Panther

Database Driver–Sybase CT Library

Release 4.25

Prolifics®

May 2000

Database Driver for SYBASE CT Library

The SYBASE Open Client product provides software for communicating with SYBASE SQL Server and SYBASE Open Server. Open Client has two components: programming interfaces and network services. Prolifics for SYBASE is written using the programming interfaces of Open Client.

SYBASE has two programming interfaces, DB-Library and Client-Library. Prolifics provides a version of its support routine for each programming interface. You choose one of the programming interfaces when you you install the Prolifics/SYBASE product on Windows or when you edit the Prolifics/SYBASE `makevars` file on any platform.

In most cases you will notice no difference between Prolifics applications using DB-Library and those using Client-Library. However, some advanced features might be available in only one interface. DB-Library is recommended for applications using complicated stored procedures, remote procedure calls, or two–phase commits.

Client-Library is recommended for applications requiring SYBASE 10 native cursor support. DB-Library does not have native cursor support; Prolifics uses SYBASE dbprocesses to simulate cursor support with DB-Library. Unlike a dbprocess, a native cursor allows an application to select data and update rows in the select set without risking a deadlock. This problem can be avoided in DB-Library applications but Client-Library's native cursors are recommended for applications selecting 500 or more rows for update.

This chapter provides documentation specific to SYBASE using CT Library. 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 SYBASE, this results in the following `vendor_list` structure in `dbiinit.c`:

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

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

The settings are as follows:

`sybase`	Engine name. May be changed.
`dm_sybsup`	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 SYBASE, the settings can be changed by editing the `make-vars.syb` file.

Engine Name

You can change the engine name associated with the support routine `dm_sybsup`. 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.syb` file:

```
SYB_ENGNAME=tracking
```

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

`DBMS ENGINE` *sybase-engine-name*

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

```
DBMS ENGINE sybase
```

or

```
DBMS ENGINE tracking
```

Support Routine Name

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

SYBASE is case-sensitive. SYBASE uses the exact case of a SQL statement when creating database objects like tables and columns. In subsequent SQL statements, you must use the same exact case when referring to these objects. The default setting for case-sensitive engines is `DM_PRESERVE_CASE`. This means that the SYBASE column name is matched to a Prolifics variable with the same name and case when processing `SELECT` results.

The case setting can be changed. You can force Prolifics to perform case-insensitive searches. Substitute the `l` option in the `makevars` file to match SYBASE column names to lower case Prolifics variables, or use the `u` option to match to upper case Prolifics variables.

```
SYB_INIT=l
```

or

```
SYB_INIT=u
```

If you edit `makevars.syb`, 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

SYBASE 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.

Each Prolifics connection has its own SYBASE Client-Library context structure and connection structure.

The following options are supported for connections to SYBASE:

Table 1. Database connection options.

Option	Argument
`USER`	*user-name*
`INTERFACES`	*interfaces-file-pathname*
`SERVER`	*server-name*
`DATABASE`	*database-name*
`PASSWORD`	*password*
`APPLICATION`	*application-name*
`CHARSET`	*character-set-name*
`CURSORS`	ignored with CT-Library
`TIMEOUT`	*seconds*
`HOST`	*host-name*
`SQLTIMEOUT`	*seconds*

```
DBMS [ WITH ENGINE engine ] DECLARE connection CONNECTION \
   [ FOR [ USER user-name ] [ PASSWORD password ] \
   [ DATABASE database ] [ SERVER server ] \
   [ APPLICATION application-name ] \
   [ HOST host-name ] [ INTERFACES interface-file-pathname ] \
   [ SQLTIMEOUT seconds ] [ TIMEOUT seconds ] [ CHARSET character-set ] ]
```

For example:

```
DBMS DECLARE dbi_session CONNECTION FOR \
   USER ":uname" PASSWORD ":pword" DATABASE "sales" \
   SERVER "sybase10" APPLICATION "sales" HOST "oak" \
   INTERFACES "/usr/sybase/interfaces.app" \
   SQLTIMEOUT "120" TIMEOUT "15"
```

Additional keywords are available for other database engines. If those keywords are included in your `DBMS DECLARE CONNECTION` command for SYBASE, 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 SYBASE database. When the import process is complete, each selected database table has a corresponding repository entry screen.

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

- database tables
- database views

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 or database table view.
- 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 or unique indexes 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 SYBASE data type.

Table 2. Importing Database Tables

SYBASE Data Type	Code	Prolifics Type	C Type	Widget Length	Widget Precision
binary	45	DT_BINARY	Hex Dec	column length * 2	
bit	50	FT_INT	Int	1	
char	47	FT_CHAR	Char String	column length	
datetime	61	DT_DATETIME	Default	17	
decimal	55				
scale > 0		FT_FLOAT	Float	column precision + column scale + 1	column scale
else		FT_LONG	Long Int	column precision	
double precision	62	FT_FLOAT	Float	16	2
float	62	FT_FLOAT	Float	16	2
image	34	DT_BINARY	Hex Dec	column length	
int	56	FT_LONG	Long Int	11	
money	60	DT_CURRENCY	Default	26	
nchar	47	FT_CHAR	Char String	column length	
nvarchar	47	FT_CHAR	Char String	column length	
numeric	63				
scale > 0		FT_FLOAT	Float	column precision + column scale + 1	column scale
else		FT_LONG	Long Int	column precision	
real	59	FT_FLOAT	Float	16	2
smalldatetime	58	DT_DATETIME	Default	17	

SYBASE Data Type	Code	Prolifics Type	C Type	Widget Length	Widget Precision
smallint	52	FT_INT	Int	6	
smallmoney	122	DT_CURRENCY	Default	14	
text	35	FT_CHAR	Char String	254	
timestamp	80	DT_BINARY	Hex Dec	column length	
tinyint	48	FT_INT	Int	3	
varbinary	37	DT_BINARY	Hex Dec	column length * 2	
varchar	39	FT_CHAR	Char String	column length	

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.

The Use In Update property is also set to No for certain data types. In SYBASE, this applies to the data types text, image, and for any numeric column that is defined as identity.

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

Prolifics uses SYBASE's `convert` function and the SYBASE format string, `yyyymmdd hh:mm:ss` to convert a Prolifics date-time format to a SYBASE format.

In order for conversion to take place, the widget must have the C Type set to Default and the Format/Display⇒Data Formatting property set to Date/Time. Any date-time Format Type is appropriate.

This is the format for literal dates. It is compatible with SYBASE national language support.

Formatting Currency Values

SYBASE requires a leading dollar sign for values inserted in a `money` column in order to ensure precision. Prolifics will use a leading dollar sign when it formats widgets with a Prolifics type of `DT_CURRENCY`. Any other amount formatting characters are stripped. Therefore, if a currency field contained

```
500,000.00
```

Prolifics would format it as

```
$500000.00
```

Using Text and Image Data Types

Note that when the select list includes the values of text and image data types, the limit on the length of the data returned depends on the server setting of `textsize`. The SYBASE server default is 32K; however, this value can be changed on the server via the SYBASE `set` command. The global variable `@@textsize` contains the current maximum.

Declaring Cursors

Each cursor in Prolifics for SYBASE has its own Client-Library command structure whose parent is the connection structure associated with Prolifics's connection.

Prolifics Cursor	SYBASE Default Representation	Sample JPL
default select	native cursor	`DBMS SQL SELECT ...`

default non-select	command structure	`DBMS SQL INSERT ...`
		`DBMS SQL UPDATE ...`
		`DBMS SQL DELETE ...`

named	native cursor	`DBMS DECLARE` *cursor* `CURSOR`

You can change the SYBASE representation of a Prolifics cursor if necessary. For more information, refer to the following section.

The following SQL operations are not available in this version of Prolifics for SYBASE Client-Library:

- Browse mode
- `SELECT` statements containing a `COMPUTE` clause
- `UPDATE` statements containing a `WHERE CURRENT OF` clause
- `DELETE` statements containing a `WHERE CURRENT OF` clause
- Stored procedures using remote procedure calls (rpc)
- Output parameters and return codes from stored procedures

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

Setting Cursor Options

You can specify which type of Client-Library structure is to be used for SQL statements with the following `SET` commands:

- `SET RUN CT_CURSOR` — Force a particular Prolifics cursor to be run on a Client-Library cursor.

- `SET RUN CT_COMMAND` — Force a particular Prolifics cursor to be run on a Client-Library command structure.
- `SET RUN_DEFAULT CT_CURSOR` — Force all Prolifics cursors on a connection to be run as Client-Library cursors.
- `SET RUN_DEFAULT CT_COMMAND` — Force all Prolifics cursors on a connection to be run as Client-Library command structures.

More than one Client-Library cursor can be active per connection.

However, a Client-Library cursor can only be created for a Transact-SQL command batch that either contains a single `SELECT` statement or calls a stored procedure that contains only a single `SELECT` statement. A command batch that contains more than a single `SELECT` statement or that calls a stored procedure containing more than a single `SELECT` statement must run on a Client-Library command structure. However, the results from a command structure must be processed in their entirety before any other cursor or command structure on a connection can process its results.

For example, a SQL command batch containing two `SELECT` statements must be run on a Client-Library command structure resulting in the following JPL procedure:

```
proc select2
DBMS SET RUN CT_COMMAND
DBMS SQL SELECT xx, xx FROM pubs2..xxx SELECT xx, xx \
   FROM pubs2..xxx
```

In this example, executing `DBMS SET RUN CT_COMMAND` sets the default cursor in Prolifics to run on a Client-Library command structure so that the `SELECT` statement can execute without error.

For more information on the behavior of Client-Library cursors and command structures, refer to your SYBASE documentation.

Scrolling

Even though SYBASE Client-Library 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 SYBASE.

Errors

Prolifics initializes the following global variables for error code information:

`@dmretcode`	Standard database driver status code.
`@dmretmsg`	Standard database driver status message.
`@dmengerrcode`	SYBASE error code.
`@dmengerrmsg`	SYBASE error message.

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

In Prolifics for SYBASE Client-Library, `@dmengerrcode` and `@dmengerrmsg` can be arrays containing both client and server information. If both members of the array contain data, the error message from the client operation is in the first occurrence and the error message from the server operation is in the second occurrence. If only one occurrence has data, it can be either from the client or server operation.

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” \
   ”SYBASE error is %N” \
   @dmengerrcode[1] ” ” @dmengerrmsg[1] ”%N”\
   @dmengerrcode[2] ” ” @dmengerrmsg[2]
return 1
```

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

Using Stored Procedures

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, `EXEC`. For example:

```
DBMS SQL [ DECLARE parameter data-type \
   [ DECLARE parameter data-type ... ] ] \
   EXEC procedure-name [ parameter [, parameter ...] ]
```

An application can also use a named cursor to execute a stored procedure:

```
DBMS DECLARE cursor CURSOR FOR \
   [ DECLARE parameter data-type [ DECLARE parameter data-type ... ] ] \
   EXEC procedure-name [ parameter [, parameter ...] ]
```

The cursor can then be executed with the following statement:

```
DBMS [ WITH CURSOR cursor ] EXECUTE [ USING values ]
```

Output parameters and return codes are not supported for stored procedures in this release of Prolifics for SYBASE Client-Library.

Example

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

```
create proc update_tapes @parm1 int, @parm2 int
as
update tapes set status = '0'
   where title_id = @parm1 and copy_num = @parm2
```

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 EXEC update_tapes :+title_id, :+copy_num

DBMS DECLARE x CURSOR FOR EXEC 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.

Controlling the Execution of a Stored Procedure

Prolifics's database driver for SYBASE provides a command for controlling the execution of a stored procedure that contains more than one `SELECT` statement. The command is:

```
DBMS [ WITH CURSOR cursor ] SET behavior
```

behavior can have one of these values:

`STOP_AT_FETCH`

`EXECUTE_ALL`

If *behavior* is `STOP_AT_FETCH`, Prolifics stops each time it executes a non-scalar `SELECT` statement in the stored procedure. Therefore, a `SELECT` from a table will halt the execution of the procedure. However, a `SELECT` of a single scalar value (i.e., using the SQL functions `SUM`, `COUNT`, `AVG`, `MAX`. or `MIN`) does not halt the execution of a stored procedure.

The application can execute

`DBMS [WITH CURSOR` *cursor* `] CONTINUE`

or any of the `CONTINUE` variants to scroll through the selected records. To abort the fetching of any remaining rows in the select set, the application can execute

`DBMS [WITH CURSOR` *cursor* `] FLUSH`

To execute the next statement in the procedure the application must execute

`DBMS [WITH CURSOR` *cursor* `] NEXT`

`DBMS NEXT` automatically flushes any pending `SELECT` rows.

To abort the execution of any remaining statements in the stored procedure or the `sql` statement, the application can execute

`DBMS [WITH CURSOR` *cursor* `] CANCEL`

All pending statements are aborted. Canceling the procedure also returns the procedure's return status code. The return code `DM_END_OF_PROC` signals the end of the stored procedure.

If *behavior* is `EXECUTE_ALL`, Prolifics executes all statements in the stored procedure without halting. If the procedure selects rows, Prolifics returns as many rows as can be held by the destination variables and continues executing the procedure. The application cannot use the `DBMS CONTINUE` commands to scroll through the procedure's select sets.

Note that SYBASE does not support `SINGLE_STEP` as an option for stored procedure execution; however, it is available for execution of multi-statement cursors.

Using Transactions

A transaction is a unit of work that must be totally completed or not completed at all. SYBASE has one transaction for each cursor. Therefore, in a Prolifics

application, a transaction controls all statements executed with a single named cursor or the default cursor.

The following events commit a transaction on SYBASE:

- Executing `DBMS COMMIT`.
- Executing a data definition command such as `CREATE`, `DROP`, `RENAME`, or `ALTER`.

The following events roll back a transaction on SYBASE:

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

Transaction Control on a Single Cursor

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

SYBASE supports the following transaction commands:

- Begin a transaction on a default or named cursor.

  `DBMS [WITH CONNECTION connection ] BEGIN`

- Commit the transaction on a default or named cursor.

  `DBMS [WITH CONNECTION connection ] COMMIT`

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

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

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

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

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 SYBASE

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 SYBASE, the name of the standard transaction model is tmsyb1.c.

The standard transaction model for SYBASE calls DBMS FLUSH instead of DBMS CANCEL as part of the processing for the FINISH command. If a query has returned a very large select set, closing the screen might be longer with the FLUSH command. You can change this behavior by editing the model; however, the model is subject to change in future releases, so you should track your changes in order to update future versions.

Using Version Columns

For a SYBASE timestamp column, you can set the In Update Where and In Delete Where properties to Yes. This includes the value fetched to that widget in the SQL UPDATE and DELETE statements that are generated as part of the SAVE command.

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.

SYBASE-Specific Commands

Prolifics for SYBASE provides commands for SYBASE-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 RUN	Specify whether a cursor or command structure is used to execute SQL statements.

Using Stored Procedures

CANCEL	Abort execution of a stored procedure.
FLUSH	Abort execution of a stored procedure.
NEXT	Execute the next statement in a stored procedure.
SET	Set execution behavior for a procedure (execute all, stop at fetch, etc.).

Using Transactions

BEGIN	Begin a transaction.
COMMIT	Commit a transaction.
ROLLBACK	Rollback a transaction.
SAVE	Set a savepoint in a transaction.

BEGIN

Start a transaction

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

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

A transaction is a logical unit of work on a database 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`.

Example

Refer to the example in Using Transactions on page 17.

See Also

Using Transactions on page 17

`COMMIT`

`ROLLBACK`

`SAVE`

CANCEL

Cancel the execution of a stored procedure or discard select rows

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

`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

If the named cursor is a native cursor, this command closes the cursor. If the named cursor is a command structure, this command cancels any outstanding work on the named cursor. In particular, this command can be used to cancel a pending stored procedure or discard unwanted select rows. When the statement is executed, the following operations are performed:

- Any rows to be fetched are discarded.
- Any remaining unexecuted statements are ignored.

Prolifics calls the SYBASE routine `ct_cancel()` with the `CS_CANCEL_ALL` flag to perform this operation.

If the `WITH CURSOR` clause is not used, Prolifics executes the command on the default cursor.

See Also

Using Stored Procedures on page 15

`FLUSH`

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.

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.

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 17.

See Also

Using Transactions on page 17

`BEGIN`

`ROLLBACK`

`SAVE`

FLUSH

Flush any selected rows not fetched to Prolifics variables

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

`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

Use this command to throw away any unread rows in the select set of the default or named cursor. The named cursor can be a native cursor or a command structure.

This command is often useful in applications that execute a stored procedure. If the stored procedure executes a `SELECT`, the procedure will not return the `DM_END_OF_PROC` signal if the select set is pending. The application can execute `DBMS CONTINUE` until the `DM_NO_MORE_ROWS` signal is returned, or it can execute `DBMS FLUSH`, which discards the pending rows.

This command is also useful with queries that fetch very large select sets. The application can execute `DBMS FLUSH` after executing the `SELECT`, or after a defined time-out interval. This guarantees a release of the shared locks on all the tables involved in the fetch. Of course, after the rows have been flushed, the application cannot use `DBMS CONTINUE` to view the unread rows.

Prolifics calls the SYBASE routine `ct_cancel()` with the `CS_CANCEL_ALL` to perform this operation.

Example

```
proc large_select
# Do not allow the user to see any more rows than
# can be held by the onscreen arrays.
DBMS SQL SELECT * FROM titles
if @dmretcode != DM_NO_MORE_ROWS
   DBMS FLUSH
return 0
```

See Also

`DECLARE CURSOR`

`CANCEL`

`CONTINUE`

`NEXT`

NEXT

Execute the next statement in a stored procedure

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

`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

Unless `DBMS SET` equals `EXECUTE_ALL`, an application must execute `DBMS NEXT` after a stored procedure returns one or more `SELECT` rows to Prolifics. `DBMS NEXT` executes the next statement in the stored procedure. If the application executes `DBMS NEXT` and there are no more statements to execute, Prolifics returns the `DM_END_OF_PROC` code.

If a cursor is associated with two or more SQL statements and `DBMS SET` equals `STOP_AT_FETCH`, the application must execute `DBMS NEXT` after each `SELECT` that returns rows to Prolifics. If `DBMS SET` equals `SINGLE_STEP`, the application must execute `DBMS NEXT` after each statement, including non-`SELECT` statements. If the application executes `DBMS NEXT` after all of the cursor's statements have been executed, Prolifics returns the `DM_END_OF_PROC` code.

Example

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

See Also

Using Stored Procedures on page 15

`DECLARE CURSOR`

`CANCEL`

`CONTINUE`

`FLUSH`

`SET [EXECUTE_ALL | SINGLE_STEP | STOP_AT_FETCH]`

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.

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 17.

See Also

Using Transactions on page 17

`BEGIN`

`COMMIT`

`SAVE`

SET

Set handling for a cursor that executes a stored procedure or multiple statements

```
DBMS [WITH CURSOR cursor-name ] SET EXECUTE_ALL

DBMS [WITH CURSOR cursor-name ] SET SINGLE_STEP

DBMS [WITH CURSOR cursor-name ] SET STOP_AT_FETCH
```

`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

This command controls the execution of a stored procedure or a cursor that contains multiple SQL statements. This command allows the following options:

`EXECUTE_ALL`

Specifies that the DBMS return control to Prolifics only when all statements have been executed or when an error occurs. If a SQL `SELECT` is executed, only the first pageful of rows is returned to Prolifics variables. This option can be set for a multi-statement or a stored procedure cursor.

`SINGLE_STEP`

Specifies that the DBMS return control to Prolifics after executing each statement belonging to the multi-statement cursor. After each `SELECT`, the user can press a function key to execute a `DBMS CONTINUE` and scroll the select set. To resume executing the cursor's statements, the application must execute `DBMS NEXT`. This option can be set for a multi-statement cursor. If this option is used with a stored procedure cursor, Prolifics uses the default setting `STOP_AT_FETCH`.

`STOP_AT_FETCH`

Specifies that the DBMS return control to Prolifics after executing a SQL `SELECT` that fetches rows. (Note that control is not returned for a `SELECT` that assigns a value to a local SYBASE parameter.) The application can use `DBMS CONTINUE` to scroll through the select set. To resume executing the cursor's statements or procedure, the application must execute `DBMS NEXT`. This option can be set for a multi-statement or a stored procedure cursor.

The default behavior for both stored procedure and multi-statement cursors is STOP_AT_FETCH. Executing DBMS SET with no arguments restores the default behavior.

Example

```
DBMS DECLARE x CURSOR FOR \
SELECT cust_id, first_name, last_name, member_status \
   FROM customers WHERE cust_id = ::cust_id \
INSERT INTO rentals (cust_id, title_id, copy_num, \
   rental_date, price) \
   VALUES (::cust_id, ::title_id, ::copy_num, \
   ::rental_date, ::price)

msg d_msg ”%KPF1 START %KPF2 SCROLL SELECT\
 %KPF3 EXECUTE NEXT STEP”

proc f1
# This function is called by the PF1 key.
DBMS WITH CURSOR x SET_BUFFER 10
DBMS WITH CURSOR x SET SINGLE_STEP
DBMS WITH CURSOR x EXECUTE USING cust_id, cust_id, \
   title_id, copy_num, rental_date, price
DBMS WITH CURSOR x SET
return

proc f2
# This function is called by the PF2 key.
DBMS WITH CURSOR x CONTINUE
if @dmretcode == DM_NO_MORE_ROWS
   msg emsg ”All rows displayed.”
return

proc f3
# This function is called by the PF3 key.
DBMS WITH CURSOR x NEXT
if @dmretcode == DM_END_OF_PROC
   msg emsg ”Done!”
return
```

See Also

Using Stored Procedures on page 15

CANCEL

CONTINUE

DECLARE CURSOR

DECLARE CURSOR FOR EXEC

FLUSH

NEXT

SET

Force a SQL statement to be run on a Client–Library cursor or command structure

```
DBMS [WITH CURSOR cursor-name] SET RUN CT_COMMAND
DBMS SET RUN_DEFAULT CT_COMMAND

DBMS [WITH CURSOR cursor-name] SET RUN CT_CURSOR
DBMS SET RUN_DEFAULT CT_CURSOR
```

`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

The `SET` command can specify whether SQL statements will run on a Client-Library cursor or command structure. By default, Prolifics cursors run on Client-Library cursors. This command allows the following options:

`RUN CT _COMMAND`
Specifies that any subsequent `DBMS` statements be run on a Client-Library command structure instead of a Client-Library cursor.

`RUN_DEFAULT CT_COMMAND`
Specifies that for any Prolifics cursors on subsequent `DBMS DECLARE CURSOR` statements, the Prolifics cursor will be created on a Client-Library command structure instead of a Client-Library cursor.

`RUN CT_CURSOR`
Specifies that any subsequent `DBMS` statements be run on a Client-Library cursor instead of a Client-Library command structure.

`RUN_DEFAULT CT_CURSOR`
Specifies that for any Prolifics cursors on subsequent `DBMS DECLARE CURSOR` statements, the Prolifics cursor will be created as a Client-Library cursor on top of a command structure.

By default, Prolifics uses `RUN_DEFAULT CT_CURSOR` for the default select cursor and any named cursors and `RUN CT_CURSOR` for the default non-select cursor.

Command Directory for SYBASE

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

Table 3. Commands for SYBASE

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 22
BINARY	Create a Prolifics variable for fetching binary values	page 810
CANCEL	Abort execution of a stored procedure	page 23
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 24
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	
FLUSH	Flush any selected rows	page 25
FORMAT	Format the results of a CAT-QUERY	
NEXT	Execute the next statement in a stored procedure	page 27
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 28

Command Name	Description	Documentation Location
`SET` *parameter*	Set execution behavior for a stored procedure	page 29
`SET RUN`	Set statement execution on a cursor or command structure	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	