Panther

Database Driver–Oracle

Release 4.25

Prolifics®

May 2000

1

Database Driver for ORACLE

This chapter provides documentation specific to ORACLE. It discusses the following:

This document is designed as a supplement to information found in the *Developer's Guide*.

Initializing the Database Engine

Database engine initialization occurs in the source file `dbiinit.c`. This source file is unique for each database engine and is constructed from the settings in the `makevars` file. In Prolifics for ORACLE, this results in the following `vendor_list` structure in `dbiinit.c`:

```
static vendor_t vendor_list[] =
{
   {"oracle", dm_orasup, DM_DEFAULT_CASE ,(char *) 0},

   { (char *) 0, (int (*)()) 0, (int) 0, (char *) 0 }
};
```

The settings are as follows:

`oracle`	Engine name. May be changed.
`dm_orasup`	Support routine name. Do not change.
`DM_DEFAULT_CASE`	Case setting for matching `SELECT` columns with Prolifics variable names. May be changed.

For Prolifics for ORACLE, the settings can be changed by editing the `makevars.ora` file.

Engine Name

You can change the engine name associated with the support routine `dm_orasup`. The application then uses that name in `DBMS ENGINE` statements and in `WITH ENGINE` clauses. For example, if you wish to use “tracking” as the engine name, change the following parameter in the `makevars.ora` file:

```
ORA_ENGNAME=tracking
```

If the application is accessing multiple engines, it makes ORACLE the default engine by executing:

```
DBMS ENGINE oracle-engine-name
```

where *oracle-engine-name* is the string used in `vendor_list`. For example,

```
DBMS ENGINE oracle
```

or

```
DBMS ENGINE tracking
```

Support Routine Name

`dm_sup` is the name of the support routine for ORACLE. This name should not be changed.

Case Flag

The case flag, `DM_DEFAULT_CASE`, determines how Prolifics's database drivers use case when searching for Prolifics variables for holding `SELECT` results. This setting is used when comparing ORACLE column names to either a Prolifics variable name or to a column name in a `DBMS ALIAS` statement.

ORACLE is case insensitive. Regardless of the case in a SQL statement, ORACLE creates all database objects—tables, views, columns, etc.—with upper case names. For ORACLE, the `DM_DEFAULT_CASE` setting is treated as `DM_FORCE_TO_LOWER_CASE`. This means that Prolifics attempts to match ORACLE column names to lower case Prolifics variables when processing `SELECT` results. If your application is using this default, use lower case names when creating Prolifics variables.

The case setting can be changed. If you wish to use upper case Prolifics variable names, use the `u` option in the `makevars` file for the `DM_FORCE_TO_UPPER_CASE` flag.

```
ORA_INIT=u
```

If you edit `makevars.ora`, you must remake your Prolifics executables. For more information on engine initialization, refer to Chapter 7 in the *Developer's Guide*.

Connecting to the Database Engine

ORACLE allows your application to use one or more connections. The application can declare any number of named connections with `DBMS DECLARE CONNECTION` statements, up to the maximum number permitted by the server.

The following options are supported for connections to ORACLE:

Table 1. Database connection options.

Option	Argument
`USER`	*user-name*
`PASSWORD`	*password*
`DEFERRED_PARSING`	`ON | OFF`

`USER` and `PASSWORD` have different configurations for SQL*Net V1 and SQL*Net V2.

For SQL*Net V1, a Prolifics application connects to the default ORACLE database unless the program supplies an ORACLE connect string or an ORACLE connect alias. This connect string or alias is appended to the *user-name* argument. For example:

```
# Connect string for TCP/IP
DBMS DECLARE c CONNECTION FOR USER "scott@T::nysales::P" \
   PASSWORD "tiger"

# Connect alias
DBMS DECLARE c CONNECTION FOR USER "scott@ny" \
   PASSWORD "tiger"
```

In the connect string example, the *network-prefix* is `T` for TCP/IP, the *host-name* is `nysales`, and the *system-ID* is `P`. In connect strings, use two colons between the parameters, instead of one, to prevent Prolifics from performing colon expansion on the names.

Even though you can specify a connect string as part of your *user-name* or *password*, better error messages are returned from ORACLE if it is part of the *user-name*.

For SQL*Net 2, the *user-name* argument contains the logon name and the service name or connect descriptor found in your `TNSNAMES.ORA` file.

```
# Service name for SQL*Net V2
DBMS DECLARE c CONNECTION FOR USER "scott@listener" \
   PASSWORD "tiger"
```

Refer to your SQL*Net documentation for more information on connect strings and connect descriptors.

Additional keywords are available for other database engines. If those keywords are included in your `DBMS DECLARE CONNECTION` command for ORACLE, it is treated as an error.

Connecting to the XA Library

In ORACLE 7, distributed transaction processing (DTP) can be handled by a transaction manager using ORACLE as one of its resource managers. ORACLE's XA library provides an interface to this environment.

Prolifics for ORACLE provides a special logon syntax for programs operating as application servers in an X/Open distributed processing environment. These logon options indicate that Prolifics should use ORACLE's XA library to set connection information.

In order to access the XA library, you must specify the following options in the `DBMS DECLARE CONNECTION` statement:

Option	Argument
`XA_CONN`	`ON │ OFF`
`XA_DBNAME`	*character_string*

`XA_CONN ON` tells Prolifics to use the ORACLE XA library. `XA_DBNAME` should be used when connecting to an open string with the DB field set.

For example, the following string does not set the DB field:

```
Oracle_XA+Acc=P/scott/tiger+SesTm=30
```

To connect using this open string:

```
DBMS [WITH ENGINE engine] DECLARE connection CONNECTION \
    FOR XA_CONN
```

For example, the following string sets DB to `resources`:

```
Oracle_XA+DB=resources+Acc=P/scott/tiger+SesTm=30
```

To connect using this open string:

```
DBMS [WITH ENGINE engine] DECLARE connection CONNECTION \
    FOR XA_CONN ON XA_DBNAME "resources"
```

or

```
DBMS [WITH ENGINE engine] DECLARE connection CONNECTION \
    FOR XA_CONN ON XA_DBNAME "RESOURCES"
```

Importing Database Tables

The Import⇒Database Objects option in the screen editor creates Prolifics repository entries based on database tables in an ORACLE database. When the import process is complete, each selected database table has a corresponding repository entry screen.

In Prolifics for ORACLE, the following database objects can be imported as repository entries:

- database tables
- database views
- synonyms

After the import process is complete, the repository entry screen contains:

- A widget for each column in the table, using the column's characteristics to assign the appropriate widget properties.
- A label for each column based on the column name.
- A table view named for the database table, database table view, or synonym.
- Links that describe the relationship between table views.

Each import session allows you to display and select up to 1000 database tables. Each database table can have up to 255 columns. If your database contains more than 1000 tables, use the filter to control which database tables are displayed.

Table Views

A table view is a group of associated widgets on an application screen. As a general rule, the members of a table view are derived from the same database table. When a database table is first imported to a Prolifics repository, the new repository screen has one table view that is named after the database table. All the widgets corresponding to the database columns are members of that table view.

The import process inserts values in the following table view properties:

- Name — The name of the table view, generally the same as the database table.
- Table — The name of the database table.
- Primary Keys — The columns that are defined as primary keys for the database table.
- Columns — A list of the columns in the database table is displayed when you click on the More button. However, this list is for reference only. It cannot be edited.

- Updatable — A setting that determines if the data in the table can be modified. The default setting for Updatable is Yes.

For each repository entry based on a database view, the primary key widgets must be available if you want to update data in that view. To do this, check that the Prolifics table view's Primary Keys property is set to the correct value. Then, the widgets corresponding to the primary keys must be members of either the Prolifics table view or one of its parent table views. For repository entries based on database tables, this information is automatically imported.

Links

Links are created from the foreign key definitions entered in the database. The application screen must contain links if you are using the transaction manager and the screen contains more than one table view.

Check the link properties to see if they need to be edited for your application screen. The Parent and Child properties might need to be reversed or the Link Type might need to be changed.

Refer to Chapter 30 in the *Developer's Guide* for more information on links.

Widgets

A widget is created for each database column. The name of the widget corresponds to the database column name. The Inherit From property is set to `@DATABASE` indicating that the widget was imported from the database engine. The Justification property is set to Left. Other widget properties are assigned based on the data type.

The following table lists the values for the C Type, Length, and Precision properties assigned to each ORACLE data type.

Table 2. Importing Database Tables

ORACLE Data Type	Prolifics Type	C Type	Widget Length	Widget Precision
`CHAR`	`FT_CHAR`	`Char String`	Column length	
`DATE`	`DT_DATETIME`	`Default`	20	
`LONG`	`FT_CHAR`	`Char String`	36	
`LONG RAW`	`DT_BINARY`	`Hex Dec`		
`NUMBER` (ORACLE scale = 0)	`FT_LONG`	`Long Int`	Column length plus 1 for sign	
`NUMBER` (ORACLE scale > 0)	`FT_DOUBLE`	`Double`	Column length plus 2 for +/– sign and decimal point	Same as column precision (scale)
`RAW`	`DT_BINARY`	`Hex Dec`	Column length * 2	
`ROWID`	`FT_CHAR`	`Char String`	18	
`VARCHAR2`	`FT_CHAR`	`Char String`	Column length	

Precision in ORACLE is equivalent to length in Prolifics, and scale in ORACLE is equivalent to precision in Prolifics.

Other Widget Properties

Based on the column's data type or on the Prolifics type assigned during the import process, other widget properties might be automatically set when importing database tables.

UseInUpdate property

If a column's length is defined as larger than 254 in the database, then the database importer sets the Use In Update property to No for the widget corresponding to that column. Because widgets in Prolifics have a maximum length of 254, the data originally in the database column could be truncated as part of a `SAVE` command in the transaction manager.

In Prolifics for ORACLE, this is applied to `LONG RAW` and `RAW` data types.

DT_DATETIME

`DT_DATETIME` widgets also have the Format/Display⇒Data Formatting property set to Date/Time and Format Type set to `DEFAULT`. Note that dates in this Format Type appear as:

```
MM/DD/YY HH:MM
```

Null Field property

If a column is defined to be `NOT NULL`, the Null Field property is set to No. For example, the `roles` table in the `videobiz` database contains three columns:

title_id, actor_id and role. title_id and actor_id are defined as NOT NULL so the Null Field property is set to No. role, without a NOT NULL setting, is implicitly considered to allow null values so the Null Field property is set to Yes.

For more information about usage of Prolifics type and C type, refer to Chapter 29 of the *Developer's Guide*.

Formatting for Colon Plus Processing and Binding

This section contains information about the special data formatting that is performed for the engine. For general information on data formatting, refer to Chapter 29 in the *Developer's Guide*.

Formatting Dates

Prolifics uses ORACLE's built-in TO_DATE function and the ORACLE format string, ddmmyyyy hh24miss to convert a Prolifics date-time format to an ORACLE format.

Formatting Character Strings

Long Character String Values

ORACLE 6 does not permit quoted character strings longer than 255 characters. Furthermore, in all versions of ORACLE, there is a 64K limit on the size of a SQL statement. Therefore, you should not use colon-plus processing to supply long character string values (e.g., LONG, VARCHAR2) in a SQL INSERT or UPDATE statement. Instead, you should use binding to supply the character string. For example:

```
DBMS DECLARE x CURSOR FOR INSERT INTO mytable \
   (code, comments) VALUES (::code, ::comments)

DBMS WITH CURSOR x EXECUTE USING code-fld, comments-fld
```

Typically, a word-wrapped multi-text array is used for these long strings.

Empty Character Strings

In Prolifics for ORACLE, colon plus processing expands an empty character string ('') to a quoted space (' ') if the widget's Null Field property is set to No. This is to circumvent ORACLE's behavior. Since ORACLE converts an empty character string to NULL, null values were being entered into the database even though they were not specified.

Specifying Optimization Hints

In ORACLE, you can specify the optimization of a SQL statement by including hints in the statement itself. Because the syntax for hints matches Prolifics's syntax for comments, you must escape the first slash to prevent the hint from being interpreted as a comment.

For example, to include the hint `/*+ ALL ROWS */` in the SQL statement, the statement would be written as follows:

```
DBMS SQL SELECT \/*+ ALL ROWS */ empno, ename, job FROM emp
```

Refer to your ORACLE documentation for more information on using hints.

Declaring Cursors

When a connection is declared to an ORACLE engine, Prolifics automatically declares a default cursor for SQL `SELECT` statements executed with the JPL command `DBMS SQL`. For all non-`SELECT` operations performed with `DBMS SQL`, Prolifics uses ORACLE's `EXECUTE IMMEDIATE` feature rather than another default cursor. If the application needs to select multiple rows and update the rows one at a time, the application does not need to declare named cursors.

Declaring a named cursor might improve the performance of some `SELECT` statements. In particular, if an application is executing a `SELECT` statement more than once and the `SELECT` fetches 40 or more columns from a remote server, a named cursor is recommended. In this case, the parse and describe is done just once when the cursor is declared, not each time the cursor is executed.

For OCI applications, Prolifics does not put any limit on the number of cursors an application can declare to an ORACLE engine. For Pro*C applications, Prolifics defines 10 cursors for an application accessing ORACLE. It reserves one for itself (i.e., the "default" cursor); the other nine are available for the application's use. If the application attempts to declare a tenth cursor, Prolifics returns the `DM_MANY_CURSORS` error. In this case, the application must close a cursor using `DBMS CLOSE CURSOR` before it can declare a new one. If nine cursors are not enough for your application, you must modify the distributed source file `oraemb.pc`.

For more information on cursors, refer to Chapter 27 in the *Developer's Guide*.

Scrolling

Even though ORACLE does not have native support for non-sequential scrolling in a select set, Prolifics scrolling is available. Before using any of the following commands:

`DBMS` *[* `WITH CURSOR` *cursor-name]* `CONTINUE_BOTTOM`

`DBMS` *[* `WITH CURSOR` *cursor-name]* `CONTINUE_TOP`

`DBMS` *[* `WITH CURSOR` *cursor-name]* `CONTINUE_UP`

the application must set up a continuation file for the cursor. This is done with this command:

`DBMS` *[* `WITH CURSOR` *cursor-name]* `STORE FILE` *[filename]*

To turn off Prolifics scrolling and close the continuation file, use this command:

`DBMS` *[* `WITH CURSOR` *cursor-name]* `STORE`

or close the Prolifics cursor with `DBMS CLOSE CURSOR`.

For more information on scrolling, refer to Chapter 28 in the *Developer's Guide*.

Error and Status Information

Prolifics uses the global variables described in the following sections to supply error and status information in an application. Note that some global variables can not be used in the current release; however, these variables are reserved for use in other engines and for use in future releases of Prolifics for ORACLE.

Errors

Prolifics initializes the following global variables for error code information:

`@dmretcode`	Standard database driver status code.
`@dmretmsg`	Standard database driver status message.
`@dmengerrcode`	ORACLE error code.
`@dmengerrmsg`	ORACLE error message.
`@dmengwarncode`	Not used in Prolifics for ORACLE.
`@dmengwarnmsg`	Not used in Prolifics for ORACLE.
`@dmengreturn`	Not used in Prolifics for ORACLE.

ORACLE returns error codes and messages when it aborts a command. It usually aborts a command because the application used an invalid option or because the user did not have the authority required for an operation. Prolifics writes ORACLE error codes to the global variable `@dmengerrcode` and writes ORACLE messages to `@dmengerrmsg`.

All ORACLE errors are Prolifics errors. Therefore, Prolifics always calls the default error handler or the installed error handler when an error occurs.

Using the Default Error Handler

The default error handler displays a dialog box if there is an error. The first line indicates whether the error came from the database driver or database engine, followed by the text of the statement that failed. If the error comes from the database driver, `Database interface` appears in the `Reported by` list along with the database engine. The error number and message contain the values of `@dmretcode` and `@dmretmsg`. If the error comes from the database engine, only the name of the engine appears in the `Reported by` list. The error number and message contain the values of `@dmengerrcode` and `@dmengerrmsg`.

Using an Installed Error Handler

An installed error or exit handler should test for errors from the database driver and from the database engine. For example:

```
DBMS ONERROR JPL errors
DBMS DECLARE dbi_session CONNECTION FOR ...

proc errors (stmt, engine, flag)
if @dmengerrcode == 0
   msg emsg ″JAM error: ″ @dmretmsg
else
   msg emsg ″JAM error: ″ @dmretmsg ″ %N″ \
   ″:engine error is ″ @dmengerrcode ″ ″ @dmengerrmsg
return 1
```

For additional information about engine errors, refer to your ORACLE documentation. For more information about error processing in Prolifics, refer to Chapter 36 in the *Developer's Guide* and Chapter 12 in the *Programming Guide*.

Row Information

Prolifics initializes the following global variables for row information:

`@dmrowcount`	Count of the number of ORACLE rows affected by an operation.
`@dmserial`	Not used in Prolifics for ORACLE.

ORACLE returns a count of the rows affected by an operation. Prolifics writes this value to the global variable `@dmrowcount`.

As explained on the manual page for `@dmrowcount`, the value of `@dmrowcount` after a SQL `SELECT` is the number of rows fetched to Prolifics variables. This number is less than or equal to the total number of rows in the select set. The value of `@dmrowcount` after a SQL `INSERT`, `UPDATE`, or `DELETE` is the total number of rows affected by the operation. Note that this variable is reset when another `DBMS` statement is executed, including `DBMS COMMIT`.

Using Stored Procedures

A stored subprogram is a precompiled set of SQL statements that are recorded in the database and executed by calling the subprogram name. Since the SQL parsing and syntax checking for a stored subprogram are performed when the subprogram is created, executing a stored subprogram is faster than executing the same group of SQL statements individually. By passing parameters to and from the stored subprogram, the same procedure can be used with different values. In addition to SQL statements, stored subprograms can also contain control flow language, such as `if` statements, which gives greater control over the processing of the statements.

Database engines implement stored subprograms very differently. If you are porting your application from one database engine to another, you need to be aware of the differences in the engine implementation.

ORACLE as part of its PL/SQL language has two types of subprograms: stored procedures and stored functions. Prolifics support for each type of subprogram is discussed in the following sections. To access to stored subprograms, you must use ORACLE's OCI Interface with Version 7 of ORACLE. Consult the file `$SMBASE/notes/readme.ora` for the file names and versions of ORACLE libraries needed. For more information on writing stored subprograms, refer to your ORACLE PL/SQL documentation.

Executing Stored Procedures

To execute a stored procedure, you must declare a named cursor. The `DECLARE CURSOR` statement must include the keyword `STORED_SUB`. All parameters to the stored procedure must have corresponding bind parameters in the `DECLARE CURSOR` statement.

PL/SQL defines three modes for parameters: input, output and input/output. An input parameter can be a constant, literal, initialized variable, or expression. Arrays are not supported as input parameters in this release. Output and input/output parameters must be variables.

The output parameters in a stored procedure must be a table data type. Record data types are not supported as output parameters in this release.

The syntax for the `DECLARE CURSOR` statement is as follows:

```
DBMS DECLARE cursor-name CURSOR FOR STORED_SUB \
    [ package-name. ]procedure-name [ (::parameter [, ::[ parameter ]...) ]
```

When the cursor is executed, the Prolifics variables named in the `USING` clause must have enough occurrences to hold all the rows that are returned. You cannot use a `DBMS CONTINUE` command to fetch additional rows.

The Prolifics variables must also be equal to, or greater than, the length of the output parameter. Otherwise, ORACLE returns error 6502.

Use one of the following formats to execute the cursor:

```
DBMS [ WITH CURSOR cursor ] EXECUTE [ USING variable [, variable ... ] ]
```

```
DBMS [ WITH CURSOR cursor ] EXECUTE [ USING parameter=variable \
    [, parameter=variable ... ] ]
```

Return Codes

ORACLE stored procedures, by definition, do not have return codes.

Example

For example, `update_tapes` is a stored procedure that changes the video tape status to `O` whenever a video is rented.

```
PROCEDURE update_tapes (tid IN INTEGER, copy IN INTEGER) IS
BEGIN
   UPDATE tapes SET status = 'O'
      WHERE title_id = tid AND copy_num = copy;
END update_tapes;
```

The following JPL procedure executes this stored procedure. First, a `DECLARE CURSOR` statement identifies the parameters. Then, the cursor is executed with a `USING` clause that gets the onscreen values of the widgets `title_id` and `copy_num`.

```
proc sp1
DBMS DECLARE x CURSOR FOR STORED_SUB update_tapes \
    (::parm1, ::parm2)
DBMS WITH CURSOR x EXECUTE USING parm1=title_id,\
   parm2=copy_num
return
```

Remember to use double colons (::) in a `DECLARE CURSOR` statement for cursor parameters. If a single colon or colon-plus were used, the data would be supplied when the cursor was declared, not when it was executed. Refer to Chapter NO TAG in the *Developer's Guide* for more information.

Example

`rent_history` is a stored procedure containing both input and output parameters, which finds the video rentals for a customer.

```
CREATE PACKAGE rentals AS
   TYPE charArrayTyp IS TABLE OF CHAR(30)
      INDEX BY binary_integer;
   TYPE dateArrayTyp IS TABLE OF DATE
      INDEX BY binary_integer;
   TYPE numArrayTyp IS TABLE OF INTEGER
      INDEX BY binary_integer;
PROCEDURE rent_history (
   cid      IN    INTEGER,
   tid      OUT   numArrayTyp,
   tname    OUT   charArrayTyp,
   rstatus  OUT   charArrayTyp,
   due_date OUT   dateArrayTyp,
   ret_date OUT   dateArrayTyp) IS
BEGIN
   SELECT rentals.title_id, titles.name,
   rentals.rental_status, rentals.due_back,
   rentals.return_date
   INTO tid, tname, rstatus, due_date, ret_date
   FROM rentals, titles
      WHERE rentals.title_id = titles.title_id AND
      cust_id = cid;
END rent_history;
```

The following JPL procedure executes the stored procedure. First, a `DECLARE CURSOR` statement identifies the name of the stored procedure and its parameters. Then, the cursor is executed with a `USING` clause that gets the onscreen value of `cust_id` and returns the output parameters to arrays having a maximum number of occurrences large enough to hold the select results.

```
proc sp3
DBMS DECLARE y CURSOR FOR STORED_SUB rentals.rent_history \
    (::parm1, ::parm2, ::parm3, ::parm4, ::parm5, ::parm6)
DBMS WITH CURSOR y EXECUTE USING parm1=cust_id,\
   parm2=title_id, parm3=name, parm4=rental_status, \
   parm5=due_back, parm6=return_date
return
```

Executing Stored Functions

To execute a stored function, you must also use a `DECLARE CURSOR` statement including the keyword `STORED_SUB`. However, since a stored function has a return code, the syntax of the statement differs from the syntax used for stored procedures.

In the current version of Prolifics for ORACLE, the return code must be one of the scalar data types (`CHAR`, `INT`, `REAL`, etc.).

```
DBMS DECLARE cursor-name CURSOR FOR STORED_SUB \
    ::parameter1 ::= function-name (::parameter [ , ::[ parameter ]... ])
```

In this statement, *parameter1* holds the return code. *function-name* is any existing ORACLE stored function. Any other parameters follow the function name. All parameters to the stored function must have corresponding bind parameters in the `DECLARE CURSOR` statement.

When the cursor is executed, the return code is written to *variable1*. Any additional parameters follow the return code.

```
DBMS [ WITH CURSOR cursor ] EXECUTE USING variable1 [ , variable# ... ]
```

Return Codes

The return code from an ORACLE stored function is not written to the Prolifics variable `@dmengreturn`. Because the `@dmengreturn` is designed to hold integer values and the return code from a stored function can be of any data type, it is written to the first Prolifics variable in an `EXECUTE USING` statement as illustrated in the preceding examples.

Example

`cust_rent` calculates the new total `rent_amount` column in the `customers` table.

```
FUNCTION cust_rent (cid IN  INTEGER, total IN REAL) RETURN
REAL IS
   old_rent REAL;
   calc_rent REAL;
BEGIN
   SELECT rent_amount INTO old_rent FROM customers
      WHERE cust_id = cid;
   calc_rent := total + old_rent;
RETURN calc_rent;
END cust_rent;
```

The following JPL procedure executes the stored function. First, a `DECLARE CURSOR` statement identifies the parameters and return code. Then, the cursor is executed with a `USING` clause that gets the onscreen value of `cust_id` and `total` and returns the `title_id` and `copy_num`.

```
proc sp3
DBMS DECLARE z CURSOR FOR STORED_SUB ::a \
   ::=cust_rent (::b, ::c)
DBMS WITH CURSOR z EXECUTE USING calc_rent, cust_id, total
return
```

Using Transactions

A transaction is a unit of work that must be totally completed or not completed at all. ORACLE has one transaction for each connection. Therefore, in a Prolifics application, a transaction controls all statements executed with a single named connection or the default connection.

The following events commit a transaction on ORACLE:

- Executing `DBMS COMMIT`.
- Executing a data definition command such as `CREATE`, `DROP`, `RENAME`, or `ALTER`, which causes an implicit commit.
- Closing the connection.

The following events roll back a transaction on ORACLE:

- Executing `DBMS ROLLBACK`.

When an application closes a connection with `CLOSE_ALL_CONNECTIONS` or `CLOSE CONNECTION`, ORACLE commits any pending transactions on those connections. If an application terminates without explicitly closing its connections, ORACLE rolls back any pending transactions on those connections. However, these procedures are not recommended. Instead, it is strongly recommended that applications use explicit `COMMIT` and `ROLLBACK` statements to terminate transactions.

For information on transaction processing for ORACLE XA connections, refer to page 22.

Transaction Control on a Single Connection

After an application declares a connection, a transaction automatically starts on that connection.

ORACLE supports the following transaction commands:

- Set availability of autocommit processing.

  ```
  DBMS [WITH CONNECTION connection] AUTOCOMMIT { ON | OFF }
  ```

- Commit the transaction on a default or named connection.

  ```
  DBMS [WITH CONNECTION connection] COMMIT
  ```

- Rollback to a savepoint or to the beginning of the transaction on a default or named connection.

  `DBMS` *[* `WITH CONNECTION` *connection]* `ROLLBACK` *[savepoint]*

- Create a savepoint in the transaction on a default or named connection.

  `DBMS` *[* `WITH CONNECTION` *connection]* `SAVE` *[savepoint]*

The setting for autocommit processing also determines the availability of other transaction commands. If the setting is `AUTOCOMMIT ON`, every statement is committed immediately. The other transaction commands—`COMMIT`, `ROLLBACK`—are invalid. If the setting is `AUTOCOMMIT OFF`, the statements in a transaction must be committed in order for the work to be saved and visible to the rest of the application or other users. `AUTOCOMMIT OFF` is the default setting.

Example

The following example contains a transaction on the default connection with an error handler.

```
# Call the transaction handler and pass it the name
# of the subroutine containing the transaction commands.

call tran_handle ”new_title()”

proc tran_handle (subroutine)
{
# Declare a variable jpl_retcode and
# set it to call the subroutine.
   vars jpl_retcode
   jpl_retcode = :subroutine

# Check the value of jpl_retcode. If it is 0, all statements
# in the subroutine executed successfully and the transaction
# was committed. If it is 1, the error handler aborted the
# subroutine. If it is -1, Prolifics aborted the subroutine.
# Execute a ROLLBACK for all non-zero return codes.

   if jpl_retcode == 0
   {
      msg emsg ”Transaction succeeded.”
   }
   else
   {
      msg emsg ”Aborting transaction.”
      DBMS ROLLBACK
   }
}
```

```
proc new_title
    DBMS SQL INSERT INTO titles VALUES \
        (:+title_id, :+name, :+genre_code, \
        :+dir_last_name, :+dir_first_name, :+film_minutes, \
        :+rating_code, :+release_date, :+pricecat)
    DBMS SQL INSERT INTO title_dscr VALUES \
        (:+title_id, :+line_no, :+dscr_text)
    DBMS SQL INSERT INTO tapes VALUES \
        (:+title_id, :+copy_num, :+status, :+times_rented)
DBMS COMMIT
return 0
```

The procedure `tran_handle` is a generic handler for the application's transactions. The procedure `new_title` contains the transaction statements. This method reduces the amount of error checking code.

The application executes the transaction by executing

```
call tran_handle "new_title()"
```

The procedure `tran_handle` receives the argument "new_title" and writes it to the variable `subroutine`. It declares a JPL variable, `jpl_retcode`. After performing colon processing, `:subroutine` is replaced with its value, `new_title`, and JPL calls the procedure. The procedure `new_title` begins the transaction, performs three inserts, and commits the transaction.

If `new_title` executes without any errors, it returns 0 to the variable `jpl_retcode` in the calling procedure `tran_handle`. JPL then evaluates the `if` statement, displays a success message, and exits.

If however an error occurs while executing `new_title`, Prolifics calls the application's error handler. The error handler should display any error messages and return the abort code, 1.

For example, assume the first `INSERT` in `new_title` executes successfully but the second `INSERT` fails. In this case, Prolifics calls the error handler to display an error message. When the error handler returns the abort code 1, Prolifics aborts the procedure `new_title` (therefore, the third `INSERT` is not attempted). Prolifics returns 1 to `jpl_retcode` in the calling procedure `tran_handle`. JPL evaluates the `if` statement, displays a message, and executes a rollback. The rollback undoes the insert to the table `titles`.

Transaction Manager Processing

Transaction Model for ORACLE

Each database driver contains a standard transaction model for use with the transaction manager. The transaction model is a C program which contains the main processing for each of the transaction manager commands. You can edit this program; however, be aware that the transaction model is subject to change with each release. For ORACLE, the name of the standard transaction model is `tmora1.c`.

In Tuxedo, the transaction model for Oracle supports database transactions using the XA interface. For XA connections, the transaction model can call `sm_tp_exec` to begin, rollback, or commit the database transaction instead of using `DBMS` commands.

Specifying FOR UPDATE Clauses

The `dm_gen_change_select_suffix` function appends text to SQL `SELECT` statements generated by the transaction manager. You can use this function to append a `FOR UPDATE` clause during SQL generation.

SAVE Commands

If you specify a `SAVE` command with a table view parameter, it is called a partial command. A partial command is not applied to the entire transaction tree. In the standard transaction models, partial `SAVE` commands do not commit the database transaction. In order to save those changes, you must do an explicit `DBMS COMMIT`. Otherwise, those changes could be rolled back if the database engine performs an automatic rollback when the database connection is closed.

Using the XA Interface

With the XA interface, the transaction processing monitor provided by the transaction manager vendor starts and ends a transaction that can include operations on several resource managers, including ORACLE.

Since ORACLE does not control the transaction processing in the XA environment, the following commands should not be used with ORACLE XA connections:

`DBMS` *[* `WITH CONNECTION` *connection]* `AUTOCOMMIT { ON | OFF }`

`DBMS` *[* `WITH CONNECTION` *connection]* `COMMIT`

`DBMS` *[* `WITH CONNECTION` *connection]* `ROLLBACK` *[savepoint]*

`DBMS` *[* `WITH CONNECTION` *connection]* `SAVE` *[savepoint]*

Also, because SQL data definition statements such as `CREATE TABLE` cause an implicit commit in , these statements should not be executed on ORACLE XA connections.

For additional information about ORACLE's XA library, refer to your ORACLE 7 Server for UNIX Administrator's Reference.

ORACLE-Specific Commands

Prolifics for ORACLE provides commands for ORACLE-specific features. This section contains a reference page for each command. If you are using multiple engines or are porting an application to or from another engine, please note that these commands may work differently or may not be supported on some engines.

Using Stored Procedures

`DECLARE CURSOR FOR STORED_SUB`	Declare a cursor to execute a stored subprogram.

Using Transactions

`AUTOCOMMIT`	Turn autocommit processing on or off.
`COMMIT`	Commit a transaction.
`ROLLBACK`	Rollback a transaction.
`SAVE`	Set a savepoint in a transaction.

AUTOCOMMIT

Turn autocommit transaction processing on or off

`DBMS` *[* `WITH CONNECTION` *connection-name]* `AUTOCOMMIT ON`

`DBMS` *[* `WITH CONNECTION` *connection-name]* `AUTOCOMMIT OFF`

`WITH CONNECTION` *connection-name*	Specify the connection for this command. If this clause is not included, Prolifics issues the command on the default connection.

Environment

This command is not available for ORACLE XA connections.

Description

This command controls whether changes to a database occur immediately upon execution of an `INSERT`, `UPDATE`, or `DELETE` command, or whether they occur when a `DBMS COMMIT` is explicitly executed.

The default setting is `AUTOCOMMIT OFF`. This means that the engine automatically starts a transaction after an application declares a connection. When a recoverable statement (`INSERT`, `UPDATE`, and `DELETE`) is executed, it is not automatically committed. The effects of the statement are not visible until the transaction is terminated. If the transaction is terminated by `DBMS COMMIT`, the updates are committed and visible to other users. If the transaction is terminated by `DBMS ROLLBACK`, the updates are not committed, and the database is restored to its state prior to the start of the transaction. After a transaction is terminated, the engine automatically begins a new transaction.

If the setting is changed to `AUTOCOMMIT ON`, a statement is committed automatically upon successful execution. Its effects are immediately visible to other users, and it cannot be rolled back.

ORACLE recommends `AUTOCOMMIT OFF` mode to improve performance.

Example

```
proc new_title
   DBMS WITH CONNECTION xxx1 AUTOCOMMIT ON
   call update_title
   msg emsg ”New title data successfully entered.”
   DBMS WITH CONNECTION xxx1 AUTOCOMMIT OFF
return 0
```

```
proc update_title
   DBMS SQL INSERT INTO titles VALUES \
      (:+title_id, :+name, :+genre_code, \
      :+dir_last_name, :+dir_first_name, :+film_minutes, \
      :+rating_code, :+release_date, :+pricecat)
   DBMS SQL INSERT INTO title_dscr VALUES \
      (:+title_id, :+line_no, :+dscr_text)
   DBMS SQL INSERT INTO tapes VALUES \
      (:+title_id, :+copy_num, :+status, :+times_rented)
return 0
```

See Also

`COMMIT`

`ROLLBACK`

`SAVE`

COMMIT

Commit a transaction

`DBMS` *[* `WITH CONNECTION` *connection-name]* `COMMIT`

`WITH CONNECTION` *connection-name*	Specify the connection for this command. If the command does not contain a `WITH CONNECTION` clause, Prolifics issues the commit on the default connection.

Environment

This command is not available for ORACLE XA connections.

Description

Use this command to commit a pending transaction. Committing a transaction saves all the work since the last `COMMIT` or `SAVE`. Changes made by the transaction become visible to other users. If the transaction is terminated by `ROLLBACK`, the updates are not committed, and the database is restored to its state prior to the start of the transaction.

After a transaction is terminated, the engine automatically begins a new transaction.

Before beginning a transaction, the application should ensure that the connection is using `AUTOCOMMIT OFF` mode; this is usually the default. It should `COMMIT` or `ROLLBACK` any pending transactions before starting a new one.

If an application is using `AUTOCOMMIT ON` mode, this command is not needed.

This command is available depending on the setting of various parameters in your environment. Refer to the section on transactions and your documentation for more information.

Example

Refer to the example in Using Transactions on page 19.

See Also

Using Transactions on page 19

`AUTOCOMMIT`

```
ROLLBACK

SAVE
```

DECLARE CURSOR FOR STORED_SUB

Declare a named cursor for a stored subprogram

```
DBMS [ WITH CONNECTION connection-name ] DECLARE cursor-name CURSOR FOR STORED_SUB \
    [ package-name. ]procedure-name [ (::parameter[ , ::parameter ]...) ]
```

```
DBMS [ WITH CONNECTION connection-name ] DECLARE cursor-name CURSOR FOR STORED_SUB \
    ::return-code ::function-name (::parameter [ , ::[ parameter ]... ])
```

function-name	Specifies the stored function name.
package-name	Specifies the PL/SQL package containing the stored subprogram.
parameter	For stored procedures, specifies an input or output parameter used in the stored procedure. For stored functions, specifies input parameter used in the stored function.
procedure-name	Specifies the stored procedure name.
return-code	Specifies the name of the return code in the stored function.
`WITH CONNECTION` *connection-name*	Specify the connection for this command. If this clause is not included, Prolifics associates the cursor with the default connection.

Description

Use this command to create or redeclare a named cursor to execute a stored subprogram. The keyword `STORED_SUB` is required and can be used for both stored procedures and stored functions. However, the format of the command varies for these two types of subprograms. The first format shown is for stored procedures. The second format is for stored functions.

All parameters must begin with a double colon, which is the Prolifics syntax for cursor parameters.

The application executes a cursor associated with a stored subprogram as it executes any named cursor, with `DBMS EXECUTE`. However, the format of this command differs for stored procedures and stored functions. Refer to the examples in Using Stored Subprograms on page 15.

Example

Refer to the example in Using Stored Subprograms on page 15.

See Also

Using Stored Subprograms on page 15

ROLLBACK

Roll back a transaction

`DBMS` *[* `WITH CONNECTION` *connection-name]* `ROLLBACK` *[savepoint]*

`WITH CONNECTION` *connection-name*	Specify the connection for this command. If the command does not contain a `WITH CONNECTION` clause, Prolifics issues the rollback on the default connection.
savepoint	If included, only the statements that were issued after the specified savepoint are rolled back.

Environment

This command is not available for ORACLE XA connections.

Description

Use this command to rollback a transaction and restore the database to its state prior to the start of the transaction or at the time of the specified savepoint.

If a statement in a transaction fails, an application must attempt to reissue the statement successfully or else roll back the transaction. If an application cannot complete a transaction, it should roll back the transaction. If it does not, it might inadvertently commit the partial transaction when it commits a later transaction.

Example

Refer to the example in Using Transactions on page 19.

See Also

Using Transactions on page 19

`AUTOCOMMIT`

`COMMIT`

`SAVE`

SAVE

Set a savepoint within a transaction

`DBMS` *[* `WITH CONNECTION` *connection-name]* `SAVE` *savepoint*

savepoint	Specify the name of the savepoint.
`WITH CONNECTION` *connection-name*	Specify the connection for this command. If this clause is not included, Prolifics issues the command on the default connection.

Environment

This command is not available for ORACLE XA connections.

Description

This command creates a savepoint in the transaction. A savepoint is a place-marker set by the application within a transaction. When a savepoint is set, the statements following the savepoint can be cancelled using `DBMS ROLLBACK` *savepoint*. A transaction can have multiple savepoints.

When the transaction is rolled back to a savepoint, the transaction must then either be completed *or* rolled back to the beginning.

This feature is useful for any long, complicated transaction. For example, an order entry application might involve many screens where an end-user must enter data regarding the order. As the user completes each screen, the application can issue a savepoint. Therefore, if an error occurs on the fifth screen, the application can simply rollback the procedures on the fifth screen.

Example

```
proc new_title
   DBMS SQL INSERT INTO titles VALUES \
      (:+title_id, :+name, :+genre_code, \
      :+dir_last_name, :+dir_first_name, :+film_minutes, \
      :+rating_code, :+release_date, :+pricecat)
   DBMS SAVE s1
call new_dscr
call new_tapes
DBMS COMMIT
return 0
```

```
proc new_dscr
   DBMS SQL INSERT INTO title_dscr VALUES \
      (:+title_id, :+line_no, :+dscr_text)
   DBMS SAVE s2
return 0

proc new_tapes
   DBMS SQL INSERT INTO tapes VALUES \
      (:+title_id, :+copy_num, :+status, :+times_rented)
return 0
```

See Also

Using Transactions on page 19

`AUTOCOMMIT`

`COMMIT`

`ROLLBACK`

Command Directory for ORACLE

The following table lists all commands available in Prolifics's database driver for ORACLE. Commands available to all database drivers are described in the *Programming Guide*.

Table 3. Commands for ORACLE

Command Name	Description	Documentation Location
`ALIAS`	Name a Prolifics variable as the destination of a selected column or aggregate function	*Programming Guide*
`AUTOCOMMIT`	Turn on/off autocommit processing	page 24
`BINARY`	Create a Prolifics variable for fetching binary values	page 810
`CATQUERY`	Redirect select results to a file or a Prolifics variable	
`CLOSE_ALL_CONNECTIONS`	Close all connections on all engines	
`CLOSE CONNECTION`	Close a named connection	
`CLOSE CURSOR`	Close a named cursor	
`COLUMN_NAMES`	Return the column name, not column data, to a Prolifics variable	
`COMMIT`	Commit a transaction	page 26
`CONNECTION`	Set a default connection and engine for the application	
`CONTINUE`	Fetch the next screenful of rows from a select set	*Database Guide* & *Database Drivers*
`CONTINUE_BOTTOM`	Fetch the last screenful of rows from a select set	*Database Guide* & *Database Drivers*
`CONTINUE_DOWN`	Fetch the next screenful of rows from a select set	*Database Guide* & *Database Drivers*

Command Name	Description	Documentation Location
`CONTINUE_TOP`	Fetch the first screenful of rows from a select set	*Database Guide* & *Database Drivers*
`CONTINUE_UP`	Fetch the previous screenful of rows from a select set	*Database Guide* & *Database Drivers*
`DECLARE CONNECTION`	Declare a named connection to an engine	*Database Guide* & *Database Drivers*
`DECLARE CURSOR`	Declare a named cursor	*Database Guide* & *Database Drivers*
`DECLARE CURSOR FOR STORED_SUB`	Declare a cursor to execute a stored subprogram	page 28
`ENGINE`	Set the default engine for the application	
`EXECUTE`	Execute a named cursor	
`FORMAT`	Format the results of a `CAT-QUERY`	
`OCCUR`	Set the number of rows for Prolifics to fetch to an array and set the occurrence where Prolifics should begin writing result rows	
`ONENTRY`	Install a JPL procedure or C function that Prolifics will call before executing a `DBMS` statement	
`ONERROR`	Install a JPL procedure or C function that Prolifics will call when a `DBMS` statement fails	*Database Guide* & *Database Drivers*
`ONEXIT`	Install a JPL procedure or C function that Prolifics will call after executing a `DBMS` statement	
`ROLLBACK`	Roll back a transaction	page 29
`SAVE`	Set a savepoint in a transaction	page 30

Command Name	Description	Documentation Location
`START`	Set the first row for Prolifics to return from a select set	
`STORE`	Store the rows of a select set in a temporary file so the application can scroll through the rows	
`UNIQUE`	Suppress repeating values in a selected column	
`WITH CONNECTION`	Specify the connection to use for a command	
`WITH CURSOR`	Specify the cursor to use for a command	
`WITH ENGINE`	Specify the engine to use for a command	