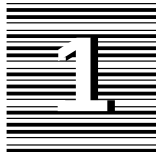
Panther

Database Driver–Informix

Release 4.25

Prolifics®

May 2000



Database Driver for Informix

This chapter provides documentation specific to Informix. 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 Informix, this results in the following `vendor_list` structure in `dbiinit.c`:

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

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

The settings are as follows:

`informix`	Engine name. May be changed.
`dm_infsup`	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 Informix, the settings can be changed by editing the `makevars.inf` file.

Engine Name

You can change the engine name associated with the support routine `dm_infsup`. 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.inf` file:

```
INF_ENGNAME=tracking
```

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

```
DBMS ENGINE informix-engine-name
```

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

```
DBMS ENGINE informix
```

or

```
DBMS ENGINE tracking
```

Support Routine Name

`dm_sup` is the name of the support routine for Informix. 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 Informix column names to either a Prolifics variable name or to a column name in a `DBMS ALIAS` statement.

Informix is case insensitive. Regardless of the case in a SQL statement, Informix creates all database objects—tables, views, columns, etc.—with lower case names. For Informix, the `DM_DEFAULT_CASE` setting is treated as `DM_FORCE_TO_LOWER_CASE`. Because Informix uses only lower case, the `DM_FORCE_TO_LOWER_CASE` setting is the same as `DM_PRESERVE_CASE`. For either of these flags, Prolifics attempts to match Informix 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.

If you wish to use upper case variable names, substitute the `u` option in the `makevars` file that sets the `DM_FORCE_TO_UPPER_CASE` flag.

```
INF_INIT=u
```

If you edit `makevars.inf`, 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

Informix 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 Informix:

Table 1. Database connection options.

Option	Argument
USER	*user-name*
SERVER	*server-name*
DATABASE	*database-name*
PASSWORD	*password*
DB_PATH	*database-path*
HOST	*host-name*
PROTOCOL	*protocol-type*
SERVICE	*service*

For UNIX, DATABASE, which specifies the database name, is the only option.

For Windows, you have the USER, PASSWORD, DATABASE, DB_PATH, HOST, PROTOCOL, and SERVICE options.

USER and PASSWORD are the user name and password for your account on the host computer.

DATABASE specifies the database name, and DB_PATH specifies the database path.

HOST specifies a character string to identify the host computer with which you establish a connection.

PROTOCOL is the name of the protocol used by your network.

SERVICE is the service name that the remote database server uses to listen to all incoming requests.

Note that you can declare a connection without the DATABASE option if you set the database with DBMS SQL DATABASE *database-name*.

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

Importing Database Tables

The Import⇒Database Objects option in the screen editor creates Prolifics repository entries based on database tables in an Informix database. When the

import process is complete, each selected database table has a corresponding repository entry screen.

In Prolifics for Informix, 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. If you are working with a version of Informix that does not support foreign keys, you must create the links needed by the transaction manager manually if the application screen contains more than one table view.

If you are using the screen wizard to create screens, the links must also be added to the repository entries in order for the wizard to allow more than one table view in each section of a screen.

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 Informix data type.

Table 2. Importing Database Tables

Informix Data Type	Prolifics Type	C Type	Widget Length	Widget Precision
char	FT_CHAR	Char String	Column length	
date	DT_DATETIME	Default	20	
datetime	DT_DATETIME	Default	20	
decimal	FT_DOUBLE	Double	Column length plus 2 for +/– sign and decimal point	Column scale
float	FT_DOUBLE	Double	16	2
integer	FT_LONG	Long Int	11	
interval	FT_CHAR	Char	Varies according to column qualifiers	
money	DT_CURRENCY	Default	16	
serial	FT_LONG	Long Int	11	
smallfloat	FT_FLOAT	Float	16	2
smallint	FT_INT	Int	6	
varchar	FT_CHAR	Char	Column length	

Precision in Informix is equivalent to length in Prolifics, and scale in Informix 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.

DT_CURRENCY

`DT_CURRENCY` widgets have the Format/Display⇒Data Formatting property set to Numeric and Format Type set to 2 Dec Places.

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

Informix supports three types of date data types:

Declaring Cursors

When a connection is declared to an Informix 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 Informix'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.

If you use Informix 5, `SELECT` cursors can be either `HOLD` cursors or non-`HOLD` cursors. If the cursor is a `HOLD` cursor, it maintains its positioning information while other cursors perform `INSERT`, `UPDATE`, and `DELETE` statements. This allows you to fetch additional data with `DBMS CONTINUE` after committing or rolling back another transaction. If a cursor is a non-`HOLD` cursor, it is closed at the end of a transaction. Informix closes all non-`HOLD` cursors when it commits or rolls back a transaction.

By default, Prolifics for Informix declares all cursors as `HOLD` cursors. To cause all subsequently declared cursors to be non-`HOLD` cursors, issue the following command:

```
DBMS SET HOLD_DEFAULT OFF
```

This can be reversed and cause cause all subsequently declared cursors to be `HOLD` cursors by issuing the following:

```
DBMS SET HOLD_DEFAULT ON
```

Both of these commands affect only cursors declared after the command is executed. Currently active cursors are not affected.

In addition, you can set the HOLD behavior for an individual cursor with this command:

```
DBMS [WITH CURSOR cursor-name] SET HOLD OFF
```

If the command is issued for the default cursor, all subsequent SELECT statements are with non-HOLD cursors. If the command is issued on a named cursor, then all subsequent executions and declarations of SELECT statements on the cursor are on a non-HOLD cursor. To restore the default behavior, issue the following command:

```
DBMS [WITH CURSOR cursor-name] SET HOLD ON
```

For Informix 5, Prolifics does not put any limit on the number of cursors an application can declare to an Informix engine. For previous versions, Prolifics defines 10 cursors for an application accessing Informix. 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, please contact JYACC Technical Support.

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

Scrolling

Informix has native support for non-sequential scrolling in a select set. This capability is available on any cursor. As an alternative, you can switch to Prolifics scrolling. Both systems allow you to use the following commands:

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

DBMS [WITH CURSOR cursor-name] CONTINUE_TOP

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

For native scrolling, use this command:

```
DBMS [WITH CURSOR cursor-name] SET_BUFFER 1
```

To turn off native scrolling, use this command:

```
DBMS [WITH CURSOR cursor-name] SET_BUFFER 0
```

Then, set Prolifics scrolling 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`.

With Informix-based scrolling, Informix maintains a temporary table to hold the select set. With Prolifics-based scrolling, Prolifics maintains a temporary binary file to hold the select set. A cursor using Informix-based scrolling cannot use the SQL syntax `SELECT FOR UPDATE`. Use Prolifics-based scrolling if you need `SELECT FOR UPDATE`.

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 Informix.

Errors

Prolifics initializes the following global variables for error code information:

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

In Prolifics for Informix, `@dmengerrcode` and `@dmengerrmsg` are arrays that contain both Informix and ISAM information.

`@dmengerrcode [1]`	Informix error message.
`@dmengerrcode [2]`	ISAM error code.
`@dmengerrmsg [1]`	Informix error message.
`@dmengerrmsg [2]`	ISAM error message.

If the error handler queries for the values of `@dmengerrcode` and `@dmengerrmsg` without any occurrence numbers, both sets of codes and messages are returned.

Informix 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 Informix error codes to the global variable `@dmengerrcode` and writes Informix messages to `@dmengerrmsg`.

All Informix 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[1] == 0
   msg emsg ”JAM error: ” @dmretmsg
else
   msg emsg ”JAM error: ” @dmretmsg ” %N” \
   ”INFORMIX error: ” @dmengerrcode[1] ” ” @dmengerrmsg[1] \
   ”ISAM error: ” @dmengerrcode[2] ” ” @dmengerrmsg[2]
return 1
```

For additional information about engine errors, refer to your Informix 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*.

Warnings

Prolifics initializes the following global variables for warning information:

`@dmengwarncode`	Informix warning code.
`@dmengwarnmsg`	Not used in Prolifics for Informix.

Informix uses a warning byte called SQLAWARN to signal conditions it considers unusual but not fatal. @dmengwarncode derives its value from this byte. @dmengwarncode is an 8-occurrence array. If Informix sets a bit in SQLAWARN, Prolifics puts a "W" in the corresponding occurrence of @dmengwarncode.

In Informix, the meaning of these settings depends on the statement that was just executed. Also, Informix might change the value of SQLAWARN between releases. The settings for SQLAWARN after connecting to a database are:

Array Index	Meaning (Informix 5.x)
1	Set to W if any of 2 through 8 are set to W. If this is blank, the other fields do not need to be checked.
2	Set to W if the database has a transaction log that makes transactions available.
3	Set to W if the database is an ANSI database.
4	Set to W if the database server is an Informix On-Line engine.
5	Set to W if the database server stores FLOATs as DECIMALs.
6	Not used.
7	Not used.
8	Not used.

The settings for SQLAWARN for all other operations are:

Array Index	Meaning
1	Set to W if any of 2 through 8 are set to W. If this is blank, the other fields do not need to be checked.
2	Not applicable in Prolifics for Informix.
3	Set to W if an aggregate function encounters a NULL value.
4	Not applicable in Prolifics for Informix.
5	Set to W when a cursor is declared for an UPDATE or DELETE statement and the statement does not contain a WHERE clause.
6	Set to W if the Informix environment variable DBANSIWARN is set and the executed statement does not conform to ANSI SQL syntax.

Array Index	Meaning
7	Not used.
8	Not used.

Before using @dmengwarncode, you should verify these settings for your release of Informix by consulting your Informix documentation.

You might wish to use an exit hook function to process warnings. An exit hook function is installed with DBMS ONEXIT. A sample exit hook function is shown below.

```
proc check_status (stmt, engine, flag)

if @dmretcode == 0
{
 if @dmengwarncode [1] == ”W”
 {
   if @dmengwarncode [3] == ”W”
       msg emsg ”A NULL value was found.”
   if @dmengwarncode [5] == ”W”
       msg emsg ”The operation did not use a WHERE clause.”
   if @dmengwarncode [6] == ”W”
       msg emsg ”This does not conform to ANSI standards.”
 }
}
return
```

Row Information

Prolifics initializes the following global variables for row information:

@dmrowcount	Count of the number of Informix rows affected by an operation.
@dmserial	Informix-generated value for a serial column.

Informix 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.

The value of @dmserial is updated when an application inserts a row into a table with a serial column. Because this variable is cleared when a new DBMS statement is executed, you must copy its value to another location if you wish to use it in subsequent statements.

Using Stored Procedures

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

Database engines implement stored procedures 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.

Executing Stored Procedures

An application can execute a stored procedure with DBMS SQL and the engine's command for execution, EXECUTE PROCEDURE. For example:

DBMS SQL EXECUTE PROCEDURE *procedure-name*

Example

For example, update_tapes is a stored procedure that changes the video tape status to 0 whenever a video is rented.

```
create procedure update_tapes (parm1 int, parm2 int)
update tapes set status = 'O'
   where title_id = parm1 and copy_num = parm2
end procedure
```

The following statement executes this stored procedure, updating the status column of the tapes table using the onscreen values of the widgets title_id and copy_num.

```
DBMS SQL EXECUTE PROCEDURE update_tapes \
   (:+title_id, :+copy_num)
```

A `DECLARE CURSOR` statement can also execute a stored procedure. First, a cursor is declared identifying the parameters. Then, the cursor is executed with a `USING` clause that gets the onscreen values of the widgets `title_id` and `copy_num`.

```
DBMS DECLARE x CURSOR FOR EXECUTE PROCEDURE update_tapes \
   (::parm1, ::parm2)
DBMS WITH CURSOR x EXECUTE USING title_id, copy_num
```

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.

Viewing SELECT Results

In order to return data from a stored procedure in Informix, you must include a `RETURN` statement and a `RETURNING` clause when you create the stored procedure. You can return multiple rows by including a `RETURN WITH RESUME` statement. Also, your application must define positional aliases for the result columns using a `DBMS ALIAS` statement. The order of the variables in this statement must match the order of the variables in the `RETURNING` clause of the stored procedure.

This stored procedure, `avail_video`, selects the video titles that are available for rental and returns values for `title_id`, `name`, and `genre_code` to the application.

```
CREATE PROCEDURE avail_video ()
   RETURNING integer, char(60), char(4);
DEFINE p_title_id integer;
DEFINE p_name char(60);
DEFINE p_genre_code char(4);
DEFINE vcount int;
LET vcount = 1;
FOREACH
SELECT titles.title_id, name, genre_code
INTO p_title_id, p_name, p_genre_code
FROM titles, tapes WHERE titles.title_id = tapes.title_id
AND tapes.status = 'A';
RETURN p_title_id, p_name, p_genre_code WITH RESUME;
LET vcount = vcount +1;
END FOREACH;
END PROCEDURE
;
```

The Prolifics application screen contains three widgets named `title_id`, `name`, and `genre_code`. When the application executes the following statements, the screen displays the available videos.

```
proc get_video
DBMS ALIAS title_id, name, genre_code
DBMS SQL EXECUTE PROCEDURE avail_video ()
return
```

The next example, `unpaid_orders`, uses the `stores` database and returns data about unpaid orders to the application.

```
CREATE PROCEDURE unpaid_orders ()
   RETURNING integer, date, integer, char(10), date;
DEFINE p_order_num integer;
DEFINE p_order_date date;
DEFINE p_customer_num integer;
DEFINE p_po_num char(10);
DEFINE p_ship_date date;
DEFINE lcount int;
LET lcount = 1;
FOREACH
SELECT order_num, order_date, customer_num, po_num, ship_date
INTO p_order_num, p_order_date, p_customer_num, p_po_num,
  p_ship_date
FROM informix.orders
WHERE paid_date is NULL
ORDER BY ship_date
RETURN p_order_num, p_order_date, p_customer_num, p_po_num,
  p_ship_date WITH RESUME;
LET lcount = lcount +1;
END FOREACH;
END PROCEDURE
;
```

The application contains Prolifics variables named `order_num`, `order_date`, `customer_num`, `po_num`, and `ship_date`. The procedure is executed using the following statements. The order of the variables in the `DBMS ALIAS` statement and in the `RETURNING` clause of the procedure are the same.

```
proc unpaid
DBMS ALIAS order_num, order_date, customer_num, po_num, \
   ship_date
DBMS SQL EXECUTE PROCEDURE unpaid_orders ()
return
```

Using Transactions

A transaction is a unit of work that must be totally completed or not completed at all. Informix 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 Informix:

- Executing `DBMS COMMIT`.

The following events roll back a transaction on Informix:

- Executing `DBMS ROLLBACK`.
- Closing the transaction's connection before the transaction is committed.

Informix keeps a record of the database modifications performed in each transaction in a transaction log. It uses this log to undo the database changes when a `ROLLBACK` command is executed. However, Informix databases do not automatically have a transaction log. If transaction processing is not available, see your database administrator to activate this feature.

As noted earlier in the document, the behavior of named cursors differs between Prolifics and Informix when transactions are terminated. A named cursor has actually two representations. One is a Prolifics structure and the other is an Informix cursor in the database. The two representations have the same lifetime (declaring the Prolifics cursor creates the Informix cursor, closing the Prolifics cursor closes the Informix cursor) *except* when a transaction is terminated. When Informix commits or rolls back a transaction, it closes all Informix cursors. Therefore, if an application has a select set pending when it begins a transaction, it cannot fetch the remaining rows after executing a rollback or commit because Informix has closed its cursors and the positioning information is no longer available. To begin the fetch again, the application must simply re-execute the cursor using `DBMS EXECUTE`; it is not necessary to re-declare the Prolifics cursor.

If your application needs to keep the positioning information, you can use the continuation file in Prolifics. Before issuing the select statement, set up the continuation file. Then, fetch all the rows to the continuation file before continuing with the application. For example:

```
proc getrows
# Set up a continuation file. Use WITH CURSOR if needed.
DBMS STORE FILE
#Execute the select.
DBMS SQL SELECT ...
#Fetch all the rows to the continuation file.
DBMS CONTINUE_BOTTOM
#Reposition to the top of the select.
DBMS CONTINUE_TOP
return
```

Transaction Control on a Single Connection

After an application declares a connection, an application can begin a transaction on the default connection or on any declared connection.

Informix supports the following transaction commands:

- Begin a transaction on a default or named connection.

  ```
  DBMS [WITH CONNECTION connection] BEGIN
  ```

- Commit the transaction on a default or named connection.

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

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

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

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 BEGIN
   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 Informix

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 Informix, the name of the standard transaction model is `tminf1.c`.

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.

Informix-Specific Commands

Prolifics for Informix provides commands for Informix-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 Cursors

`SET HOLD`	Control behavior of Informix cursors for `SELECT` statements.
`SET HOLD_DEFAULT`	Set connection behavior for Informix cursors when executing `SELECT` statements.

Using Scrolling

`BUFFER_DEFAULT`	Set buffer size for scrolling for entire application.
`SET_BUFFER`	Control availability of Informix-based scrolling for `DBMS CONTINUE_BOTTOM`, `DBMS CONTINUE_TOP`, `DBMS CONTINUE_UP`.

Using Transactions

`BEGIN`	Begin a transaction.
`COMMIT`	Commit a transaction.
`ROLLBACK`	Rollback a transaction.

BEGIN

Start a transaction

```
DBMS [ WITH CONNECTION connection-name ] BEGIN
```

`WITH CONNECTION` *connection-name*	Specify the connection for this command. Because Informix does not support multiple connections, the `WITH CONNECTION` clause is necessary only in applications using more than one engine.

A transaction is a logical unit of work on a database. In Informix, transaction behavior differs for ANSI and non-ANSI databases.

For non-ANSI Informix databases, a transaction is contained within `DBMS BEGIN` and `DBMS COMMIT` statements. `DBMS BEGIN` defines the start of a transaction. After a transaction is begun, changes to the database are not committed until a `DBMS COMMIT` is executed. Changes are undone by executing `DBMS ROLLBACK`. Before beginning a new transaction, the application should `COMMIT` or `ROLLBACK` any pending work. Otherwise, you might receive an error.

For ANSI Informix databases, all statements up to a `DBMS COMMIT` are contained within a transaction. `DBMS BEGIN` has no effect. Changes can be undone by executing `DBMS ROLLBACK`.

Example

Refer to the example in Using Transactions on page 18.

See Also

Using Transactions on page 18

`COMMIT`

`ROLLBACK`

BUFFER_DEFAULT

Specifies setting for engine-based non-sequential scrolling

```
DBMS [WITH CONNECTION connection-name] BUFFER_DEFAULT value
```

`0`	Disable Informix-based scrolling on all cursors on the specified connection.
`1`	Enable Informix-based scrolling on all cursors on the specified connection.

Description

Informix supports sequential and scroll cursors. By default, Prolifics creates Informix sequential cursors.

An Informix sequential cursor can fetch only the next row in sequence from the select set. The sequential cursor can read through the active set once; to reread the rows, the application must re-execute the cursor.

An Informix scroll cursor allows an application to fetch rows in any sequence. The scroll cursor can re-fetch rows without re-executing the cursor.

A Prolifics application can use either Prolifics-based or Informix-based scrolling to execute `DBMS CONTINUE`, `DBMS CONTINUE_TOP`, `DBMS CONTINUE_UP`, and `DBMS CONTINUE_BOTTOM`.

To enable Prolifics-based scrolling an application executes `DBMS STORE FILE` for a specified cursor. To enable Informix-based scrolling an application executes `DBMS SET_BUFFER` for a specified cursor or `DBMS BUFFER_DEFAULT` for all cursors on an Informix connection.

To support Informix–based scrolling, Informix buffers the select rows in a temporary table. You might want to change the cursor's isolation level to prevent other users from modifying the rows when using Informix–based scrolling. See your Informix documentation for more information.

See Also

`SET_BUFFER`

COMMIT

Commit a transaction

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

`WITH CONNECTION` *connection-name*	Specify the connection for this command. This clause is necessary only in applications using more than one engine because Informix does not support multiple connections.

Description

Use this command to commit a pending transaction. Committing a transaction saves all the work since the last `COMMIT`. 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 new transaction, the application should `COMMIT` or `ROLLBACK` any pending transactions. Otherwise, you will receive an error.

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 18.

See Also

Using Transactions on page 18

`BEGIN`

`ROLLBACK`

ROLLBACK

Roll back a transaction

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

`WITH CONNECTION` *connection-name*	This clause is necessary only in applications using more than one engine because Informix does not support multiple connections.

Description

Use this command to rollback a transaction and restore the database to its state prior to the start of the transaction.

Example

Refer to the example in Using Transactions on page 18.

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 receive an error when it starts the next transaction.

Prolifics's database driver for Informix issues a `DBMS ROLLBACK` before closing a connection.

See Also

Using Transactions on page 18

`BEGIN`

`COMMIT`

SET_BUFFER

Use engine-based scrolling

```
DBMS [WITH CURSOR cursor-name ] SET_BUFFER 1

DBMS [WITH CURSOR cursor-name ] SET_BUFFER 0
```

`WITH CURSOR` *cursor-name*	Specify a named cursor for the command. If this clause is not included, Prolifics issues the command on the default cursor of the default connection.

Description

There are two methods of using the non-sequential scrolling commands `DBMS CONTINUE_BOTTOM`, `DBMS CONTINUE_TOP`, and `DBMS CONTINUE_UP`. In one method, an application uses Prolifics-based scrolling by setting up a continuation file with `DBMS STORE FILE`. In the other method, an application uses Informix-based scrolling by setting a flag for a cursor with `DBMS SET_BUFFER`.

By default, Prolifics declares Informix cursors without sequential scrolling. Use this command to allow a `SELECT` cursor to use Informix-based scrolling.

The argument for this command sets the availability of the scrolling. To turn on Informix-based scrolling, use this command:

```
DBMS [WITH CURSOR cursor-name ] SET_BUFFER 1
```

To turn off Informix-based scrolling, use this command:

```
DBMS [WITH CURSOR cursor-name ] SET_BUFFER 0
```

If the `WITH CURSOR` clause is used, Prolifics sets the flag for the named cursor. If the `WITH CURSOR` clause is not used, Prolifics sets the flag for the default `SELECT` cursor.

Note the following restrictions:

- When Informix-based scrolling is used, Informix prohibits the cursor from using some features, such as `SELECT FOR UPDATE`.
- Only a few engines support native scrolling. Therefore, this command might not be supported with other engines. Prolifics-based scrolling is supported on all engines with `DBMS STORE FILE`.

- Each `DBMS CONTINUE_BOTTOM`, `DBMS CONTINUE_TOP`, and `DBMS CONTINUE_UP` requires a trip to the server. With Prolifics-based scrolling, the rows are fetched once. When the application attempts to view rows already fetched, Prolifics reads them from the continuation file rather than requesting them from the server.

Example

```
DBMS DECLARE t_cursor CURSOR FOR SELECT * FROM titles
DBMS WITH CURSOR t_cursor SET_BUFFER 1

proc scroll_up
DBMS WITH CURSOR t_cursor CONTINUE_UP
return

proc scroll_down
DBMS WITH CURSOR t_cursor CONTINUE_DOWN
return
```

See Also

`CONTINUE_BOTTOM`

`CONTINUE_TOP`

`CONTINUE_UP`

`STORE`

SET HOLD

Set the HOLD behavior for a cursor

```
DBMS [ WITH CURSOR cursor-name ] SET HOLD { OFF | ON }
```

`WITH CURSOR` *cursor-name*	Specify a named cursor for the command. If this clause is not included, Prolifics issues the command on the default cursor of the default connection.

Description

Non-hold cursors in Informix are closed at the end of a transaction, even if the cursor only executed `SELECT` statements. Hold cursors remain open and keep their position even if other cursors execute and commit `UPDATE`, `INSERT` and `DELETE` statements.

In the current release, Prolifics for Informix declares all cursors to be hold cursors.

If `DBMS SET HOLD OFF` is issued for the default `SELECT` cursor, all subsequent SQL `SELECT` statements are on non-hold cursors. Therefore, after a transaction is committed or rolled back, positioning information for a select set is no longer available, and the `SELECT` statement needs to be re-executed. To reset the default behavior, issue `DBMS SET HOLD ON`.

If `DBMS SET HOLD OFF` is issued for a named cursor, it is a non-hold cursor throughout all subsequent executions and redeclarations of the cursor. To reset the default behavior, issue `DBMS WITH CURSOR` *cursor-name* `SET HOLD ON`.

Example

```
proc select_titles
DBMS DECLARE t_cursor CURSOR FOR \
   SELECT title_id, name, genre_code FROM titles
DBMS WITH CURSOR t_cursor SET HOLD OFF
DBMS WITH CURSOR t_cursor EXECUTE
```

SET HOLD_DEFAULT

Set the connection's default behavior for HOLD cursors

```
DBMS SET HOLD_DEFAULT { OFF | ON }
```

Description

Non-hold cursors in Informix are closed at the end of a transaction, even if the cursor only executed `SELECT` statements. Hold cursors remain open and keep their position even if other cursors execute and commit `UPDATE`, `INSERT` and `DELETE` statements.

In the current release, Prolifics for Informix declares all connections to create `SELECT` cursors as hold cursors.

If `DBMS SET HOLD_DEFAULT OFF` is issued for a connection, all subsequent SQL `SELECT` statements are on non-hold cursors. Therefore, after a transaction is committed or rolled back, positioning information for a select set is no longer available, and the `SELECT` statement needs to be re-executed. To reset the default behavior, issue `DBMS SET HOLD_DEFAULT ON`.

Example

```
proc connect_nonhold
DBMS DECLARE non_conn CONNECTION FOR \
   DATABASE ”videobiz”
DBMS WITH CONNECTION non_conn SET HOLD_DEFAULT OFF
DBMS CONNECTION non_conn
DBMS SQL SELECT title_id, name, genre_code FROM titles
```

Command Directory for Informix

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

Table 3. Commands for Informix

Command Name	Description	Documentation Location
ALIAS	Name a Prolifics variable as the destination of a selected column or aggregate function	*Programming Guide*
BEGIN	Begin a transaction	page 24
BINARY	Create a Prolifics variable for fetching binary values	page 810
BUFFER_DEFAULT	Set engine-based scrolling	page 25
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*
`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 27
`SET_BUFFER`	Set engine-based scrolling for a cursor	page 28
`SET HOLD`	Set behavior for `SELECT` cursors	page 30

Command Name	Description	Documentation Location
SET HOLD_DEFAULT	Set SELECT cursor behavior for the connection	page 31
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	