types the value is driver defined, see [1].
- ColName = string()
  The name of a column in the resulting table.
- Row = [Value]
  One row of the resulting table.
- Value = string() | null
  One value in a row.
- Fcn = atom()
  The originating function.
- Reason = {SqlState, MoreInfo}
  An ODBC error tuple.
- SqlState = string()
  The SQL state, see [1].
- MoreInfo = {NativeCode, Msg, LenMsg}
  More error info.
- NativeCode = string()
  Data source specific error code.
- Msg = string()
  Error message.
- LenMsg = integer()
  Length of `Msg` before truncation.

Executes a single SQL statement. All changes to the data source are, by default, automatically committed if successful. Data that is returned for SELECT statements is in string form.

> **Note:**
> {updated, 0} or {updated, -1} is returned when a statement that does not select or update any rows is successfully executed.
>
> The ColNames are ordered the same way as the Values in the Rows (the first ColName is associated with the first Value of each Row etc.). The Rows have no defined order since they represent a set.
>
> Column names will be truncated if they are longer than the maximum string length (see option to start_link/[2, 3]). Table values will be truncated if they are longer than the maximum data length, or longer than the value of the statement attribute SQL_ATTR_MAX_LENGTH. If the amount of memory needed to retrieve a table value from a data source can not be determined, the default maximum data length (see start_link/[2, 3]) is used.

```
disconnect(Server, RefConnHandle) ->
disconnect(Server, RefConnHandle, Timeout) -> ok | {error, {Fcn, [Reason]}}
```

Types:

- Server = pid() | Name | {global, Name} | {Name, Node}
  The pid of the server process, a registered name, a globally registered name, or a registered name on a remote node.
- RefConnHandle = term()
  Reference to an open connection. Returned by connect/[3,4,5,6].
- Timeout = integer() | infinity
  Maximum time (ms) for serving the request.
- Fcn = atom()
  The originating function.
- Reason = {SqlState, MoreInfo}
  An ODBC error tuple.
- SqlState = string()
  The SQL state, see [1].
- MoreInfo = {NativeCode, Msg, LenMsg}
  More error info.
- NativeCode = string()
  Data source specific error code.
- Msg = string()
  Error message.
- LenMsg = integer()
  Length of Msg before truncation.

Closes the connection to a data source.

```
terminate_env(Server, RefEnvHandle) ->
terminate_env(Server, RefEnvHandle, Timeout) -> ok | {error, {Fcn, [Reason]}}
```

Types:

- Server = pid() | Name | {global, Name} | {Name, Node}
  The pid of the server process, a registered name, a globally registered name, or a registered name on a remote node.
- RefEnvHandle = term()
  Reference to the environment. Returned by `init_env/[1,2]`.
- Timeout = integer() | infinity
  Maximum time (ms) for serving the request.
- Fcn = atom()
  The originating function.
- Reason = {SqlState, MoreInfo}
  An ODBC error tuple.
- SqlState = string()
  The SQL state, see [1].
- MoreInfo = {NativeCode, Msg, LenMsg}
  More error info.
- NativeCode = string()
  Data source specific error code.
- Msg = string()
  Error message.
- LenMsg = integer()
  Length of `Msg` before truncation.

Cleans up the ODBC environment on the C node.

Basic API

To use the Basic API it is necessary to gain a comprehensive understandingof ODBC by studying [1]. ODBC defines the concept of deferred buffers. A deferred buffer is one that exists longer than one function call, so it can be used in several calls. Deferred buffers come in pairs: one data buffer and one length/indicator buffer. The length/indicator buffer is used for communicating the length of data in the data buffer, or to indicate something about the data (e.g. that it is a null-value). The Basic API handles these buffers accordingly: they are allocated, deallocated, read, and written pair-wise.

Exports

```
 sql_alloc_handle(Server, HandleType, RefInputHandle) ->
sql_alloc_handle(Server, HandleType, RefInputHandle, Timeout) -> {Result,
            RefOutputHandle}
```

Types:

- Server = pid() | Name | {global, Name} | {Name, Node}
  The pid of the server process, a registered name, a globally registered name, or a registered name on a remote node.

- HandleType = ?SQL_HANDLE_ENV | ?SQL_HANDLE_DBC | ?SQL_HANDLE_STMT
  Macros that determine which type of handle to allocate.
- RefInputHandle = term() | ?SQL_NULL_HANDLE
  The context in which the new handle is to be allocated. When allocating an environment handle, use `?SQL_NULL_HANDLE`. When allocating a connection handle the argument must be an environment handle and when allocating a statement handle it must be a connection handle.
- Timeout = integer() | infinity
  Maximum time (ms) for serving the request.
- Result = ?SQL_SUCCESS | ?SQL_SUCCESS_WITH_INFO | ?SQL_INVALID_HANDLE | ?SQL_ERROR
  Result macro.
- RefOutputHandle = term() | ?SQL_NULL_HENV | ?SQL_NULL_HDBC | ?SQL_NULL_HSTMT
  Reference to the allocated handle, or a value representing an error.

Allocates memory for an environment, connection, or statement handle. See SQLAllocHandle in [1].

Differences from the ODBC Function:

Allocation of descriptor handles is not supported. The parameters `Server` and `Timeout` have been added. The ODBC output parameter `OutputHandlePtr` has been changed into the returned value `RefOutputHandle`. Connection pooling is not supported.

```
sql_bind_col(Server, RefStmtHandle, ColNum, RefBuf) ->
sql_bind_col(Server, RefStmtHandle, ColNum, RefBuf, Timeout) -> Result
```

Types:

- Server = pid() | Name | {global, Name} | {Name, Node}
  The pid of the server process, a registered name, a globally registered name, or a registered name on a remote node.
- RefStmtHandle = term()
  Reference to the statement handle.
- ColNum = integer()
  Column number from left to right starting at 1.
- RefBuf = integer() | ?NULL_REF
  Reference to the buffer where the column data is placed (and to the associated length/indicator buffer). `?NULL_REF` removes the binding between a buffer and a column.
- Timeout = integer() | infinity
  Maximum time (ms) for serving the request.
- Result = ?SQL_SUCCESS | ?SQL_SUCCESS_WITH_INFO | ?SQL_INVALID_HANDLE | ?SQL_ERROR
  Result macro.

Assigns storage and data type for a column in a result set (binds a buffer to a column). See SQLBindCol in [1]. Buffers/columns can also be unbound.

> **Note:**
> The memory associated with `RefBuf` has to be allocated already.

Differences from the ODBC Function:
Neither binding of arrays nor the use of binding offsets is supported. It is not possible to unbind the data buffer without also unbinding the length/indicator buffer. The parameters `Server` and `Timeout` have been added. The input parameters `TargetType`, `TargetValuePtr`, `BufferLength`, and `StrLen_or_IndPtr` of the ODBC function have been replaced with the `RefBuf` parameter (which represents the same data).

```
sql_close_cursor(Server, RefStmtHandle) ->
sql_close_cursor(Server, RefStmtHandle, Timeout) -> Result
```

Types:

- Server = pid() | Name | {global, Name} | {Name, Node}
  The pid of the server process, a registered name, a globally registered name, or a registered name on a remote node.
- RefStmtHandle = term()
  Reference to the statement handle.
- Timeout = integer() | infinity
  Maximum time (ms) for serving the request.
- Result = ?SQL_SUCCESS | ?SQL_SUCCESS_WITH_INFO | ?SQL_INVALID_HANDLE | ?SQL_ERROR
  Result macro.

Closes a cursor that has been opened on a statement and discards pending results. See SQLCloseCursor in [1].

Differences from the ODBC Function:

The parameters `Server` and `Timeout` have been added.

```
sql_connect(Server, RefConnHandle, DSN, UID, Auth) ->
sql_connect(Server, RefConnHandle, DSN, UID, Auth, Timeout) -> Result
```

Types:

- Server = pid() | Name | {global, Name} | {Name, Node}
  The pid of the server process, a registered name, a globally registered name, or a registered name on a remote node.
- RefConnHandle = term()
  Reference to the connection handle.
- DSN = string()
  The name of the data source.
- UID = string()
  The user ID
- Auth = string()
  The user's password for the data source.
- Timeout = integer() | infinity
  Maximum time (ms) for serving the request.

- Result = ?SQL_SUCCESS | ?SQL_SUCCESS_WITH_INFO | ?SQL_INVALID_HANDLE | ?SQL_ERROR
  Result macro.

Establishes a connection to a driver and a data source. See SQLConnect in [1].
Differences from the ODBC Function:

Connection pooling is not supported. The parameters `Server` and `Timeout` have been added. The input parameters `NameLength1`, `NameLength2`, and `NameLength3` of the ODBC function have been excluded.

```
sql_describe_col(Server, RefStmtHandle, ColNum, BufLenColName) ->
sql_describe_col(Server, RefStmtHandle, ColNum, BufLenColName, Timeout) ->
              {Result,{ColName, LenColName}, SqlType, ColSize, DecDigs, Nullable}
```

Types:

- Server = pid() | Name | {global, Name} | {Name, Node}
  The pid of the server process, a registered name, a globally registered name, or a registered name on a remote node.
- RefStmtHandle = term()
  Reference to the statement handle.
- ColNum = integer()
  The column number from left to right, starting at 1.
- BufLenColName = integer()
  Length (>0) of the `ColName` buffer. Allow room for null-termination.
- Timeout = integer() | infinity
  Maximum time (ms) for serving the request.
- Result = ?SQL_SUCCESS | ?SQL_SUCCESS_WITH_INFO | ?SQL_INVALID_HANDLE | ?SQL_ERROR
  Result macro.
- ColName = string()
  The column name.
- LenColName = integer()
  The actual length of `ColName`. An ODBC SQL data type (ODBC supported data types are supplied through macros).
- SqlType = integer()
  An ODBC SQL data type (ODBC supported data types are supplied through macros) or a driver-specific type (not supplied through macros).
- ColSize = integer()
  The precision of the column (see appendix D in [1]). If the precision cannot be determined, 0 is returned.
- DecDigs = integer()
  The scale of the column (see appendix D in [1]). If the scale cannot be determined, or is not applicable, 0 is returned.
- Nullable = ?SQL_NO_NULLS | ?SQL_NULLABLE | ?SQL_NULLABLE_UNKNOWN
  Indicates whether the column allows null values or not.

Returns the result descriptor – column name, type, column size, decimal digits, and nullability – for one column in the result set. See SQLDescribeCol in [1]. To decide the buffer size (how many characters or bytes) needed to retrieve data for the column it is necessary to calculate the display size (see also appendix D in [1]). The function

`display_size(SqlType, ColSize) -> integer()` does the calculation. The input parameters are returned by `sql_describe_col/[4, 5]`.

Differences from the ODBC Function:

The function does not support retrieval of bookmark column data. The parameters `Server` and `Timeout` have been added. The output parameters `ColumnName`, `NameLengthPtr`, `DataTypePtr`, `ColumnSizePtr`, `DecimalDigitsPtr`, and `NullablePtr` of the ODBC function have been changed into the returned values `ColName`, `LenColName`, `SqlType`, `ColSize`, `DecDigs`, and `Nullable`. `BufLenColName` must be > 0.

```
sql_disconnect(Server, RefConnHandle) ->
sql_disconnect(Server, RefConnHandle, Timeout) -> Result
```

Types:

- Server = pid() | Name | {global, Name} | {Name, Node}
  The pid of the server process, a registered name, a globally registered name, or a registered name on a remote node.
- RefConnHandle = term()
  Reference to the connection handle.
- Timeout = integer() | infinity
  Maximum time (ms) for serving the request.
- Result = ?SQL_SUCCESS | ?SQL_SUCCESS_WITH_INFO | ?SQL_INVALID_HANDLE | ?SQL_ERROR
  Result macro.

Closes the connection associated with a specific connection handle. See SQLDisconnect in [1].

Differences from the ODBC Function:

Connection pooling is not supported. The parameters `Server` and `Timeout` have been added.

```
sql_driver_connect(Server, RefConnHandle, InConnStr, BufLenOutConnStr, DrvCompletion)
            ->
sql_driver_connect(Server, RefConnHandle, InConnStr, BufLenOutConnStr, DrvCompletion,
            Timeout) -> {Result, {OutConnStr, LenOutConnStr}}
```

Types:

- Server = pid() | Name | {global, Name} | {Name, Node}
  The pid of the server process, a registered name, a globally registered name, or a registered name on a remote node.
- RefConnHandle = term()
  Reference to the connection handle.
- InConnStr = string()
  A complete connection string (enough for connecting anyway).
- BufLenOutConnStr = integer()
  Length (>0) of the `OutConnStr` buffer. Allow room for null-termination.
- DrvCompletion = ?SQL_DRIVER_NOPROMPT
  No prompting with pop-ups.

- Timeout = integer() | infinity
  Maximum time (ms) for serving the request.
- Result = ?SQL_SUCCESS | ?SQL_SUCCESS_WITH_INFO | ?SQL_INVALID_HANDLE | ?SQL_ERROR | ?SQL_NO_DATA
  Result macro.
- OutConnStr = string()
  A complete connection string.
- LenOutConnStr = integer()
  The length of `OutConnStr` before truncation.

Establishes a connection to a driver and a data sourc, which needs more connection information than SQLConnect offers. See SQLDriverConnect in [1].

Differences from the ODBC Function:

The function does not support prompting with pop-ups, so the connection string supplied must be complete or, at least, complete enough for connecting. The parameters `Server` and `Timeout` have been added. The input parameters `WindowHandle` and `StringLength1` of the ODBC function have been excluded. The output parameters `OutConnectionString` and `StringLength2Ptr` have been changed into the returned values `OutConnStr` and `LenOutConnStr`. `BufLenOutConnStr` must be > 0.

```
sql_end_tran(Server, HandleType, RefHandle, ComplType) ->
sql_end_tran(Server, HandleType, RefHandle, ComplType, Timeout) -> Result
```

Types:

- Server = pid() | Name | {global, Name} | {Name, Node}
  The pid of the server process, a registered name, a globally registered name, or a registered name on a remote node.
- HandleType = ?SQL_HANDLE_ENV | ?SQL_HANDLE_DBC
  The type of handle for which to perform the transaction (all connections associated with an environment or a specific connection).
- RefHandle = term()
  Reference to the handle.
- ComplType = ?SQL_COMMIT | ?SQL_ROLLBACK
  Commit operation or rollback operation.
- Timeout = integer() | infinity
  Maximum time (ms) for serving the request.
- Result = ?SQL_SUCCESS | ?SQL_SUCCESS_WITH_INFO | ?SQL_INVALID_HANDLE | ?SQL_ERROR
  Result macro.
-

Requests a commit or rollback operation for all active operations on all statement handles associated with a connection. It can also request that a commit or rollback operation be performed for all connections associated with the environment handle. See SQLEndTran in [1].

> **Note:**
> Rollback of transactions may be unsupported by core level drivers.

Differences from the ODBC Function:

The parameters `Server` and `Timeout` have been added.

```
sql_exec_direct(Server, RefStmtHandle, Stmt) ->
sql_exec_direct(Server, RefStmtHandle, Stmt, Timeout) -> Result
```

Types:

- Server = pid() | Name | {global, Name} | {Name, Node}
  The pid of the server process, a registered name, a globally registered name, or a registered name on a remote node.
- RefStmtHandle = term()
  Reference to the statement handle.
- Stmt = string()
  An SQL statement.
- Timeout = integer() | infinity
  Maximum time (ms) for serving the request.
- Result = ?SQL_SUCCESS | ?SQL_SUCCESS_WITH_INFO | ?SQL_INVALID_HANDLE | ?SQL_ERROR | ?SQL_NEED_DATA | ?SQL_NO_DATA
  Result macro.

Executes a statement. See SQLExecDirect in [1].

Differences from the ODBC Function:

`?SQL_NO_DATA` is returned only in connection with positioned updates, which are not supported. The parameters `Server` and `Timeout` have been added. The input parameter `TextLength` of the ODBC function has been excluded.

```
sql_fetch(Server, RefStmtHandle) ->
sql_fetch(Server, RefStmtHandle, Timeout) -> Result
```

Types:

- Server = pid() | Name | {global, Name} | {Name, Node}
  The pid of the server process, a registered name, a globally registered name, or a registered name on a remote node.
- RefStmtHandle = term()
  Reference to the statement handle.
- Timeout = integer() | infinity
  Maximum time (ms) for serving the request.
- Result = ?SQL_SUCCESS | ?SQL_SUCCESS_WITH_INFO | ?SQL_INVALID_HANDLE | ?SQL_ERROR | ?SQL_NO_DATA
  Result macro.

Fetches a row of data from a result set. The driver returns data for all columns that were bound to storage locations with `sql_bind_col/[4, 5]`. See SQLFetch in [1].

Differences from the ODBC Function:

The parameters `Server` and `Timeout` have been added.

```
sql_free_handle(Server, HandleType, RefHandle) ->
sql_free_handle(Server, HandleType, RefHandle, Timeout) -> Result
```

Types:

- Server = pid() | Name | {global, Name} | {Name, Node}
  The pid of the server process, a registered name, a globally registered name, or a registered name on a remote node.
- HandleType = ?SQL_HANDLE_ENV | ?SQL_HANDLE_DBC | ?SQL_HANDLE_STMT
  Macros which define the type of handle to free.
- RefHandle = term()
  Reference to the handle.
- Timeout = integer() | infinity
  Maximum time (ms) for serving the request.
- Result = ?SQL_SUCCESS | ?SQL_INVALID_HANDLE | ?SQL_ERROR
  Result macro.

Releases a handle and frees all resources associated with it. See SQLFreeHandle in [1].

Differences from the ODBC Function:

The function does not support deallocation of descriptor handles. The parameters `Server` and `Timeout` have been added.

```
sql_get_connect_attr(Server, RefConnHandle, Attr, BufType) ->
sql_get_connect_attr(Server, RefConnHandle, Attr, BufType, Timeout) -> {Result, Value}
```

Types:

- Server = pid() | Name | {global, Name} | {Name, Node}
  The pid of the server process, a registered name, a globally registered name, or a registered name on a remote node.
- RefConnHandle = term()
  Reference to the connection handle.
- Attr = integer()
  One of the attributes described below or a driver-specific attribute.
- BufType = {?SQL_C_CHAR, BufLen} | ?SQL_C_ULONG | {?SQL_C_ULONG, IntType}
  The buffer type used for retrieving the data. For character type data also the buffer size. For integer type data that is driver-specific, also a subtype.
- BufLen = integer()
  Buffer size (>0) for character type data. Allow room for null-termination
- IntType = ?SQL_IS_UINTEGER | ?SQL_IS_INTEGER
  Used only for driver-specific attributes. See SQLGetConnectAttr in [1].
- Timeout = integer() | infinity
  Maximum time (ms) for serving the request.
- Result = ?SQL_SUCCESS | ?SQL_SUCCESS_WITH_INFO | ?SQL_INVALID_HANDLE | ?SQL_ERROR | ?SQL_NO_DATA
  Result macro.
- Value = {CharValue, LenCharValue} | NumValue
  Attribute data.

- CharValue = string()
  The value of the attribute when of character type.
- LenCharValue = integer()
  The length of CharValue before truncation.
- NumValue = integer()
  The value of the attribute when of numeric type.

Returns the current setting of a connection attribute. See SQLGetConnectAttr in [1].

Differences from the ODBC Function:

Only the following attributes, and their possible values, are supported (through macros). More information can be found under SQLSetConnectAttr in [1]. Driver-specific attributes are not supported through macros, but can be retrieved, if they are of character or signed/unsigned long integer types.

- ?SQL_ATTR_ACCESS_MODE
- ?SQL_ATTR_AUTOCOMMIT
- ?SQL_ATTR_ODBC_CURSORS
- ?SQL_ATTR_TRACE
- ?SQL_ATTR_TRACEFILE
- ?SQL_ATTR_TRANSLATE_LIB
- ?SQL_ATTR_TRANSLATE_OPTION

According to [1], `BufLen` (BufferLength) can be set to `?SQL_NTS`. This is probably not correct, since it would make it impossible for the driver to detect that data needs to be truncated. Hence, the `?SQL_NTS` value has been disallowed. The function takes a `BufType` parameter to distinguish between character type attributes and numeric type attributes. For character data the maximum string length must be supplied (allow room for null-termination). For driver-specific numeric type attributes, a subtype must be supplied. The returned value is either a tuple containing the attribute string and its length, or an integer, depending on the specified buffer type. The parameters `Server` and `Timeout` have been added. The output parameters `ValuePtr` and `StringLengthPtr` of the ODBC function have been changed into the returned values `CharValue` and `LenCharValue` for character type attributes and `NumValue` for integer types. The input parameter `BufferLength` has been included in the `BufType` parameter. `BufLen` must be > 0.

```
sql_get_diag_rec(Server, HandleType, RefHandle, RecNum, BufLenErrMsg) ->
sql_get_diag_rec(Server, HandleType, RefHandle, RecNum, BufLenErrMsg, Timeout) ->
                {Result, SqlState, NativeErr, {ErrMsg, LenErrMsg}}
```

Types:

- Server = pid() | Name | {global, Name} | {Name, Node}
  The pid of the server process, a registered name, a globally registered name, or a registered name on a remote node.
- HandleType = ?SQL_HANDLE_ENV | ?SQL_HANDLE_DBC | ?SQL_HANDLE_STMT
  The type of handle for which to retrieve information.
- RefHandle = term()
  Reference to the handle.

- RecNum = integer()
  Indicates the status record from which to retrieve information (> 0).
- BufLenErrMsg = integer()
  Length of the ErrMsg buffer (>0). Allow room for null-termination.
- Timeout = integer() | infinity
  Maximum time (ms) for serving the request.
- Result = ?SQL_SUCCESS | ?SQL_SUCCESS_WITH_INFO | ?SQL_INVALID_HANDLE | ?SQL_ERROR | ?SQL_NO_DATA
  Result macro.
- SqlState = string()
  The SQL state pertaining to the diagnostic record.
- NativeErr = integer()
  Data-source specific error code.
- ErrMsg = string(
  Error message.
- LenErrMsg = integer()
  The length of ErrMsg before truncation.

Retrieves the current values of multiple fields of a diagnostic record that contains error, warning, and status information. See SQLGetDiagRec in [1].

Differences from the ODBC Function:

Retrieving information associated with descriptor handles is not supported. The parameters Server and Timeout have been added. The output parameters SqlState, NativeErrorPtr, MessageText, and TextLengthPtr of the ODBC function have been changed into the returned values SqlState, NativeErr, ErrMsg, and LenErrMsg. BufLenErrMsg must be > 0.

```
sql_num_result_cols(Server, RefStmtHandle) ->
sql_num_result_cols(Server, RefStmtHandle, Timeout) -> {Result, ColCount}
```

Types:

- Server = pid() | Name | {global, Name} | {Name, Node}
  The pid of the server process, a registered name, a globally registered name, or a registered name on a remote node.
- RefStmtHandle = term()
  Reference to the statement handle.
- Timeout = integer() | infinity
  Maximum time (ms) for serving the request.
- Result = ?SQL_SUCCESS | ?SQL_SUCCESS_WITH_INFO | ?SQL_INVALID_HANDLE | ?SQL_ERROR
  Result macro.
- ColCount = integer()
  The number of columns in the result set.

Returns the number of columns in a result set. See SQLNumResultCols in [1].

Differences from the ODBC Function:

The parameters Server and Timeout have been added. The output parameter ColumnCountPtr of the ODBC function has been changed into the returned value ColCount.

```
sql_row_count(Server, RefStmtHandle) ->
sql_row_count(Server, RefStmtHandle, Timeout) -> {Result, RowCount}
```

Types:

- Server = pid() | Name | {global, Name} | {Name, Node}
  The pid of the server process, a registered name, a globally registered name, or a registered name on a remote node.
- RefStmtHandle = term()
  Reference to the statement handle.
- Timeout = integer() | infinity
  Maximum time (ms) for serving the request.
- Result = ?SQL_SUCCESS | ?SQL_SUCCESS_WITH_INFO | ?SQL_INVALID_HANDLE | ?SQL_ERROR
  Result macro.
- RowCount = integer()
  The number of affected rows. If the number of affected rows is not available -1 is returned. For exceptions, see SQLRowCount in [1].

Returns the number of rows affected by an UPDATE, INSERT, or DELETE statement. See SQLRowCount in [1].

Differences from the ODBC Function:

The parameters `Server` and `Timeout` have been added. The output parameter `RowCountPtr` of the ODBC function has been changed into the returned value `RowCount`.

```
sql_set_connect_attr(Server, RefConnHandle, Attr, Value, BufType) ->
sql_set_connect_attr(Server, RefConnHandle, Attr, Value, BufType, Timeout) -> Result
```

Types:

- Server = pid() | Name | {global, Name} | {Name, Node}
  The pid of the server process, a registered name, a globally registered name, or a registered name on a remote node.
- RefConnHandle = term()
  Reference to the connection handle.
- Attr = integer()
  One of the attributes described under `sql_get_connect_attr/[4, 5]` or a driver-specific attribute. The attributes defined by ODBC are supplied through macros, but driver-specific attributes are not.
- Value = string() | integer()
  The new attribute value.
- BufType = ?SQL_C_CHAR | ?SQL_C_ULONG | {?SQL_C_ULONG, IntType}
  The buffer type. Either a (null-terminated) string, an ODBC defined attribute of integer type, or a driver-specific attribute of integer type (which also has a subtype).
- IntType = ?SQL_IS_UINTEGER | ?SQL_IS_INTEGER
  Subtype for driver-specific integer attributes.
- Timeout = integer() | infinity
  Maximum time (ms) for serving the request.
- Result = ?SQL_SUCCESS | ?SQL_SUCCESS_WITH_INFO | ?SQL_INVALID_HANDLE | ?SQL_ERROR
  Result macro.

Sets attributes that govern aspects of connections. See SQLSetConnectAttr in [1]. The supported attributes are listed under `sql_get_connect_attr/[4, 5]`. Driver-specific attributes are not supported through macros, but can be set if they are strings or signed/unsigned long integers.

Differences from the ODBC Function:

Only character and signed/unsigned long integer attribute types are supported. The parameters `Server` and `Timeout` have been added. The input parameter `StringLength` of the ODBC function has been replaced with the input parameter `BufType`.

```
sql_set_env_attr(Server, RefEnvHandle, Attr, Value, BufType) ->
sql_set_env_attr(Server, RefEnvHandle, Attr, Value, BufType, Timeout) -> Result
```

Types:

- Server = pid() | Name | {global, Name} | {Name, Node}
  The pid of the server process, a registered name, a globally registered name, or a registered name on a remote node.
- RefEnvHandle = term()
  Reference to the environment handle.
- Attr = integer()
  One of the supported attributes described below.
- Value = string() | intiger()
  The new attribute value.
- BufType = ?SQL_C_CHAR | ?SQL_C_ULONG
  The buffer type. Either a (null-terminated) string or an ODBC defined attribute of integer type.
- Timeout = integer() | infinity
  Max time (ms) for serving the request.
- Result = ?SQL_SUCCESS | ?SQL_SUCCESS_WITH_INFO | ?SQL_INVALID_HANDLE | ?SQL_ERROR
  Result macro.

Sets attributes that govern aspects of environments. The following attributes, and their possible values, are supported (through macros). More information can be found under SQLSetEnvAttr in [1]. Other data types than character or unsigned long integer are not supported.

- ?SQL_ATTR_ODBC_VERSION

Differences from the ODBC Function:

Only character and unsigned long integer attribute types are supported. The parameters `Server` and `Timeout` have been added. The input parameter `StringLength` of the ODBC function has been replaced with the input parameter `BufType`.

```
alloc_buffer(Server, BufCType, Size) ->
alloc_buffer(Server, BufCType, Size, Timeout) -> {ok, RefBuf}
```

Types:

- Server = pid() | Name | {global, Name} | {Name, Node}
  The pid of the server process, a registered name, a globally registered name, or a registered name on a remote node.
- BufCType = ?SQL_C_CHAR | ?SQL_C_BINARY
  The C data type of the buffer.
- Size = integer()
  The buffer size (>0). For character data, allow room for null-termination.
- Timeout = integer() | infinity
  Maximum time (ms) for serving the request.
- RefBuf = term()
  A handle to the buffer.

Allocates a deferred data buffer and an associated length/indicator buffer.

```
dealloc_buffer(Server, RefBuf) ->
dealloc_buffer(Server, RefBuf, Timeout) -> ok
```

Types:

- Server = pid() | Name | {global, Name} | {Name, Node}
  The pid of the server process, a registered name, a globally registered name, or a registered name on a remote node.
- RefBuf = term()
  A handle to the buffer.
- Timeout = integer() | infinity
  Maximum time (ms) for serving the request.

Deallocates a deferred data buffer and the associated length/indicator buffer.

```
read_buffer(Server, RefBuf) ->
read_buffer(Server, RefBuf, Timeout) -> {ok, {Value, LenInd}}
```

Types:

- Server = pid() | Name | {global, Name} | {Name, Node}
  The pid of the server process, a registered name, a globally registered name, or a registered name on a remote node.
- RefBuf = term()
  A handle to the buffer.
- Timeout = integer() | infinity
  Maximum time (ms) for serving the request.
- Value = string()
  Contents of the buffer associated with `RefBuf`.
- LenInd = integer() | ?SQL_NULL_DATA | ?SQL_NO_TOTAL
  Length/indicator value associated with `RefBuf`.

Returns the contents of a deferred data buffer and its associated length/indicator buffer. Used in connection with `sql_fetch/[2, 3]`.

Error Messages and Exceptions

Errors caused by inability to contact the C node, allocate memory, or otherwise call ODBC functions cause exceptions. Exceptions are common to all functions. Errors caused by ODBC not being able to execute calls are reported through returned errors. These exceptions terminate the client only.

- {'EXIT', {badarg, M, F, A, ArgNo, Info}}
  The argument is of wrong type or out of range.
- {'EXIT', {internal_error, Info}}
  Internal error.
- {'EXIT', GenServerSpecificInfo}
  Error detected by `gen_server`.

These cause the ODBC server, and the C node, to terminate as well:

- {'EXIT', {timeout, Info}}
  Timeout expired.
- {'EXIT', {stopped, Reason}}
  The ODBC server died.

References

[1]: Microsoft ODBC 3.0, Programmer's Reference and SDK Guide

Index

Modules are typed in *this way*.
Functions are typed in `this way`.