TABLE OF CONTENTS

Contents

Panther

Programming Guide

Prolifics®

Release 5.51

Document 0404
March 2017

Send suggestions and comments regarding this document to:

Technical Publications Manager	http://prolifics.com
Prolifics, Inc.	support@prolifics.com
24025 Park Sorrento, Suite 405	(800) 458-3313
Calabasas, CA 91302	

Contents

2. JPL Command Reference

About This Document

Programming Guide is a reference tool for Panther users who already have a general understanding about Panther concepts and design techniques. This book offers general and specific information on how to use Panther language resources to code back-end processing for your application. The sections on JPL assume that you already have general programming experience; while the library function and Java descriptions assume specific experience with C and Java programming.

This manual is divided into the following sections:

- Descriptions of each JPL command.
- Descriptions of the preinstalled, or built-in, control functions that you can call from the application.
- Descriptions of Panther's library of C functions, which provide precise runtime control over your application.
- Descriptions of Panther's Java interfaces.
- Descriptions of the transaction manager commands and error messages.
- A database reference, including descriptions of DBMS statements and commands, global variables and keywords in database drivers.

Documentation Website

The Panther documentation website includes manuals in HTML and PDF formats and the Java API documentation in Javadoc format. The website enables you to search the HTML files for both the manuals and the Java API.

Panther product documentation is available on the Prolifics corporate website at http://docs.prolifics.com/panther/.

How to Print the Document

You can print a copy of this document from a web browser, one file at a time, by using the File→Print option on your web browser.

A PDF version of this document is available from the Panther library page of the documentation website. You can open the PDF in Adobe Acrobat Reader and print the entire document (or a portion of it) in book format.

If you do not have the Adobe Acrobat Reader, you can get it for free from the Adobe website at https://get.adobe.com/reader/otherversions/.

Documentation Conventions

The following documentation conventions are used throughout this document.

Convention	Item
Ctrl+Tab	Indicates that you must press two or more keys simultaneously. Initial capitalization indicates a physical key.
italics	Indicates emphasis or book titles.
UPPERCASE TEXT	Indicates Panther logical keys. *Example*: XMIT
boldface text	Indicates terms defined in the glossary.

`monospace text`	Indicates code samples, commands and their options, directories, and file names and their extensions. Monospace text also indicates text that you must enter from the keyboard.
	Examples:
	`#include <smdefs.h>`
	`chmod u+w *`
	`/usr/prolifics`
	`prolifics.ini`

`monospace italic text`	Identifies variables in code representing the information you supply. *Example*: `String` *`expr`*

`MONOSPACE UPPERCASE TEXT`	Indicates environment variables, logical operators, SQL keywords, mnemonics, or Panther constants.
	Examples:
	`CLASSPATH`
	`OR`

`{ }`	Indicates a set of choices in a syntax line. One of the items should be selected. The braces themselves should never be typed.
`|`	Separates mutually exclusive choices in a syntax line. The symbol itself should never be typed.

Convention	Item
`[ ]`	Indicates optional items in a syntax line. The brackets themselves should never be typed. *Example*: `formlib [-v]` *`library-name`* `[`*`file-list`*`]...`
`...`	Indicates one of the following in a command line: ■ That an argument can be repeated several times in a command line ■ That the statement omits additional optional arguments ■ That you can enter additional parameters, values, or other information The ellipsis itself should never be typed. *Example*: `formlib [-v]` *`library-name`* `[`*`file-list`*`]...`
. . .	Indicates the omission of items from a code example or from a syntax line. The vertical ellipsis itself should never be typed.

Contact Us!

Your feedback on the Panther documentation is important to us. Send us e-mail at support@prolifics.com if you have questions or comments. In your e-mail message, please indicate that you are using the documentation for Panther 5.50.

If you have any questions about this version of Panther, or if you have problems installing and running Panther, contact Customer Support via:

- Email at support@prolifics.com
- Prolifics website at http://profapps.prolifics.com

When contacting Customer Support, be prepared to provide the following information:

- Your name, e-mail address and phone number

- Your company name and company address
- Your machine type
- The name and version of the product you are using
- A description of the problem and the content of pertinent error messages

CHAPTER

1 JPL Command Overview

Below is a summary of the JPL commands organized according to category. All JPL statements begin with one of these commands. For more information on each command, refer to Chapter 2, "JPL Command Reference."

Control Flow

Table 1-1 JPL Control Flow Commands

`break`	Exits a loop
`for`	Executes an indexed loop
`if...else if...else`	Conditionally executes statements
`next`	Skips to next iteration of loop
`return`	Exits a JPL procedure
`switch...case...default`	Execute different statements based on the value of an expression
`while`	Repeatedly executes statements while a condition is true

Procedure Structure

Table 1-2

`parms`	Declares parameters in an unnamed JPL procedure
`proc`	Begins a named procedure

Variable Declaration

Table 1-3

`global`	Declares global JPL variables
`vars`	Declares JPL variables

Command/Function Execution

Table 1-4

`call`	Executes an installed function or JPL procedure
`runreport`	Run a report

Module Access and Availability

Table 1-5

`include`	Includes contents of another module at current statement line
`public`	Reads a JPL module into memory and enables access to its named procedures
`unload`	Unloads modules loaded through the public command and releases their associated memory

Text Display

Table 1-6

`flush`	Flushes buffered output to the display
`msg`	Displays a message to the terminal

Data/Message Transfer

Table 1-7

`receive`	Receives data from a screen sent via send or from a remote client via `service_call`
`send`	Sends data to a buffer for retrieval by the receive command

Database Drivers

Table 1-8

`dbms`	Executes a command available in Panther's database drivers.

JetNet/Oracle Tuxedo Processing

Table 1-9

`jif_check`	Determines if the JIF has changed
`jif_read`	Rereads the JIF
`log`	Logs a message to the machine event log

Connection

Table 1-10

`client_exit`	Closes the middleware session
`client_init`	Attaches a client to the middleware

Data/Message Transfer

Table 1-11

`broadcast`	Sends a message to a client
`dequeue**`	Releases a message from a reliable queue

Table 1-11

`enqueue**`	Places a message on a reliable queue
`notify`	Sends an unsolicited message to a client
***Oracle Tuxedo only*	

Service Request Processing

Table 1-12

`advertise`	Advertises services offered by a server to a client
`service_call`	Initiates a service call from a client agent
`service_cancel`	Cancels an outstanding service request
`service_forward`	Forwards service request data to another service
`service_return`	Returns from a service request invocation
`unadvertise`	Unadvertises services from a server
`unload_data`	Writes data received remotely via the middleware to target Panther variables
`wait`	Waits for service calls to return before processing resumes

Event Broker Processing

Table 1-13

`post**`	Posts an event
`subscribe**`	Subscribes to an event
`unsubscribe**`	Unsubscribes from an event
***Oracle Tuxedo only*	

Two-Phase Commit Transaction Processing

Table 1-14

`xa_begin**`	Starts a middleware transaction
`xa_commit**`	Commits a middleware transaction
`xa_end**`	Completes a middleware transaction
`xa_rollback**`	Aborts a middleware transaction
***Oracle Tuxedo only*	

Component Processing (COM, EJB)

Table 1-15

`log`	Logs a message to server.log
`raise_exception`	Sends an error code back to the client
`receive_args`	Receives a method's parameters from the client
`return_args`	Returns the method's parameters to the client

CHAPTER

2 JPL Command Reference

This section lists JPL commands in alphabetical order. It serves as a reference for users who already have a working knowledge of JPL. Each command description tells you what the command does, and where and how to use it. Command descriptions are organized into the following components, as applicable:

- Command name and brief description.
- Syntax line and parameter descriptions.
- Unless stated otherwise, JPL command arguments can be variables or quoted strings.
- Environment-specificity; that is, if the command is dependent on a particular environment feature.
- Client and/or server applicability.
- Description of the command.
- Possible exceptions that can be raised due to the command's execution.
- Examples.
- Related commands.

advertise

Advertises services offered by a server to a client

Synopsis `advertise {ALL | SERVICE serviceName | GROUP serviceGroup}`

Arguments

`ALL`
: Advertise all services defined in the JIF.

`SERVICE serviceName`
: Advertise a service that is defined in the JIF, where *`serviceName`* can be up to 15 characters long. The JIF is consulted to validate the service name.

`GROUP serviceGroup`
: Advertise all services belonging to the named group, as defined in the JIF, where *`serviceGroup`* can be up to 31 characters long.

Environment JetNet, Oracle Tuxedo

Scope Server

Description The `advertise` command reads the specified services from the JIF and advertises them to clients. If a service is already advertised, its JIF definition is reread and the service is readvertised. This might occur during development or maintenance when a server is reinitialized after a JIF entry is updated or corrected. If successful, `advertise` sets the `tp_return` property to the number of services advertised.

You can advertise individual services, all services in a group, or all services. For example, the following code advertises service `transfer`:

```
advertise SERVICE "transfer"
```

This code advertises all services in the `emp_account` group:

```
advertise GROUP "emp_account"
```

The next example advertises all services and logs a message:

```
proc adv_and_log
vars message

advertise ALL
message = @app()->tp_return##" services advertised."
log message
```

Exceptions

Execution of `advertise` can raise the following exceptions:

Exception	**Severity**	**Cause**
`TP_GROUP_NOT_IN_JIF`	`TP_COMMAND`	Group cannot be found in the JIF.
`TP_IDENTIFIER_TRUNCATED`	`TP_WARNING`	Service name exceeded 15 characters.
`TP_MONITOR_ERROR`	`TP_ERROR`	An error was reported by the middleware.
`TP_NO_SERVICES_ADVERTISED`	`TP_WARNING`	`ALL` was specified and there are no services in the JIF, or `GROUP` was specified and the group is empty.
`TP_SERVICE_NOT_IN_JIF`	`TP_COMMAND`	Service cannot be found in the JIF.
`TP_SVC_ADVERTISE_LIMIT`	`TP_COMMAND`	The limit on number of advertised services has been reached.

See Also

`unadvertise`

break

Stops loop execution

Synopsis `break [`*`intConstant`*`]`

Arguments *`intConstant`*

The number of nested loops to stop, where a value of `1` specifies the current loop. If you omit this argument, `break` exits the current loop.

Description The `break` command stops execution of the current `while` or `for` loop. If the current loop is nested inside one or more other loops, and *`intConstant`* is greater than `1`, `break` stops execution of the specified number of outer loops. If *`intConstant`* is greater than or equal to the number of loops currently being executed, JPL stops each loop until it exits the outermost one.

Example

```
// Concatenate address and execute function for 100 entries.
// If cities[i] is empty stop executing the loop.
//

vars i, address, total
for i = 1 while i <= 100
{
    if cities[i] == ""
        break
    address = cities[i]##", "##states[i]##"  "##zips[i]
    call do_process (address)
}
total = i - 1
msg emsg "Done!  :total addresses processed."
```

See Also `for`, `next`, `while`

broadcast

Broadcasts a message to a client

Synopsis

`broadcast [`*`broadcastOption`*`]... TYPE`*`msgType`*` (`*`message`*`)`

Arguments

`broadcastOption`

One or more of the following options:

`CLIENT [`*`clientName`*`]`

Identifies the client to receive the message, where *`clientName`* can be up to 30 characters long. A client's name is set by `client_init`, which establishes the client connection. If you omit *`clientName`*, all clients receive the message.

`LMID` *`lmid`*

Specifies the logical ID of a machine to get the broadcast, where *`lmid`* can be up to 30 characters long. (Oracle Tuxedo only)

`NOTIMEOUT`

Disregards the blocking timeout. Transaction timeouts remain in effect.

`USER [`*`userName`*`]`

Identifies the user to receive the message, where *`userName`* can be up to 30 characters long. A client's user name is set by `client_init`.

If *`userName`* is not found on the system, this option is ignored and no error is raised. If you omit *`userName`*, all users receive the message.

`TYPE` *`msgType`*

Specifies the message's data type, where *`msgType`* is one of these values:

- `JAMFLEX`
- `STRING` (Oracle Tuxedo only)
- `FML` (Oracle Tuxedo only)
- `FML32` (Oracle Tuxedo only)

For more information on message data types, refer to “Service Messages and Data Types” on page 5-15 in *JetNet/Oracle Tuxedo Guide*.

`message`
: The message to broadcast. The message data must conform to the `<TYPE>`-specified data type.

Environment JetNet, Oracle Tuxedo

Scope Client, Server

Description The `broadcast` command is used by the middleware to broadcast a message to all clients that match the criteria specified in *`broadcastOption`*. Clients and servers can broadcast a message to other clients. If no options are specified, the message is broadcast to all clients.

For example, the following command broadcasts a `JAMFLEX`-type message to client supervisor. It uses source to identify itself as the source of the message:

```
broadcast CLIENT "supervisor" TYPE JAMFLEX \
    ({source="broadcast_security", ACCOUNT=acct, DATE=date,\
     SECURITY=code, MSG=message})
```

Messages delivered via broadcast are unsolicited. In order for unsolicited messages to be interpreted correctly by agents receiving them, a message handler must be installed. Because the handler is unaware of a message's origin, it is important that a standard method of identifying the source of unsolicited messages be established for the entire application. For more information on writing a message handler for your application, refer to “Message Handlers” on page 6-17 in *JetNet/Oracle Tuxedo Guide*.

Exceptions Execution of the `broadcast` command can raise the following exceptions:

Exception	Severity	Cause
`TP_IDENTIFIER_TRUNCATED`	`TP_WARNING`	*`clientName`*, *`lmid`*, or *`userName`* exceeds 30 characters.
`TP_INVALID_ARGUMENT_LIST`	`TP_COMMAND`	More than one argument is passed to message.
`TP_MONITOR_ERROR`	`TP_COMMAND`	Error reported from middleware.
`TP_TIMEOUT`	`TP_COMMAND`	Timeout condition occurs.

Example

```
// get the option menu choice and
// broadcast message accordingly

proc brdcast_to()
if opt_mnu == "All"
{
   broadcast (message)
}
else if opt_mnu == "Customers"
{
   broadcast USER "Customer" (message)
}
else if opt_mnu == "Employees"
{
   broadcast USER "Employee" (message)
}
else if opt_mnu == "Select Customer"
{
   send BUNDLE "scr_title" DATA opt_menu
   call sm_jwindow(&get_name)
   receive BUNDLE "name" DATA cust_last_name
   broadcast USER "Customer" CLIENT cust_last_name (message)
}
else if opt_mnu == "Select Employee"
{
   send BUNDLE "scr_title" DATA opt_menu
   call sm_jwindow(&get_name)
   receive BUNDLE "name" DATA emp_last_name
   broadcast CLIENT emp_last_name (message)
}

return 0
```

See Also `client_init`, `notify`, `receive`

call

Executes an installed function or JPL procedure

Synopsis `call executable[([argList])]`

Arguments

`executable`

The name of an installed function or JPL module or procedure.

Refer to “Precedence of Called Objects” on page 19-24 in *Application Development Guide* for more information on how Panther resolves this argument.

`argList`

One or more comma- or space-delimited arguments optionally to pass to parameters in executable. Enclose the entire argument list in parentheses. You can pass the following as arguments:

- Variables, including those declared by the `vars` command, field names, and LDB entries.
- String and numeric constants.
- Global constants.
- `@NULL` for any parameter in a C function that accepts `NULL` as an argument.
- Colon-expanded variables.

Description

The `call` command can call one of the following executables:

- Built-in and installed functions. Installed functions can include Panther library functions and your own functions.
- JPL modules and procedures.

When Panther gets a call command, it must ascertain whether the executable is a JPL module or procedure, or an installed function. Panther looks for executable's name first among all built-in and installed functions, then among JPL modules and procedures. Refer to “Precedence of Called Objects” on page 19-24 in *Application Development Guide* for more information on how Panther searches among JPL modules and procedures. If no match is found, Panther issues an error message.

Panther evaluates the call statement to its return value–either integer, string, or double, according to the procedure definition. Therefore, you can implicitly call a function within an expression and gets its return value as follows:

```
vars i
i = myproc (a,b)
```

Panther assumes that the executable has the same number and type of parameters. Panther passes arguments by value, so changes to the receiving parameter's value leave its corresponding caller's argument unchanged. If the executable is an installed function, you can pass it hex, binary or octal numbers.

You can install C functions so that arguments can be passed by value. Refer to "Installing Functions" on page 44-5 in *Application Development Guide* for information about installation options.

If you pass a variable's name, you can use Panther library functions to change the contents of the variable. For example, if you pass a field name to a prototyped function, the function can change the field's contents by using `sm_n_putfield`.

client_exit

Disconnects from the middleware

Synopsis

```
client_exit
```

Environment JetNet, Oracle Tuxedo

Scope Client

Description The `client_exit` command closes a connection between a client and the middleware.

When it closes a connection, Panther cleans up all resources associated with it. This includes aborting outstanding requests and rolling back incomplete transactions. It automatically terminates the connection to all resource managers.

`client_exit` sets the `tp_return` property to the number of connections actually closed.

If you have established a direct connection to an XA-compliant resource manager with the `GROUP` option of `client_init`, this command closes that connection.

Exceptions The `client_exit` command can raise the following exceptions:

Exception	Severity	Cause
`TP_INVALID_CONNECTION`	`TP_COMMAND`	No connection to close.
`TP_INVALID_SERVER_COMMAND`	`TP_COMMAND`	Server attempts to use the command.
`TP_MONITOR_ERROR`	`TP_ERROR`	Operating system error is detected.
`TP_XA_CLOSE_FAILED`	`TP_ERROR`	Unable to close XA-connection resource managers.

See Also `client_init`

client_init

Attaches a client to the middleware

Synopsis

```
client_init [connectionOpt]... [authLevel2] [authLevel3]
    [NOTIFICATION notifyMethod]
```

Arguments

`connectionOpt`

One or more options that supply information about the client:

`CLIENT` *`clientName`*

Specifies the name of the client, where *`clientName`* can be up to 30 characters long. If omitted, the default is an empty string.

`USER` *`userName`*

Specifies a user account (login) name, where *`userName`* can be up to 30 characters long. If omitted, the default is an empty string.

`GROUP` *`groupName`*

(Oracle Tuxedo only) Specifies the name of a group associated with this client, where *`groupName`* can be up to 15 characters long. If omitted, the default is an empty string.

You can set *`groupName`* to associate the client with a resource manager group that is defined in the configuration file. This allows the client to access an XA-compliant resource manager as part of a global transaction.

`authLevel2`

Supply the application password for client access with this syntax:

`PASSWORD` *`password`*

`password` specifies the application password, a string of up to 30 characters; however, only the first 8 characters are significant. If omitted, the default is an empty string. You set the application password through the JetNet manager; refer to “Application Password” on page 3-12 in *JetNet/Oracle Tuxedo Guide* for more information.

`authLevel3`

(Oracle Tuxedo only) Supply level-3 authentication data with one of these syntax clauses:

`DATA` *`password`*

Specifies the user-specific password required by the authentication service.

`DATAFUNC` *`dataFunc`* `[POSTFUNC` *`postFunc`*`]`

Specifies a function that provides level-3 authentication data and, optionally, a post-connection function that handles successful or failed connections. For the `DATAFUNC` function's prototype and description, refer to page 2-13, "Authentication Data Function"; for the `POSTFUNC` function, page 2-14, "Post-Connection Function."

Both functions must be installed. For information on installing a `DATAFUNC` function, refer to "Client Authentication Functions" on page 44-28 in *Application Development Guide*; for `POSTFUNC` functions, refer to "Client Post-Connection Functions" on page 44-30 in *Application Development Guide.*

`NOTIFICATION` *`notifyMethod`*

For use by administrator clients only: Specify how the client is notified of unsolicited messages, where *`notifyMethod`* has one of these values:

- `POLL`—Notify by polling for messages. The default polling interval is 10 seconds. To change the interval, set the `tp_unsol_poll_interval` property. If you are using Panther, this is the default. For Oracle Tuxedo applications, the default can be set in the `tuxconfig`.
- `SIGNAL`—Notify by signal. If this option is not available on a given platform, Panther uses `POLL` instead.
- `IGNORE`— All unsolicited messages are ignored.

For example, the following code opens a client connection and specifies to ignore all unsolicited messages:

```
client_init CLIENT "shipping" USER user NOTIFICATION IGNORE
```

This code specifies that the client be notified of unsolicited messages by polling:

```
client_init CLIENT last_name USER "Customer" \
       NOTIFICATION POLL
```

Environment JetNet, Oracle Tuxedo

Scope Client

Description

The `client_init` command opens a client connection to the application middleware–either JetNet or Oracle Tuxedo. Only one connection between client and middleware is allowed. Panther connects a workstation client to the middleware through its settings in configuration variables `SMRBHOST` and `SMRBPORT`, which specify the network addresses of one or more server machines; native clients use the settings in the configuration file specified by `SMRBCONFIG`.

Client Authentication

`client_init` offers several ways to authorize client access to an application. Level-2 authentication provides a single application-wide password; the `PASSWORD` option is required; the `USER` and `CLIENT` options are for informational purposes only and are not validated.

For example, this statement opens a client connection and specifies an application password:

```
client_init PASSWORD appPassword
```

Level-3 authentication, available with Oracle Tuxedo, offers user-specific validation as an additional layer of security. Level-3 authentication uses a Oracle Tuxedo service that validates the user name, client name, and user password, as supplied by the `USER`, `CLIENT`, and `DATA` options, respectively. The Oracle Tuxedo service can be configured to validate different combinations of this data.

For example, this command assumes validation at the application and user levels:

```
client_init USER username PASSWORD appPassword DATA userPassword
```

For more information about setting user-level security in Oracle Tuxedo applications, refer to "Security Administration" in the *Oracle Tuxedo Administrator's Guide*.

Authentication Data Function

Use the `DATAFUNC` option if the user password data is more complex than a simple string. On return, the `DATAFUNC`-specified C function supplies all data that is required by the authentication service, including user name, client name, and group name. For example, the following code opens a client connection for a named client, specifying a function that produces the authentication data:

```
client_init CLIENT "shipping" DATAFUNC "ship_authorize"
```

All `DATAFUNC` functions must be installed; for more information, refer to “Client Authentication Functions” on page 44-28 in *Application Development Guide*.

A `DATAFUNC` function must conform to this prototype:

```
long DATAFUNC (VOIDPTR *data,
               const char *usrname, const char *cltname,
               const char *passwd, cons char *grpname);
```

`data`
: Must be set by the `DATAFUNC` function to the address of the user data. This address must remain valid on return from the function.

`srname, cltname, passwd, grpname`
: Contain the values specified by the corresponding `USER`, `CLIENT`, `PASSWORD`, and `GROUP` options; otherwise `NULL`. These values must not be changed by the `DATAFUNC` function.

When successful, the `DATAFUNC` function returns the length of the data; a negative value indicates that an error occurred. If the function is successful, the output value of data should be the address of the user data. If *`dataFunc`* returns a negative value, or if the address in data is `NULL` when a positive value is returned, `client_init` raises a `TP_DATAFUNC_FAILED` exception of severity `TP_COMMAND`.

Post-Connection Function

Use the `POSTFUNC` option to pair a post-connection function with the `DATAFUNC` function. This C function is always called whether or not the connection is successful.

Panther installs a built-in `postFunc` `sm_tp_free`, which deallocates any memory allocated by the `dataFunc`.

All `POSTFUNC` functions must be installed; for more information, refer to “Client Post-Connection Functions” on page 44-30 in *Application Development Guide*.

A `POSTFUNC` function must conform to this prototype:

```
void POSTFUNC (VOIDPTR data, long datalen,
               const char *usrname, const char *cltname,
               const char *passwd, const char *grpname);
```

`data`
: Contains the address of the user data as set by the `DATAFUNC` function.

`datalen`
: Contains the length returned by the `DATAFUNC` function.

`usrname, cltname, passwd, grpname`
: Contain the values specified by the corresponding `USER`, `CLIENT`, `PASSWORD`, and `GROUP` options; otherwise `NULL`. These values must not be changed by the `POSTFUNC` function.

Exceptions

`client_init` can raise the following exceptions:

Exception	Severity	Cause
`TP_CONNECTION_LIMIT`	`TP_COMMAND`	A second connection to the middleware is attempted.
`TP_CONNECTION_OPEN_FAILED`	`TP_ERROR`	Connection to the middleware cannot be initiated.
`TP_DATAFUNC_FAILED`	`TP_COMMAND`	`DATAFUNC` function returned a negative value, or address in data is `NULL` when positive value is returned.
`TP_HANDLER_MISSING`	`TP_ERROR`	`DATAFUNC` and/or `POSTFUNC` options are specified but do not exist in the appropriate Panther function list.
`TP_INVALID_SERVER_COMMAND`	`TP_COMMAND`	Server attempts to use this command.
`TP_MONITOR_ERROR`	`TP_ERROR`	Operating system error is detected.
`TP_NO_SIGNALS`	`TP_INFORMATION`	Client is not capable of signal-based notification.
`TP_PERMISSION_DENIED`	`TP_ERROR`	The middleware does not accept connection.
`TP_XA_OPEN_FAILED`	`TP_ERROR`	Unable to open XA-connection resource managers.

See Also

`client_exit`

dbms

Executes a command available in Panther's database drivers

Synopsis `dbms dbmsStmt`

Arguments *`dbmsStmt`*

The command to execute, where `dbmsStmt` can include one of the following:

- SQL statements preceded by the keyword `RUN` or `QUERY`.
- Directives that are a part of Panther's database drivers—for example, fetch the next 10 rows.
- Directives that are not standardized across dialects of SQL, such as commit transaction.

Description The `dbms` command executes the specified command after colon expansion and syntax checking. These commands control the connections to database engines, process information fetched in `SELECT` statements, and update database information. For information on available commands, refer to Chapter 11, “DBMS Statements and Commands.”

There are three methods of executing SQL statements:

- `DBMS QUERY` and `DBMS RUN` pass the statement directly to the database engine.
- `DBMS DECLARE CURSOR` creates a named cursor to use for executing the SQL statement.

For more information, refer to Chapter 28, “Writing SQL Statements,” in *Application Development Guide*.

Because each database engine has unique features, refer to *Database Drivers* for information about database-specific features and commands.

Additional forms of colon expansion–colon plus processing and colon equal processing—are available with the `dbms` command to help format information before passing it to the database engine. For more information, refer to Chapter 29, “Reading Information from the Database,” in *Application Development Guide*.

Example

```
// Fetch next set of rows
dbms continue

// Commit transaction
dbms commit

// SQL statement
dbms QUERY select * FROM titles WHERE title_id = :+title_id
```

dequeue

Releases a message from a reliable queue

Synopsis

```
dequeue QSPACE queueSpace NAME queue message [which]
    [inputOption]... [outputOption]...
```

Arguments

`QSPACE` *`queueSpace`*

Names the queue space to which queue belongs.

`NAME` *`queue`*

Names the queue as defined in the JIF.

`message`

A message output argument to receive the dequeued message. The argument's format must conform to the message data type as specified in the queue's JIF definition. For more information on message data types, refer to "Service Messages and Data Types" on page 5-15 in *JetNet/Oracle Tuxedo Guide*.

`which`

Specifies which message to dequeue; use one of the following arguments. If no argument is supplied, the first message in the queue is removed.

`FIRST_AVAILABLE [WAIT]`

Remove the first message in the queue. This is the default behavior if no argument is supplied. If the queue is empty, the `WAIT` option specifies to wait until a message becomes available for dequeuing; otherwise, an error message is posted.

Note, you should only use `FIRST_AVAILABLE WAIT` if there is more than one `TMQUEUE` server running, otherwise the one server will wait indefinitely for a message that cannot be enqueued. For further information on Oracle Tuxedo-provided servers, consult your Oracle Tuxedo documentation.

`BY_MSGID` *`msgId`*

Dequeue the message corresponding to *`msgId`*. The message identifier is generated when the message is successfully enqueued.

`BY_CORRID` *`corrID`*

Dequeue the message corresponding to *`corrID`*. The correlation identifier is set by the application when enqueuing the message.

`inputOption`

Use one or more of the following options to control the behavior of `dequeue`:

`NOTIMEOUT`
: Specifies that the dequeue operation is unaffected by the blocking timeout; however, transaction timeouts remain in effect.

`OUTSIDE_TRANSACTION`
: Specifies to perform the dequeuing operation outside the current transaction. If message dequeuing fails, the current transaction is unaffected. If you specify this option, transaction-level exception and unload handlers are not executed when their corresponding events are generated.

`outputOption`

Use any the following keywords to set output arguments with information about the dequeued message:

`APPL_AUTH_KEY` *`key`*
: Returns the application authentication key associated with the client that enqueued the message.

`CLIENT` *`clientId`*
: Returns the client ID of the agent that originated the request.

`CORRID` *`corrID`*
: Returns the message's correlation ID, set by enqueue. For more information about using correlation IDs, refer to the enqueue command.

`FAILUREQ` *`queue`*
: Returns the name of the queue where a failure message should be stored. The value is set if the dequeued message is associated with a failure queue.

`MSGID` *`qMsgId`*
: Returns the unique message ID if set and the dequeue was successful. The identifier is generated when the message is successfully enqueued.

`PRIORITY` *`priority`*
: Returns the message's priority relative to other messages in the queue as an integer between 1 and 100, where 100 indicates the highest priority. A message with the highest number is dequeued before all others.

`RCODE` *`returnCode`*
: Returns the return code specified by enqueue when the message was enqueued.

`REPLYQ` *`queue`*
Returns the name of queue where the reply message should be stored. The value is set if the dequeued message is associated with a reply queue.

Environment Oracle Tuxedo

Scope Client, Server

Description The dequeue command removes a message from the specified queue. You can identify the message you want dequeued; otherwise, dequeue uses the first message. The order of messages in the queue is specified when they are enqueued. You can request a particular message for dequeuing by specifying its message identifier (`BY_MSGID` *`msgId`*) or correlation ID (`BY_CORRID` *`corrID`*). You can also indicate that the application wait for a message that is not immediately available.

When `dequeue` is successful, it can return additional information about the message:

- The message identifier for the dequeued message.
- A user-assigned correlation identifier that should accompany any reply or failure message. This allows the originator to correlate the message with the original request
- The name of a reply queue if a reply is desired
- The name of the failure queue on which the application can enqueue information regarding failure to process the message.

You can determine the success or failure of `dequeue` by checking the severity level that is set in the `tp_severity` property.

For more information about Oracle Tuxedo System /Q, refer to “Reliable Queues” on page 8-11 in *JetNet/Oracle Tuxedo Guide* and refer to your Oracle Tuxedo documentation.

Exceptions `dequeue` can generate the following exceptions:

Exception	Severity	Cause
`TP_INVALID_COMMAND_SYNTAX`	`TP_COMMAND`	Command syntax is invalid.

Exception	Severity	Cause
`TP_INVALID_VARIABLE_REF`	`TP_COMMAND or TP_WARNING`	Unable to resolve reference to Panther variable.
`TP_NO_OUTSIDE_TRANSACTION`	`TP_WARNING`	No transaction exists.
`TP_QUEUE_SPACE_NOT_IN_JIF`	`TP_COMMAND`	Queue space not found in the JIF.
`TP_QUEUE_NO_MSG`	`TP_ERROR`	No message was available for dequeuing.
`TP_TIMEOUT`	`TP_ERROR`	Message does not successfully dequeue within specified timeout.

See Also

enqueue

enqueue

Places a message on a reliable queue

Synopsis

```
enqueue QSPACE queueSpace NAME queueName message
    [enqueueOption]... [ MSGID msgId]
```

Arguments

`QSPACE` *`queueSpace`*
: Specifies *`queueName`*'s queue space.

`NAME` *`queueName`*
: The name of the queue as defined in the JIF.

`message`
: The message data to enqueue, as defined by the queue data type description in the JIF. For more information on message data types, refer to "Service Messages and Data Types" on page 5-15 in *JetNet/Oracle Tuxedo Guide*.

`enqueueOption`
: One or more of the following options:

`CORRID` *`corrID`*
: Associates the message with a queue-independent identifier of up to 32 characters. Because this identifier can be maintained across all queues, the dequeue command can use it to identify a message. Other related messages can also be enqueued with it, such as reply or failure messages that are associated with message.

`DQTIME` *`dequeueTime`*
: Specifies when to make the message available for dequeuing. If you omit this option, the message can be dequeued immediately. This option is valid only if the queue has been configured for time-based ordering; that is, queue order is set to time. For more information, refer to the *Oracle Tuxedo /Q Guide* of the Oracle Tuxedo SDK in your Oracle Tuxedo documentation.

The server dequeues the message and calls the appropriate service if it is monitoring the queue. *`dequeueTime`* can be a relative time (time elapsed after the message is enqueued) or an absolute time. An absolute time must be greater than January 1 1970 00:00:00 UTC. In either case, Panther can dequeue the command only after the specified amount of time has elapsed.

A relative *`dequeueTime`* can be specified in this format:

`"[ +days hours::minutes::]seconds"`

Seconds are required; minutes, hours, or days (space delimiter between days and hours) can also be specified. If more than seconds is specified, the + symbol and the quotation marks are mandatory. If only seconds are specified, both are optional.

Note: JPL's colon preprocessor expands colon-prefixed variables. To prevent expansion of variables that contain colons, you must prefix literal colons with another colon (`::`) or a backslash (`\:`).

An absolute *`dequeueTime`* can be specified in one of these ways:

- The value from a Panther date/time field.
- A date/time string in this format: *`"mm/dd/yy HH::MM"`*

`FAILUREQ` *`queue`* | `NOFAILUREQ`

Specify a failure queue for failure responses, or use `NOFAILUREQ` if no failure message is expected. If neither option is specified, the JIF is checked for the default failure queue.

`FRONT` | `BEFORE_MSGID` *`msgId`*

Place the message in the queue:

- `FRONT`—Put the message at the head of the queue.
- `BEFORE_MSGID` *`msgId`*—Put the message ahead of the message with Oracle Tuxedo message identifier *`msgId`*.

`NOTIMEOUT`

Specifies that the enqueue operation is unaffected by the blocking timeout. This option has no effect on transaction timeouts. If you omit this option, Panther use the setting in the `tp_timeout` property.

`OUTSIDE_TRANSACTION`

Specifies to perform the enqueuing operation outside the current transaction. If message enqueuing fails, the current transaction is unaffected. If you specify this option, transaction-level exception and unload handlers are not executed when their corresponding events are generated.

`PRIORITY` *`priority`*

An integer between 1 and 100, inclusive, that establishes the message's priority, where 100 specifies the highest priority. This option is valid only if the queue's queue order parameter includes a priority setting. An out-of-range priority value generates the exception `TP_INVALID_OPTION_VALUE`. For further information on priority enqueuing, refer to the *Oracle Tuxedo /Q Guide* of the Oracle Tuxedo SDK in your Oracle Tuxedo documentation.

`RCODE` *`returnCode`*
An integer that specifies the return status to be made available to the dequeuing agent. The return code is handed to the reply queue from the service that replies to the message.

`REPLYQ` *`queue NOREPLYQ`*
Specifies a reply queue for replies to the message, or that no reply message is wanted (*`NOREPLYQ`*). If neither option is specified, the JIF is checked for a reply queue.

`MSGID` *`msgId`*
On return, *`msgId`* contains the unique Oracle Tuxedo message identifier that is generated after enqueue executes successfully. You can use this identifier to reference the enqueued message as long as it remains on the original queue.

Environment JetNet, Oracle Tuxedo

Scope Client, Server

Description The enqueue command puts a message in the specified queue. You identify the queue by specifying its queue name and queue space. This queue must be defined in the JIF; otherwise, the command fails and generates the exception `TP_IN VALID_QUEUE`. The enqueued message must conform to the data type defined in the JIF for the designated queue.

An enqueued message can only be removed from its queue by the `dequeue` command, and only after the `DQTIME` time delay (if any) elapses. The reply to this message is put in the `REPLYQ`-specified reply queue or, if this option is omitted, in the reply queue specified in *`queueName`*'s JIF definition. enqueue can also specify a failure queue to supersede the one specified by the JIF. You can prevent reply and failure queueing with the `NOREPLYQ` and `NOFAILUREQ` options, respectively.

By default, messages are enqueued in first-in/first-out (FIFO) order. You can rely on this ordering when dequeuing messages, or you can use specific identifiers. Two kinds of identifiers are available:

- The Oracle Tuxedo-assigned message identifier that is returned with a successful enqueue command; you obtain this identifier through the `MSGID` option.
- A correlation ID that you explicitly assign to the enqueued message. This identifier is independent of the queue in which the message is placed; you should assign a correlation ID in order to identify the reply or failure messages that are generated in response to a dequeued message.

enqueue sets the `tp_return` property to `NULL`.

For more information on queueing and Oracle Tuxedo System /Q, refer to “Reliable Queues” on page 8-11 in *JetNet/Oracle Tuxedo Guide* and refer to your Oracle Tuxedo documentation.

See Also enqueue can generate the following exceptions:

Exception	**Severity**	**Cause**
`TP_INVALID_COMMAND_SYNTAX`	`TP_COMMAND`	Command syntax is invalid.
`TP_INVALID_VARIABLE_REF`	`TP_COMMAND` or `TP_WARNING`	Unable to resolve reference to Panther variable.
`TP_INVALID_OPTION_VALUE`	`TP_COMMAND`	An invalid time value is specified in the `DQTIME` option or `PRIORITY` value is out-of-range.
`TP_INVALID_QUEUE`	`TP_COMMAND`	Queue is not defined in the JIF.
`TP_QUEUE_SPACE_NOT_IN_JIF`	`TP_COMMAND`	Queuespace not found in the JIF.
`TP_NO_OUTSIDE_TRANSACTION`	`TP_WARNING`	There is no current transaction.

See Also `dequeue`

flush

Flushes buffered output to the display

Synopsis

```
flush
```

Description

The flush command performs delayed writes and flushes all buffered output to the display. Panther automatically performs this operation when the keyboard is open and the input queue is empty. This command calls the library function `sm_flush`.

Because Panther uses a delayed-write feature, Panther does not immediately display output from assignments and `msg` statements. Instead, it updates the screen image in memory. When the keyboard is opened or the flush command is called, Panther updates the display from this image.

Frequent calls to this command and its library equivalent `sm_flush` can significantly slow execution. Panther always calls `sm_flush` when the keyboard opens, so the display is always up to date before data entry occurs. Use this command when your procedure requires timed output or non-interactive display–for example, to update a time field.

Example

```
// If this procedure is called as a screen entry function,
// it prints text one character at a time in field
// banner when the screen is opened.


proc welcome
vars w, i
w = "-------Sam's Discount Rentals-------"
for i = 1 while w(i,1) != "" step 1
{
    banner(i) = w(i,1)
    flush
    call delay
}

proc delay
// Lengthen the interval between flushes.
vars i
for i =1 while i < 5 step 1
{ }
```

for

Executes one or more JPL statements the specified number of times

Synopsis

```
for counter = initValue while logicalExpr [ step stepValue]
 [statementBlock]
```

Arguments

`counter`
: A variable whose value may be tested as a condition for continuing or ending for execution

`initValue`
: The initial value of counter

`logicalExpr`
: Specifies the condition for continuing for execution. Execution remains inside the for loop until *`logicalExpr`* evaluates to false. You can specify multiple conditions with the logical operators `AND` (&&) and `OR` (||).

`step` *`stepValue`*
: Optionally specifies the value by which counter is incremented or decremented, where *`stepValue`* is a positive or negative integer constant or variable. The default step value is 1. If *`stepValue`* is a variable, JPL evaluates it only once, before the first evaluation of *`logicalExpr`*. Subsequent changes in the value of the *`stepValue`* variable during loop execution have no effect on step processing.

`statementBlock`
: One or more JPL statements to execute as long as *`logicalExpr`* evaluates to true. If *`statementBlock`* has multiple statements, enclose them with open and close blocking characters {0} on the lines before and after. If there is no statement to execute, enter a null statement {}.

Description

The `for` command starts a loop whose iterations increment a counter variable. Each for statement contains up to three clauses–initialization of the counter variable, a logical expression whose evaluation determines whether to reenter the loop, and optionally, the number by which to increment the counter variable. Panther executes a for statement as follows:

1. Initializes counter to the value of *`initValue`*.

2. Evaluates *`stepValue`*.

3. Evaluates *`logicalExpr`*:

   - If *`logicalExpr`* evaluates to false, stop execution of the loop and exit.
   - If *`logicalExpr`* evaluates to true, execute the for statement or block; increment counter by *`stepValue`*; repeat step 3 (evaluate *`logicalExpr`*).

When the value of *`logicalExpr`* is false, JPL stops loop execution. In the simplest case, it compares counter to a value that specifies the number of times that JPL executes the loop. You can use other values to decide when loop execution ends. For example, you can use counter to evaluate array occurrences and use the value of an occurrence, like a null string, to the end the loop.

When you construct a logical expression, take into account that JPL, unlike C, always fully evaluates a boolean expression. For example, the following for statement traverses a screen's fields by field number (ct) until the last field or the first modified field is reached:

```
vars ct
vars n_flds = @screen("@current")->numflds

for ct = 1 while ct <= n_flds && @field_num(ct)->mdt == PV_NO
```

If all fields are unmodified, ct increments to one greater than `n_flds` on the last pass through the for loop, so the first condition evaluates to false; however, JPL also evaluates the second condition `@field_num(ct)`, which is invalid. Consequently, JPL issues an error message and stops execution of the remaining code.

Example

```
// Change each element of an array to its absolute value.
vars i
for i = 1 while i <= 10 step 1
{
  if amounts[i] == ""
      amounts[i] = "0"
  else if amounts[i] < 0
      amounts[i] = -amounts[i]
}
```

See Also

`next`, `break`, `while`

global

Declares global JPL variables

Synopsis `global varSpec[, varSpec]...`

Arguments

`varSpec` Specifies the global variable's name and properties as follows:

`varName [[numOccurs]] [(size)] [= initValue]`

`varName`
The name of the variable, where *`varName`* is a string that contains up to 31 characters. Global names can use any combination of letters, digits, or underscores, where the first character is not a digit. Panther also allows usage of two special characters, the dollar sign ($) and period (.).

[*`numOccurs`*]
Optionally declares *`varName`* as an array of *`numOccurs`* occurrences. The default number of occurrences is 1. For example the following statement declares dependents as an array of ten occurrences:

```
global dependents[10]
```

(*`size`*)
Optionally specifies the number of bytes allocated for this variable; Panther automatically allocates an extra byte for the terminating null character. The default size is 255 bytes. For example, the following statement declares the variable zip with a size of 10 bytes:

```
global zip (10)
```

= *`initValue`*
Optionally initializes the variable to *`initValue`*, where *`initValue`* can be any expression less than or equal to the variable's size. If no value is assigned, Panther initializes the variable to null string ("").

If the variable is declared as an array, you can initialize its occurrences. For example:

```
global ratings[5] = {"G", "PG", "PG-13", "R", "NC-17"}
```

Occurrence values are comma-delimited, and can be any constants or variables that are in scope, including other global variables and widget names.

Description

The `global` command creates one or more global JPL variables. These variables are visible to the entire application and can be referenced at any time.

Avoid using names already in use by Panther itself–for example, logical key names such as `XMIT` and `EXIT`, and bit mask settings such as `K_EXPOSE` and `K_ENTRY`. Because Panther uses these variables internally, reinitializing them can yield unpredictable and possibly harmful results.

See Also

`vars`

if

Conditionally executes one or more JPL statements

Synopsis

```
if logicalExpr
   statementBlock

[else if logicalExpr
   statementBlock]
...
[else
   statementBlock]
```

Arguments

`logicalExpr`
: Specifies the condition under which JPL executes `statementBlock`, where `logicalExpr` can be any logical expression. For more information on logical expression construction, refer to “Logical Expressions” on page 19-55 in *Application Development Guide*.

`statementBlock`
: One or more statements that JPL executes if the preceding `logicalExpr` evaluates to true. If `statementBlock` has more than one statement, enclose the block with open and close blocking characters `{}` on the lines before and after.

`else if logicalExpr`
: Optionally specifies the statement block to execute if all previous `if` and `else if` conditions evaluate to false and `logicalExpr` evaluates to true.

`else`
: Optionally specifies the statement block to execute if all previous `if` and `else if` conditions evaluate to false. Each `else` must be paired with an `if` statement and follow all `else if` statements associated with that `if`.

Description

The `if` command specifies conditional execution of other JPL statements. Each `if` can be followed by one or more `else if` commands to create a chain of conditional processing. JPL executes each `if` and `else if` in the chain until it evaluates one of the conditions to true; JPL then executes the statement block and exits the chain. If all conditions in an `if` chain evaluate to false and the chain ends with an `else` command, JPL executes the `else` statement block. If the `if` chain omits an `else` command, JPL simply exits the chain and continues module execution.

Example

```
//Determine a person's sex, based on personal title.
if title == 'MR'
   sex = 'Male'

else if title == 'MS'
   sex = 'Female'

else if title == 'MRS'
   sex = 'Female'

else if title == 'MISS'
   sex = 'Female'

else
{
   sex = 'Unknown'
   msg err_reset 'Please supply a title.'
}
```

include

Interpolates the contents of another module at the current statement line

Synopsis `include` *`module`*

Arguments *`module`*

The name of the module to include.

Description The `include` command replaces the current `include` statement with the contents of the specified file module. `include` lets you write and maintain JPL in separate modules. You can thereby avoid hard-coding the same procedure across several modules, or allocating memory for public modules. The included module can itself contain its own `include` statements. You can nest `include` statements up to eight levels deep.

Panther looks for `module` among available modules in this order:

1. Memory-resident modules.
2. Library module in an open library.
3. The current directory.
4. File module in a directory specified by `sm_initcrt`.
5. File module in a directory specified by `SMPATH`.

At runtime, JPL compiles and loads the included module as needed. Compilation occurs before JPL executes the primary module or procedure that contains the `include` statement. Consequently, compilation errors in the included module prevent execution of the primary module.

jif_check

Determines if the JIF has changed

Synopsis

```
jif_check
```

Environment JetNet, Oracle Tuxedo

Scope Client, Server

Description The `jif_check` command checks whether changes occurred in the JIF. You typically use this command in the `request_received` handler, which is called on all service requests (refer to "Request_received Events" on page 6-22 in *JetNet/Oracle Tuxedo Guide)*. If a change has occurred, `jif_check` sets the value of `tp_return` and raises a `JIF_changed` event (refer to "Jif_changed Events" on page 6-15 in *JetNet/Oracle Tuxedo Guide*); the default handler for this event calls `jif_read` to reread the JIF, and readvertises all services.

Exceptions `jif_check` can generate these exceptions:

Exception	Severity	Cause
`TP_JIF_LOWER_VERSION`	`TP_REQUEST`	An older version of the JIF is in place; the `tp_return` property is set to `TP_JIF_OLDER`.
`TP_JIF_ACCESS_FAILED`	`TP_REQUEST`	The JIF cannot be accessed.

See Also `jif_read`

jif_read

Rereads the JIF

Synopsis

```
jif_read
```

Environment JetNet, Oracle Tuxedo

Scope Client, Server

Description The `jif_read` command rereads the JIF and updates all service information. This command is typically called in the `jif_changed` event handler before it readvertises services (refer to “Jif_changed Events” on page 6-15 in *JetNet/Oracle Tuxedo Guide*).

Exceptions `jif_read` can generate one exception:

Exception	Severity	Cause
`TP_JIF_ACCESS_FAILED`	`TP_REQUEST`	The JIF is not accessible.

See Also `jif_check`

log

Logs a message to the machine event log

Synopsis `log` *`logEntry`*

Arguments *`logEntry`*

The string to write to the log file: supply either a string constant or variable.

Environment JetNet, Oracle Tuxedo, COM, EJB

Scope Client, Server

Description The `log` command writes an entry to the machine-specific log file. For example, you can define a `server_exit` handler that logs a message when the server shuts down:

```
proc server_exit()
log "Enterprise Bank server shutting down"
return 0
```

For COM/MTS applications, the log file must be named `server.log` and reside in the service component's application directory. `log` is not supported for COM/MTS clients. For more information, refer to “Logging Server Messages” on page 3-17 in *COM/MTS Guide*.

Exceptions In JetNet/Oracle Tuxedo applications, `log` can generate one exception:

Exception	Severity	Cause
`TP_INVALID_CONNECTION`	`TP_WARNING`	There is no connection to the middleware.

msg

Writes a message to the terminal

Synopsis `msg` *`mode message`*

Arguments *`mode`*

Specifies the message's format and behavior with one of these arguments:

`emsg`
: Displays message as an error message and awaits user acknowledgement.

`err_reset`
: Identical to `emsg` except when the message is displayed on the status line: in that case, `err_reset` forces the cursor on at its current position.

`qui_msg`
: Displays message as an error message and awaits user acknowledgement. message is preceded by the `SM_ERROR` string from the message file–for example, `ERROR`. In GUIs, the `SM_ERROR` text is also preceded by the stop icon.

`quiet`
: Identical to `qui_msg` except when the message is displayed on the status line: in that case, `quiet` forces the cursor on at its current position.

`setbkstat`
: Installs message as the background status line, which displays when no other message is active.

`d_msg`
: Displays message arguments on the status line and leaves it there until cleared or replaced by another message. Text displayed using `d_msg` is buffered. You can clear the buffer by another `msg d_msg` command that supplies an empty string(""). `msg d_msg` displaces the status line message displayed by `msg setbkstat`.

`message`

One or more comma-delimited arguments that comprise the message to display. Each argument can be a string or numeric constant, or a variable.

Note that msg query allows only one argument. All other arguments for mode allow multiple arguments.

Description

The msg command displays messages on the status line or in a popup window in one of several modes. Each mode correspond to a Panther library function. To display messages in a dialog box with standard command buttons, call sm_message_box.

Window versus Status Line Display

By default, GUI versions of Panther always display messages in a popup window with an OK button. Character-mode Panther displays messages in a window only if the configuration variable MESSAGE_WINDOW is set to ALWAYS. If you set this variable to WHEN_REQUIRED (the default), character-mode Panther displays messages on the status line except when these conditions occur:

- The message overflows the status line. Note that Panther prevents the message from overlapping the cursor row/column display, if it is turned on.
- The message wraps to multiple lines.
- You specify window display with the %W format option.

Note: You can force display of a message to the status line on all GUI and character-mode platforms, regardless of MESSAGE_WINDOW's setting, if the message contains the %Mu option, or the setup variable ER_KEYUSE is set to ER_USE. Also, the setbkstat and d_msg modes always display messages on the status line.

Message Acknowledgment

Users can dismiss the error message by pressing the acknowledgement key. In a window-displayed message, OK and space bar also serve to dismiss the error message. The acknowledgement key (by default, spacebar) can be set through the setup variable ER_ACK_KEY. If the user acknowledges the message through the keyboard, Panther discards the key. You can modify this behavior for individual messages through the %Mu option, described later.

Message Appearance and Behavior

Several setup variables determine default message presentation and behavior. For more information about these variables, refer to “Message Display” on page 2-20 in *Configuration Guide*. You can change these defaults at runtime through sm_option.

You can change message behavior and appearance for individual messages by embedding percent escape options in the message text. Use these options after the call to sm_initcrt; otherwise, the percent characters appear as literals.

`%A`*`attrValue`*

Change the display of the subsequent string to the *`attrValue`*-specified attribute, where *`attrValue`* is a four-digit hexadecimal value. If the string to get the attribute change starts with a hexadecimal digit (0...F), pad *`attrValue`* with leading zeros to four digits. Refer to Table 45-2 on page 45-9 in *Application Development Guide* for valid attribute values.

This option is valid only for messages that display on the status line. Panther ignores this option if the message displays in a window.

`%B`

Beep the terminal before the message displays. This option must precede the message text.

`%K`*`keyLogical`*

Display key label for logical key, where *`keyLogical`* is a logical key mnemonic or hex value. When Panther displays the message, it replaces *`keyLogical`* with the key label string defined for that key in the key translation file. If there is no label, the `%K` is stripped out and the mnemonic remains. Key mnemonics are defined in `smkeys.h`

Note: If `%K` is used in a status line message, the user can push the corresponding logical key onto the input queue by mouse-clicking on the key label text.

`%Md`

Force the user to press the acknowledgment key (`ER_ACK_KEY`) in order to dismiss the error message. Panther discards the key that is pressed. If the user presses any other key, Panther displays an error message or beeps, depending on how setup variable `ER_SP_WIND` is set. The `%Md` option corresponds to the default message behavior when setup variable `ER_KEYUSE` is set to `ER_NO_USE`.

This option must precede the message text.

`%Mt[`*`timeOut`*`]`

Force temporary display of message to the status line. Panther automatically dismisses the message after the specified timeout elapses and restores the previous status line display. Timeout specification is optional; the default timeout is one second. You can specify another timeout in units of 1/10 second with this syntax:

#(*n*)

n is a numeric constant that specifies the timeout's length. If n is more than one digit, the value must be enclosed with parentheses. For example, this statement displays a message for 2 seconds:

```
msg emsg "%Mt(20) Changes have been saved to database."
```

The user can dismiss the message before the timeout by pressing any key or mouse clicking. Panther then processes the keyboard or mouse input.

If the message is too long to fit on the status line, Panther displays the message in a window. In this case, users can dismiss the message only by choosing OK or pressing the acknowledgement key. Panther then discards any keyboard input.

This option must precede the message text. It is ignored by `setbkstat` and `d_msg` modes.

`%Mu`

Force message display to the status line and permit any keyboard or mouse input to serve as error acknowledgment. Panther then processes the keyboard or mouse input.

If the message is too long to fit on the status line, Panther displays the message in a window. In this case, users can dismiss the message only by choosing OK or pressing the acknowledgement key. Panther then discards any keyboard input.

This option must precede the message text. It is ignored by `setbkstat` and `d_msg` modes.

`%N`

Insert a line break. This option is invalid for `setbkstat` and `d_msg` modes.

`%W`

Forces display of the message in a window. This option is ignored by `setbkstat` and `d_msg` modes.

Example

```
// Indicate that the entry to the field state is invalid.
msg err_reset ':state is not a U.S. state'

// Indicate that the current entry is being processed.
// Note that d_msg overrides delayed write and immediately
// flushes text to the screen.
msg d_msg 'Processing :name'

// Ask whether the user wants to quit the current screen.
vars quit
```

```
quit = sm_message_box \
    ('Are you ready to quit?' ,"",SM_MB_OKCANCEL,"")

if quit = SM_ID_OK
  return 0

vars field1 message
field1 = "message"
message = "Quick brown fox"

// This will display 'message' on the status line.
msg emsg field1

// This will also display 'message'.
msg emsg ":field1"

// This will display 'field1'.
msg emsg "field1"

// This will display 'Quick brown fox'.
msg emsg :field1

// These messages use percent escapes.
// Print message in red
msg emsg "%A004Stop now."

msg emsg "The menu toggle is %KMTGL"
msg emsg "Enter value.%NPress XMIT."
msg qui_msg "%WInvalid password."
msg err_reset "%MdPlease enter a positive value."
```

See Also `sm_message_box`

next

Skips to the next iteration of a loop

Synopsis

```
next
```

Description

The `next` command is valid in any `for` or `while` loop. `next` terminates the current iteration of the loop and starts the next iteration. When a `next` statement executes, JPL skips all subsequent statements until the end of the loop. If the loop is controlled by a `for` statement, JPL increments the loop's step value. It then tests the loop condition; if the condition evaluates to true, JPL executes the `while` or `for` statement block. `next` resembles the `continue` statement in C.

Example

```
// Process all the engineers in a list of people.
vars k
for k = 1 while job[k] != "" step 1
{
   if job[k] != "engineer"
      next
//process mailing label for engineers...
}
```

See Also

`break`, `for`, `while`

notify

Sends an unsolicited message to a client

Synopsis `notify TYPE` *`msgType`* `(` *`message`* `) [ NOTIMEOUT]`

Arguments

`TYPE` *`msgType`*

Specifies the message's data type, where *`msgType`* is one of these values:

- `JAMFLEX`
- `STRING` (Oracle Tuxedo only)
- `FML` (Oracle Tuxedo only)
- `FML32` (Oracle Tuxedo only)

For more information on message data types, refer to "Service Messages and Data Types" on page 5-15 in *JetNet/Oracle Tuxedo Guide*.

`message`

The message data, which must conform to *`msgType`*.

`NOTIMEOUT`

Disregard blocking timeouts; however, transaction timeouts remain in effect.

Environment JetNet, Oracle Tuxedo

Scope Server

Description The `notify` command sends a message to the client whose service request the server is currently processing. The client to be notified cannot be another server.

For example, this server procedure might be used to notify ATM clients about `bankservices` while it processes their requests. It uses source to identify itself as the source of the message:

```
proc bankinfo ()
// service BANK_INFO
// send a message with bank news to the client
notify TYPE JAMFLEX \
   ({source="notify_news", msg="Low rate mortgages with \
     no points - no closing fees. \
     Stop in your local branch for details!"})
service_return ()
```

Messages delivered via notify are unsolicited. In order for unsolicited messages to be interpreted correctly by agents receiving them, a message handler must be installed. Because the handler is unaware of a message's origin, it is important that a standard method of identifying the source of unsolicited messages be established for the entire application. For more information on writing a message handler for your application, refer to "Recognizing the Message Source" on page 6-18 in *JetNet/Oracle Tuxedo Guide*.

Exceptions

`notify` can generate the following exceptions:

Exception	**Severity**	**Cause**
`TP_INVALID_ARGUMENT_LIST`	`TP_COMMAND`	More than one argument is passed to message.
`TP_TIMEOUT`	`TP_COMMAND`	notify times out before completion.

See Also

`broadcast`, `client_init`, `receive`

parms

Declares parameters in the unnamed procedure of a JPL module

Synopsis `parms [ deref ] paramName[, paramName]...`

Arguments `deref`

Specifies to pass in the values of the caller's arguments. If you omit the `deref` qualifier, JPL passes in the literal value of the caller's arguments. In the case of a variable, JPL passes in the name of the variable instead of its value. Omit this argument if you use the `parms` command to get the standard arguments passed in by a field, group, or screen.

`paramName`

The name of the parameter, where *`paramName`* is a string that contains up to 31 characters. JPL parameter names can use any combination of letters, digits, or underscores, where the first character is not a digit. Panther also allows usage of two special characters, the dollar sign ($) and period (.).

Description The `parms` command declares one or more parameters in a JPL module's unnamed procedure. An unnamed procedure must be the first procedure in a JPL module; because this procedure omits the `proc` statement, you must use the `parms` command to receive any arguments that are passed in by its caller. Also use it in a field's validation module or in an external non-public JPL module to get the standard arguments passed by screens, groups, and fields. For more information about the standard arguments available for screen modules, refer to “Screen Function Arguments” on page 44-11 in *Application Development Guide*; for widget modules, refer to “Field Function Arguments” on page 44-15. `parms` statement can declare up to twenty comma-delimited parameters. If you declare more parameters than are actually passed, Panther initializes the extra parameters to empty strings. If you declare fewer, the undeclared parameters are inaccessible. Like variables, parameters that are declared in a module's unnamed procedure are accessible to all procedures in that module.

Example

```
// call module calculatecall calculate(subtotal, state)

//first unnamed procedure in module calculate
parms amt, st
if st == 'CA'
  tax = 0.0725
```

```
else if st == 'NY'
  tax = 0.085
else
  tax = 0.00
total = amt * (1 + tax)
```

See Also `vars`, `proc`

post

Posts an event

Synopsis

```
post EVENT eventName TYPE msgType (message) [postOption]...
```

Arguments

`EVENT` *`eventName`*

The event to be posted, where *`eventName`* can be up to 31 characters long, but cannot start with a period (.) or sm.

`TYPE` *`msgType`*

Specifies the data type of the message to accompany this event posting, where *`msgType`* is one of these values:

- `JAMFLEX`
- `STRING`
- `FML`
- `FML32`

If `TYPE` is not specified, the default is `STRING`. For more information on message data types, refer to “Service Messages and Data Types” on page 5-15 in *JetNet/Oracle Tuxedo Guide*.

`message`

Data to accompany the event posting; the message's format must conform to the `TYPE`-specified data type.

`postOption`

One or more of the following options:

`NOREPLY`

The event broker does not wait for replies from subscribers to the event before returning to the posting agent.

`NOTIMEOUT`

The event posting is not subject to blocking timeouts; however, transaction timeouts remain in effect.

`OUTSIDE_TRANSACTION`

Execute the event posting operation outside of the current transaction (if issued within a transaction). If you specify this option, transaction-level exception and unload handlers are not executed when their corresponding events are generated.

Environment Oracle Tuxedo

Scope Client, Server

Description The `post` command lets a client or server post an event. When an event is posted, the event broker notifies all subscribers of the event. If successful, post sets the `tp_return` property to the number of notifications dispatched; otherwise, it sets `tp_return` to `TP_FAILURE`. For information about the Oracle Tuxedo event broker and configuration requirements, refer to "Event Brokering" on page 8-5 in *JetNet/Oracle Tuxedo Guide*; also refer to your Oracle Tuxedo documentation.

Exceptions The `post` command can generate these exceptions:

Exception	Severity	Cause
`TP_EVTBROKER_ACCESS_FAILED`	`TP_COMMAND`	Unable to access the event broker sever.
`TP_NO_OUTSIDE_TRANSACTION`	`TP_WARNING`	There is no current transaction.
`TP_POSTING_FAILED`	`TP_COMMAND`	An error occurs when posting a transactional event to either a service routine or a reliable queue on behalf of the caller's transaction.
`TP_INVALID_COMMAND_SYNTAX`	`TP_COMMAND`	Command syntax is invalid.
`TP_INVALID_VARIABLE_REF`	`TP_COMMAND` or `TP_WARNING`	Unable to resolve reference to Panther variable.

See Also `subscribe`, `unsubscribe`

proc

Starts a JPL procedure definition

Synopsis `[returnType] proc procName [ ([ param[, param ]...] ) ]`

Arguments

returnType

Specifies the data type of the procedure's return value. An unqualified `proc` command returns an integer value. You can specify to return a string or double precision value by qualifying the `proc` command with the keywords `string` or `double`, respectively.

procName

A character string that specifies the JPL procedure name. Procedure names can be up to 31 characters long and contain any keyboard character except a blank space. When naming procedures in screen and public modules, be sure to avoid name conflicts, especially with any external modules that you wish to call by name.

param

A parameter to receive the corresponding argument passed by this procedure's caller. You specify parameters as a comma- or space-delimited argument list within parentheses. Panther passes arguments by value–that is, the called procedure gets its own private copies of the values in the calling procedure's arguments. This means that the called procedure cannot directly alter a variable in its caller; it can only alter its own copies.

Description

The `proc` command names a procedure and optionally specifies its parameters and return value's data type. If a module contains multiple procedures, each `proc` statement serves to end the previous procedure. Only named procedures can be called from other procedures, and from application hooks such as control strings and Focus properties.

In the following example, the call to procedure `process_input` passes data from variables `data1` and `data2` to the procedure's corresponding parameters. The procedure is defined to return a double value. This return value is used to determine whether the if statement evaluates to true or false:

Example

```
if process_input(data1, data2) > 0.16667
...

double proc process_input(d1, d2)
```

```
vars retval
//process d1 and d2 values
return retval
```

Because a `proc` statement marks the end of one procedure and the start of another, you cannot embed one procedure definition inside another. Refer to Chapter 19, “Programming in JPL,” in *Application Development Guide* for more information on procedure structure and execution.

See Also `call`

public

Reads JPL modules into memory and makes their procedures available to application

Synopsis `public` *`moduleName`*`[` *`moduleName`*`]...`

Arguments *`moduleName`*

Specifies the module to read into memory (if necessary), where *`moduleName`* is a string constant or colon-expanded variable that names a library module or memory-resident module. If Panther cannot find *`moduleName`*, it issues an error message.

Note: If the `public` command is issued in a screen's unnamed procedure and *`moduleName`* cannot be found, no error message is issued.

Description The `public` command reads the procedures contained in one or more JPL modules. If the modules are not already memory-resident, public compiles them and puts them in memory, making the contents of the module available to the application as a whole. It also executes the first procedure if it is unnamed. All procedures beginning with a `proc` statement are available until the application exits or you remove their module from memory with an `unload` statement. `public` lets you store generic procedures in library modules that are easy to edit and available to any application. For example, these procedures handle user exits:

Example

```
proc quit
vars ans
ans = sm_message_box \
    ("Are you ready to quit?", "", SM_MB_YESNO, "")
if ans = SM_IDYES
  return 1
else
  return 0

proc end
msg emsg 'Program exit.'
```

Given that these procedures are in library module `exit_handler`, you can make them available to the application by entering this public command in the opening screen's unnamed procedure (accessed through the screen's JPL Procedures property):

```
public exit_handler
```

You can now call quit from any available application hook, for example, from a control string that is associated with the `EXIT` key:

```
EXIT=^(0=&nextscreen; 1=^end)quit
```

You can issue the `public` command on a module only once. Panther ignores `public` commands on a module that is already public.

Note: If you test an application that loads a public module, that module remains in memory until you explicitly unload it or Panther exits. If you edit the module after exiting test mode, remember in the next test session to unload the module's earlier version and reload the new one in order to see your changes.

See Also `unload`

raise_exception

Sends an error code back to the client

Synopsis `raise_exception` *`returnCode message`*

Arguments

`returnCode`	Specifies the exception error to return to the client for COM components and EJBs.
`message`	Specifies the message to return to the client for CORBA components.

Environment COM, EJB

Scope Server

Description `raise_exception` sends an error code back to the client, generally a negative number. The client's error handler can then decide what to do based on the value sent. There are some conventional exception codes defined by Microsoft for use in COM programming.

Calling this command causes the remainder of the JPL to be skipped. The return code from the aborted JPL will be 0.

Example

```
// This JPL procedure sends an exception code.

proc my_method()
{
    . . .
    return_args (id, name)
    raise_exception -2
}
```

See Also `receive_args`, `return_args`

receive

Receives data sent via send or from a remote client via service_call

Synopsis

```
receive [bundle bundleName] [item itemNo] [keep] data fieldExpr
```

```
receive {ARGUMENTS | MESSAGE} ([receiveArg])
```

Arguments

`bundle` *`bundleName`*

Optionally names the buffer, or bundle, from which to receive data, where *`bundleName`* can be a string constant or variable. Bundle data is written by `send` commands; if the `send` command supplies a bundle name, Panther creates a bundle with that name. Panther by default maintains up to ten bundles of send data in memory; change the number of available bundles by setting the `max_bundles` property. If no name is supplied, Panther gets data from the unnamed bundle–that is, a bundle whose data is sent from the last `send` command that omitted a bundle name.

`item` *`itemNo`*

Specifies the bundle offset from which to start reading data, where item numbering begins at 1. If you omit this argument, receive starts getting bundle data from the first item. `receive` counts data items in the same order as they were sent. Each item in the bundle can contain one or more occurrences; because an array is regarded as a single data item, Panther disregards its occurrences when it evaluates *`itemNo`*.

`keep`

Specifies to leave the bundle data intact after receive completes execution. This lets multiple receive statements specify the same bundle of data. By default, `receive` destroys the bundle and frees the memory allocated for it after it completes execution.

`data` *`fieldExpr`*

Specifies the fields or occurrences to receive the bundle data. Refer to "Object Specification" on page 19-33 in *Application Development Guide* for more information about valid field expressions. You can specify multiple *`fieldExpr`* arguments delimited by commas.

If *`fieldExpr`* is a non-subscripted array, `receive` reads the bundle data into all of the array's occurrences. You can specify a single occurrence or range of occurrences within an array by subscripting it with this format:

```
array[intExpr[..[intExpr] ] ]
```

where `intExpr` evaluates to an integer. If you omit the last occurrence specifier, `receive` reads into all occurrences from the one specified to the end of the array. The following examples show different subscripts that are valid:

```
receive data @widget("empno")[1] //read only occurrence 1
receive data empno[1..10]  //read into occurrences 1-10
receive data empno[ct..]   //read all occurrences from ct
```

`ARGUMENTS`
Used by services to receive their arguments from a client agent that initiated a service request. Use of this keyword is restricted to servers.

`MESSAGE`
Enables clients to receive unsolicited messages via a message handler. Use of this keyword is restricted to clients.

`receiveArg`
Specifies the target variables to receive the incoming message data. Parentheses are required even when no argument is specified. The format of the incoming data is specified in the JIF's definition of the service. For more information on message data types, refer to "Service Messages and Data Types" on page 5-15 in *JetNet/Oracle Tuxedo Guide*.

Scope

Client, Server

Description

The `receive` command is used in different ways depending on whether it is used to receive data/messages from remote client or server agents using the middleware, or if it is receiving data locally from another screen. In JetNet/Oracle Tuxedo applications, `receive` is used either by a service to receive data from a client, or in a message handler to receive unsolicited messages from servers.

Local Receive

When a `receive` executes independently of the middleware, Panther reads data from a bundle that was written by an earlier `send` command, typically, from another screen. `receive` reads the data into its *`fieldExpr`* arguments in the same order that it was sent. Unless you supply the `keep` argument, the bundle data is discarded when `receive` completes execution.

`receive` sequentially pairs each *`fieldExpr`* argument to a data item in the bundle. If the data item contains multiple occurrences, `receive` reads as many occurrences into *`fieldExpr`* as the field allows, or as many as the *`fieldExpr`* expression specifies. If any occurrences remain unread, `receive` ignores them and reads the next data item into its corresponding target.

You can use the item argument to start reading data from a specified offset in the bundle. `receive` starts reading data from this offset.

If a bundle item has more occurrences than are currently allocated for the target array, Panther allocates new occurrences for the overflow data. If the incoming data overflows the array's maximum number of occurrences or a specified range, `receive` ignores the extra occurrences.

If a bundle item has fewer occurrences than currently allocated for the target array, `receive` writes to the array as follows:

- If no range is specified, Panther overwrites the array with the bundle data and discards previous data in remaining occurrences.
- If a range is specified, Panther writes only to those occurrences. Data in other occurrences remains intact. If the range has more occurrences than the incoming data, Panther discards previous data in the remaining occurrences.
- If an unbounded range is specified, for example, `DATA empno[4..]`, Panther overwrites the array from the specified occurrence and discards previous data in remaining occurrences. Data in occurrences that precede the range remains intact.

If a data argument is invalid, for example, the target field does not exist or the range of occurrences is invalid, the `receive` command aborts data transfer prematurely and posts an error message. Panther ignores remaining bundle data and, unless `keep` was specified, destroys the bundle.

Middleware API Receive

In JetNet/Oracle Tuxedo applications, when the middleware intercepts `receive`, it establishes a mapping for incoming data and applies that mapping (unloads the data) when the data is available.

Use the `ARGUMENTS` keyword along with *`receiveArg`* to specify a Panther mapping for the incoming data and to request that the data be unloaded to those Panther targets. For more information on specifying arguments, refer to “Service Messages and Data Types” on page 5-15 in *JetNet/Oracle Tuxedo Guide*.

A message handler uses the `receive` command to specify a Panther mapping for the incoming message and to request that the message actually be unloaded to those Panther targets based on the mapping specified in *`receiveArg`*. A message handler is invoked to process unsolicited messages. These include broadcast messages from other clients or servers, and notify messages from the servers currently processing service requests for the client.

For example, the following code shows receive used in a message handler:

```
// The message handler
proc msg_handler (type)
vars msgStr
if (type=="JAMFLEX")
{
   receive MESSAGE ({msgStr})
   msg emsg msgStr
}
return


// Install the message handler:
...
@app()->hdl_message = "msg_handler"
...
```

The next example shows the code for service `VAL_PIN`, which validates a client logging in with last name and password. The client end of this process is shown in the `service_call` command.

```
proc val_pin()
// service VAL_PIN

receive ARGUMENTS ({last_name, pin})
call sm_tm_command("VIEW")

if (!@dmrowcount)
{
   service_return failure \
      ({message = "Password or name is invalid; try again"})
}
service_return ({message = @tpi_null, owner_ssn})
```

Exceptions

When used to receive data via the middleware, the `receive` command can generate the following exceptions:

Exception	**Severity**	**Cause**
`TP_INVALID_ARGUMENT`	`TP_COMMAND`	Argument specification doesn't match incoming data.
`TP_INVALID_ARGUMENT_LIST`	`TP_COMMAND`	No arguments are available.
`TP_INVALID_BUFFER`	`TP_COMMAND`, `TP_WARNING`	The type of the data buffer received is not the same type as specified in the JIF.
`TP_INVALID_BUFFER_VERSION`	`TP_COMMAND`, `TP_WARNING`	`JAMFLEX` is the buffer type, but the version is incompatible.
`TP_INVALID_CALL`	`TP_COMMAND`	`receive ARGUMENTS` is used outside of a service.
`TP_INVALID_CONTEXT`	`TP_COMMAND`	`receive` is called from a client but not within a message handler.
`TP_NO_OUTSTANDING_MESSAGE`	`TP_COMMAND`	`receive MESSAGE` is used and there is no message to receive.

See Also `send`, `service_call`, `service_return`

receive_args

Receives in and in/out parameters for a method

Synopsis `receive_args` *`argList`*

Arguments *`argList`*

Specifies a comma-delimited list of target variables to receive the incoming data for the method.

Environment COM, EJB

Scope Server

Description The `receive_args` command processes a list of in and in/out parameters that have been passed by a client when calling a method. Since the method's parameters cannot be passed directly to a JPL procedure, this command is how the method receives incoming data.

When processing the targets on the list, the command skips the out only parameters, and takes the other parameters in the order in which they were provided.

Example

```
// This JPL procedure implements the GetCustomer method.
// The parameters are defined as:

   // [in, out]  CompanyName
   // [in]       CompanyID
   // [out]      CustomerID
   // [out]      Phone

proc GetCustomer
{
   receive_args (CompanyName, CompanyID)
   call sm_tm_command ("VIEW")
   return_args (CompanyName, CustomerID, Phone)
```

See Also `return_args`, `raise_exception`

return

Exits a JPL procedure

Synopsis `return [`*`retval`*`]`

Arguments *`retval`*

The value to return to this procedure's caller, where the data type of *`retval`* depends on the procedure definition. Supply either a constant, a variable or an expression that evaluates to a string or numeric value. If no argument is supplied, Panther returns a value of `0` or null string, depending on the procedure's return type.

Description The return command causes a JPL procedure to exit. Control is returned to the procedure's caller, if any, or to the Panther runtime system.

JPL automatically returns with either `0` or null string to a procedure's caller when it reaches the end of the called module or another `proc` statement. Use the return statement to exit before the end of a procedure, or to return a value other than zero or the null string.

Example

```
// Call procedure checknum to evaluate value of num
// field. Based on its value, return an integer that
// determines the next procedure to call


vars ret
ret = checknum()
if ret == 1
  call lownum_process()
else if ret == 2
  call midnum_process()
else
  call hinum_process()

proc checknum()
if num < 0
  return 1
if num < 500
  return 2

return 3
```

return_args

Returns in/out and out parameters for a method

Synopsis `return_args` ***`argList`***

Arguments *`argList`*

Specifies a comma-delimited list of variables whose data will be returned to the client.

Environment COM, EJB

Scope Server

Description The `return_args` command passes a list of variables whose data is to be passed back to the client. These are matched in order with the method's in/out and out parameters.

Example Refer to the example for `receive_args`.

See Also `receive_args`, `raise_exception`

runreport

Invokes the report generator and runs the specified report

Synopsis

```
runreport filename[!reportname ] [ (arg[, ... ] )] [ option ]...
```

Arguments

`filename`

Specifies the name of a report file. Panther looks for the file in the application's memory-resident list, then in all open libraries. For remote report processing, the report file must be in a server or common library.

`reportname`

The name of a report definition in filename to invoke. If `reportname` is omitted, Panther uses the first report definition in the report file.

`arg`

You can supply one or more arguments to the report. Each argument must be a valid JPL expression–either a string within quotation marks, a number, or the name of a Panther variable to evaluate when the report is run. In order to process these arguments, the following conditions must be true:

- Each argument has a corresponding parameter that is declared in the report node's Parameters property (refer to "Report Parameters" on page 5-2 in *Reports*).
- Arguments are supplied in the same order as their corresponding parameters
- Each declared parameter exists in the report as a widget, a JPL global variable, or an LDB variable.

Note: Any punctuation that has a special usage in the operating system such as parentheses (), must be prefixed with an escape character (\).

`option`

You can specify invocation options. For a description of invocation options, refer to "Setting Invocation Options" on page 9-9 in *Reports*.

Description

The `runreport` command invokes Panther's report generation facility to execute the specified report.

Example The following JPL procedure runs the `custinfo.rpt` report for the value in the `cust_num` variable and writes the report to the file `custinfo.txt`.

```
proc make_report
   runreport custinfo.rpt (cust_num) output=custinfo.txt
   return
```

See Also `sm_rw_runreport`

send

Sends data to a buffer for retrieval by the receive command

Synopsis

```
send [ bundle bundleName ] [ append ] data dataExpr[,...]
```

Arguments

`bundle` *`bundleName`*

Optionally names the buffer, or bundle, in which to store the send data, where *`bundleName`* can be a string constant or variable. Bundle names can be up to 31 characters long. By using names, you can maintain additional bundles of send data in memory. The number of available bundles (including the unnamed bundle) defaults to ten, but can be changed by setting the `max_bundles` property. For example, this command sends data to named bundle `empData`:

```
send BUNDLE "empData" DATA empno, dept, status
```

If an existing bundle is already named *`bundleName`*, Panther frees the existing bundle and replaces it with the new one. If the named bundle exceeds the number of allowable bundles, Panther removes the oldest bundle from memory.

If no name is supplied, Panther stores the data in an unnamed bundle–that is, a bundle whose name is an empty string. Panther uses the unnamed bundle for `receive` calls that specify no bundle name.

`append`

Optionally appends the send data to the specified or unnamed bundle.

`data` *`dataExpr`*

Specifies the data to send from this screen, where *`dataExpr`* can be a constant, JPL variable, or field expression. Refer to “Object Specification” on page 19-33 in *Application Development Guide* for more information about valid field expressions. You can specify multiple data arguments delimited by commas.

If *`dataExpr`* is a non-subscripted array, send writes all its occurrences. You can specify a single occurrence or range of occurrences within an array by subscripting it with this format:

```
array[intExpr[..[intExpr]]]
```

where *`intExpr`* evaluates to an integer. If you omit the last occurrence specifier, send writes all occurrences from the one specified to the end of the array. The following examples show different subscripts that are valid:

```
send DATA @widget("empno")[1] //get only occurrence 1
send DATA empno[1..10]       //get occurrences 1-10
send DATA empno[ct..]       //get all occurrences from ct to end
```

Description

The `send` command writes screen data to a buffer that is accessible through the `receive` command. `send` can send one or more values from fields and array occurrences on a screen. It can also send constant values and JPL variables, as well as parts of arrays or the current occurrence of an array.

Panther writes the send data to a temporary buffer, or bundle, which you can optionally name. Panther by default maintains up to ten named and unnamed bundles; the number of available bundles can be changed by setting the `max_bundles` property. If you omit a bundle name, Panther writes the data to an unnamed bundle; this data is accessed by the next `receive` command that omits the *`bundleName`* argument or specifies it as the empty string.

The bundle retains no information about its data sources. `receive` gets data in the order as it was sent. For example, the following `send` statement sends to an unnamed bundle the value in `credit_acctno`, the value `1000`, and all values in occurrences of the array `credit`. The `receive` command receives this data in the same order:

```
send DATA credit_acctno, 1000, credit
receive DATA acctno, amount, references
```

See Also

`receive`

service_call

Initiates a service call from a client agent

Synopsis

```
service_call serviceName ([requestMsg] [[,] replyMsg])
    [callOption]...
```

Arguments

`serviceName`

Any JIF-defined service, specified as a variable or quoted string. The service name can be up to 15 characters long.

`requestMsg`

Message data supplied by this client to the server, where *`requestMsg`*'s data type is defined in the JIF's definition of this service. For more information on message data types, refer to "Service Messages and Data Types" on page 5-15 in *JetNet/Oracle Tuxedo Guide*.

`replyMsg`

Specifies the variables to receive the data that the service returns to the client. The format of the return data is specified in the JIF's definition of the service. If the service call also specifies request message data, separate the two messages with a comma. For more information on message data types, refer to "Service Messages and Data Types" on page 5-15 in *JetNet/Oracle Tuxedo Guide*.

`callOption`

One or more options that control service call behavior. Each call option can be set through the service's JIF definition; unless the command specifies otherwise, the service call uses the JIF settings.

`service_call` call options always have precedence over their corresponding JIF settings. You set call options through these key words:

`ASYNC`

Specifies to let the client resume processing immediately after issuing the call without waiting for a response. If you omit this option, all processing on the client is suspended until the service call returns.

`NOTIMEOUT`

If service calls are blocked (the `tp_block` property is set to Yes), this option overrides the blocking timeout and blocks the call indefinitely.

`NOREPLY`

Use only in `ASYNC` mode to inform Panther that this service call expects no reply, so there is no need to poll for one.

Note: From the client's perspective, Panther makes no attempt to poll for a reply or to receive one; however, Panther might quietly poll for a reply if required by the middleware to end the client-server connection.

The following restrictions apply to the use of `NOREPLY`:

- (Oracle Tuxedo only) The `NOREPLY` option cannot be used with a service request that is part of a transaction. To remove a service request from an active transaction use the `OUTSIDE_TRANSACTION` option.
- The service call specifies no reply message data.

When the `NOREPLY` option is specified, a `post_request` event is generated directly after the `service_call` command is executed.

`OUTSIDE_TRANSACTION` (Oracle Tuxedo only)

Specifies to execute the service call independently of the active transaction, if one exists. Use this option in order to prevent events generated by this request from being affected by commits or rollbacks of the current transaction. If you specify this option, transaction-level exception and unload handlers are not executed when their corresponding events are generated.

`EXCEPTION_HANDLER` *`handler`*

Specifies an exception handler to install at the request scope, where handler is a Panther variable or a string. This handler handles any exceptions that result from the request or its response.

The handler is installed just before service invocation, that is, after all parsing, interpretation, and validation of the command has occurred. For more information on exception events and handlers, refer to “Exception Handlers” on page 6-11in *JetNet/Oracle Tuxedo Guide*.

`UNLOAD_HANDLER` *`handler`*

Specifies an unload handler to install at the request scope, where handler is a Panther variable or a string. This handler handles any unload events that might result from receiving the service's response. The service must be called synchronously (without the `ASYNC` option).

The handler is installed just before service invocation. For more information on unload events and handlers, refer to "Unload Handlers" on page 6-29 in *JetNet/Oracle Tuxedo Guide*.

`PRIORITY` *`priority`*

A signed or unsigned integer that sets the priority for *`serviceName`*. If unsigned, priority overrides this service's predefined priority; if signed, priority is added or subtracted from the predefined priority. In both cases, a service's priority level must be between 1 and 100. If you omit this option, the middleware uses the priority that is set in the JIF or (under Oracle Tuxedo), in the `TUXCONFIG` configuration file.

Environment

JetNet, Oracle Tuxedo

Scope

Client, Server

Description

The `service_call` command invokes a service request that can be issued by a client or a server. A server that requests a service acts in the role of a client. The JIF is accessed at runtime to determine predefined service attributes such as message data type.

Message Data

`service_call` can specify zero to two messages that enable exchange of data between the calling client and the server that processes the requested service. Depending on how the JIF defines the service's transport method, message data can have one of these forms:

- ()—No data sent or received.
- (*`message`*)—Data is sent in only one direction, either to or from the server, as specified in the JIF's service definition.
- (*`requestMsg`*, *`replyMsg`*)—The server receives data along with the client request (*`requestMsg`*), and returns data to the client (*`replyMsg`*)

The JIF also defines each message's data type. For more information, refer to "Service Messages and Data Types" on page 5-15 in *JetNet/Oracle Tuxedo Guide*.

Appending Transaction Manager Data to Messages

If the JIF defines a service to use the transaction manager, any data that the transaction manager needs for a database operation or that it returns is automatically appended to the corresponding request or reply message. For example, service `customer_s` specifies `Select` as its transaction type. When that service is called, the middleware appends to the request message any data that the transaction manager needs to construct a `SELECT` statement. When the service returns, the middleware appends the query results to the reply message.

When `service_call` calls a service that uses the transaction manager, the command must always specify one or both messages, according to the transaction's type. Thus, a service that specifies `Select` as its transaction type must be called with both messages; a service definition that specifies `Delete` as its transaction type can be called with only a single (request) message.

If a `service_call` command has no message data of its own, the command must use the default mapping format for one or both messages: `{...}`. For example, a `service_call` command can invoke the `customer_s` service as follows:

```
service_call "customer_s" ({...}, {...})
```

Synchronous and Asynchronous Modes

By default, a service call is issued in synchronous mode—that is, processing on the client is suspended until it receives a reply from the service. You can specify asynchronous mode for a service call through the `ASYNC` option. In this case, client processing continues without waiting for a response from the service.

Asynchronous processing might be desired if a client transaction includes several service calls that can be executed simultaneously. For example, a bank account transfer procedure requiring a debit to one account and a credit to another might be performed simultaneously as one transaction. If there is an error in the execution of one service, the entire transaction can be rolled back (Oracle Tuxedo only).

Asynchronous service calls can be issued by clients and servers, with these differences:

- For clients, Panther polls continuously for a reply from the service. This guarantees that requested data is returned when the service completes.
- Servers do not poll for replies to their own service calls, so a server that expects a reply to an asynchronous service call must use the wait command.

Service Call Event Stream

When a client agent initiates a service call, events are raised depending on the `service_call` options used. Figure 2-1 shows the sequence of events as the service call is processed.

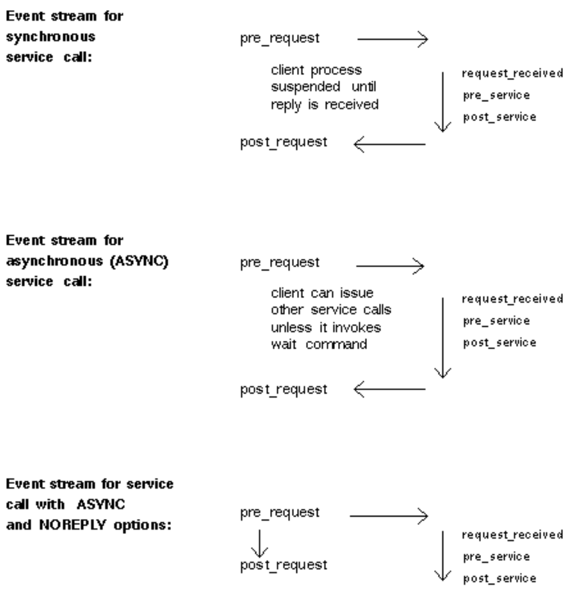

Event stream for synchronous service call:
pre_request
client process suspended until reply is received
post_request
request_received
pre_service
post_service
Event stream for asynchronous (ASYNC) service call:
pre_request
client can issue other service calls unless it invokes wait command
post_request
request_received
pre_service
post_service
Event stream for service call with ASYNC and NOREPLY options:
pre_request
post_request
request_received
pre_service
post_service

Figure 2-1 Event stream generated by a service_call

Refer to Chapter 6, "JetNet/Oracle Tuxedo Event Processing," in *JetNet/Oracle Tuxedo Guide* for information on middleware-related events, and how handlers can be used to customize your application's response to specific service calls.

The following scenarios illustrate a variety of ways of using `service_call`:

Scenario 1: If service `INQUIRY` is defined in the JIF to allow only incoming `JAMFLEX` data, you can pass the string `"Brontis"` to the service as follows:

```
service_call "INQUIRY" ({"Brontis"})
```

The following code calls the same service in asynchronous mode, passing the content of variable `name` to the service:

```
service_call "INQUIRY" ({name}) ASYNC
```

Scenario 2: If service `GET_NAME` is defined in the JIF to allow only outgoing `JAMFLEX` data, you can designate variable `name` to receive string data from the service as follows:

```
service_call "GET_NAME" ({name})
```

This code also calls the GET_NAME service, this time specifying an unload handler (name_unload) to unload the data:

```
service_call "GET_NAME" ({name}) UNLOAD_HANDLER "name_unload"
```

Scenario 3: The four examples illustrate how the service DEPOSIT is called. The service can be defined in the JIF to use JAMFLEX buffer types for messages.

- Call the DEPOSIT service and designate the content of the variables ACCOUNT_ID and AMOUNT as IN parameters, and designate the content of MESSAGE and ACCOUNT_BAL as the data to receive back from the service:

  ```
  service_call "DEPOSIT" ({ACCOUNT_ID, AMOUNT},\
     {MESSAGE, ACCOUNT_BAL})
  ```

- Call the DEPOSIT service and map local variable id to ACCOUNT_ID and local variable amt to AMOUNT in asynchronous mode with the NOREPLY option:

  ```
  service_call "DEPOSIT" ({ACCOUNT_ID=id, AMOUNT=amt})\
     ASYNC NOREPLY
  ```

- Call the DEPOSIT service and map the local variable id to ACCOUNT_ID and local variable amt to AMOUNT, and map the receiving MESSAGE into the local variable msg, and ACCOUNT_BAL into the local variable bal:

  ```
  service_call "DEPOSIT" ({ACCOUNT_ID=id, AMOUNT=amt},\
     {MESSAGE=msg, ACCOUNT_BAL=bal})
  ```

- Call the DEPOSIT service with a relative priority of +50:

  ```
  service_call "DEPOSIT" PRIORITY +50 \
     ({ACCOUNT_ID, AMOUNT},{MESSAGE, ACCOUNT_BAL})
  ```

Under Oracle Tuxedo, the same service can be defined to use FML buffer types. This requires that the FML file contain the following entries for FML fields for the bankapp database:

Name	**Number**	**Type**	**Comments**
ACCOUNT_ID	101	long	Account No.
ACCOUNT_BAL	117	float	Account balance
AMOUNT	111	float	Transaction amount
MESSAGE	126	string	Message text

The screen variables have identical names to the FML fields.

Property Settings Table 2-1 shows which application properties are affected when `service_call` executes:

Table 2-1 Properties set by service_call

Property	**Value**
`tp_return`	Set to the `callid` for the associated request, provided the command progressed as far as the `pre_request` event.
`tp_svc_return`	Set to the service return value. This property reflects the `CODE` value specified by the service when it executes a `service_return` command. It is only set if the service actually provides a return value and if the return type is a scalar or an array of scalars. Otherwise, the variable is not changed.
`tp_svc_outcome`	Set to the service return status when the service has returned its last message to the client. This status is the indication of application success or failure, if the service provides such to the middleware.

Exceptions `service_call` can generate the following exceptions:

Exception	**Severity**	**Cause**
`TP_IDENTIFIER_TRUNCATED`	`TP_WARNING`	Length of *`serviceName`* exceeds 15 characters; it is truncated to the maximum length permitted by the middleware.
`TP_INVALID_ARGUMENT`	`TP_COMMAND`	Argument is invalid.
`TP_INVALID_ARGUMENT_LIST`	`TP_COMMAND`	Argument list is invalid.
`TP_INVALID_BUFFER`	`TP_ERROR`	Buffer received has unexpected type.
`TP_INVALID_BUFFER_VERSION`	`TP_ERROR`	Buffer received is of an incompatible version.

Exception	Severity	Cause
`TP_INVALID_CONNECTION`	`TP_COMMAND`	Connection does not exist.
`TP_INVALID_OPTION_VALUE`	`TP_COMMAND`	Option value is invalid.
`TP_INVALID_SERVICE`	`TP_COMMAND`	Service is invalid or was not advertised.
`TP_MONITOR_ERROR`	`TP_COMMAND`	The middleware raises an error.
`TP_NO_OUTSIDE_TRANSACTION`	`TP_COMMAND`	There is no current transaction.
`TP_NONTRANSACTIONAL_SERVICE`	`TP_COMMAND`	Service cannot be executed within a transaction.
`TP_REQUEST_LIMIT`	`TP_COMMAND`	Number of outstanding requests has exceeded the limit.
`TP_SERVICE_FAILED`	`TP_INFORMATION`	Service returned failure status.
`TP_SERVICE_PROTOCOL_ERROR`	`TP_ERROR`	Service has violated protocol and has been abnormally terminated.
`TP_TIMEOUT`	`TP_MESSAGE`	Action terminated due to a timeout condition.

Example

The following procedure `init_atm` initiates the client identification process when a user logs on as an ATM customer. FML buffers are used in a call to the `VAL_PIN` service. For the server end of this process, refer to the receive command.

```
proc init_atm ()
vars message

...
@app()->hdl_exception = "exc_hand"
@app()->hdl_jif_changed = "jchhandc"

client_init client last_name user "Customer" \
   notification poll
if ((@app()->tp_severity) > TP_WARNING)
{
   // initiating a connection was unsuccessful
   message = @app()->tp_exc_msg
   msg quiet message
   ...
```

```
    return 0
}

// validate PIN given by the customer
service_call service "VAL_PIN" ({last_name, pin}, \
    {message, owner_ssn = user_info})

// check if validation was not successful
if ((@app()->tp_severity > TP_WARNING) || \
     (@app()->tp_svc_outcome == TP_FAILURE))
{
    msg quiet message
    client_exit
    ...
}
...
return 0
}
```

See Also `receive`, `service_cancel`, `service_forward`, `wait`

service_cancel

Cancels an outstanding service request

Synopsis `service_cancel [`*`services`*`]`

Arguments *`services`*

Specifies which services to cancel:

`CALL` *`callid`*

Cancel the specific service request identified by *`callid`*.

`ALL [(`*`callid`*`...)]`

An unqualified `ALL` cancels all outstanding service requests; if followed by a list of service call identifiers (*`callid`*), `ALL` cancels the specified service requests. Enclose the list of call identifiers with parentheses.

Environment JetNet, Oracle Tuxedo

Scope Client, Server

Description The `service_cancel` command cancels the specified service requests. An unqualified `service_cancel` cancels the most recent asynchronous request. You can cancel one or more requests with the `CALL` and `ALL` options. For example, this statement cancels the service request identified by the Panther variable `call_id`:

```
service_cancel CALL call_id
```

This statement cancels all outstanding service requests:

```
service_cancel ALL
```

You can cancel both synchronous and asynchronous service calls. Canceling a call does not stop it from running; however, it does stop the reply. If the canceled service is part of a transaction under Oracle Tuxedo, the cancellation should be accompanied by a rollback to ensure the integrity of an XA resource.

`service_cancel` sets the `tp_return` property to the number of service calls canceled.

Exceptions

`service_cancel` can generate the following exceptions:

Exception	Severity	Cause
`TP_ALREADY_CANCELED`	`TP_WARNING`	The request has already been canceled.
`TP_EXPLICIT_CANCEL`	`TP_INFORMATION`	Each request is canceled (informational only).
`TP_INVALID_CALL`	`TP_COMMAND`	Service call is unidentifiable.
`TP_INVALID_CONNECTION`	`TP_COMMAND`	No connection to the middleware exists.
`TP_MONITOR_ERROR`	`TP_COMMAND`	An error is reported from the middleware.
`TP_NO_OUTSTANDING_CALLS`	`TP_INFORMATION`	No service calls are outstanding; that is, a specified service call has already completed.

Note: A call is considered outstanding as soon as its associated `pre_request` event has been raised, and is considered complete as soon as its associated `post_request` event has been generated. After each request has terminated, a `post_request` event is generated. For more information, refer to "Pre_request and Post_request Events" on page 6-20 in *JetNet/Oracle Tuxedo Guide*.

See Also

`receive`, `service_call`, `service_forward`, `service_return`

service_forward

Forwards service request data to another service

Synopsis

`service_forward` *`serviceName`* `[(` *`message`*`)] [ PRIORITY` *`priority`*`]`

Arguments

`serviceName`
: The JIF-defined service to get the forwarded service request. This argument must be either a variable containing the name of a service or a quoted string.

`message`
: Message data to relay from the original service to *`serviceName`*. You can omit this argument only if both services have identical message data definitions; in this case, *`serviceName`* gets the original service's message data. To forward no data, supply an empty argument `()`.

  A supplied message must conform to the JIF definition for *`serviceName`*. For more information on message data types, refer to “Service Messages and Data Types” on page 5-15 in *JetNet/Oracle Tuxedo Guide*.

`PRIORITY` *`priority`*
: A signed or unsigned integer that sets the priority for *`serviceName`*. If signed, *`priority`* overrides any priority set in this service's JIF definition; if signed, *`priority`* is added or subtracted from the default priority for all services, set in the middleware configuration file. In both cases, a service's priority level must be between `1` and `100`.

Environment JetNet, Oracle Tuxedo

Scope Server

Description

The `service_forward` command passes the current service request to another service for processing. After the service is forwarded, the current service routine terminates immediately, thereby terminating processing of the current request by this agent. All properties are restored to normal default settings after execution of `service_forward`.

For example, this JPL forwards credit data from service `TRANSFER` to the `DEPOSIT` service:

```
// Service TRANSFER
...

receive ARGUMENTS ({acct_id_deb, amount_deb, \
   acct_id_cred, amount_cred})

...
service_forward "DEPOSIT" ({acct_id_cred, amount_cred})
```

If an exception of severity `TP_ERROR` or greater occurs before the forward operation begins, the service request is not forwarded. Instead, the service returns with `TP_FAILURE`. If the service is part of a transaction, the transaction is marked for abort-only and is not committed; it can only be rolled back explicitly by the user.

Services are typically forwarded with their original message data by omitting the message argument. This implicit passing of data is valid only if both services define their input message arguments identically.

Exceptions `service_forward` can generate the following exceptions:

Exception	Severity	Cause
`TP_INVALID_ARGUMENT`	`TP_COMMAND`	Data is passed by omitting (`argList`), but the services do not have identical input buffers defined
`TP_INVALID_ARGUMENT_LIST`	`TP_COMMAND`	The return parameters of the services are not identically defined
`TP_INVALID_OPTION_VALUE`	`TP_COMMAND`	The absolute priority value was out of range (must be from `1` to `100`)
`TP_INVALID_SERVICE`	`TP_COMMAND`	Service specification is invalid

See Also `receive`, `service_call`, `service_return`

service_return

Returns from a service request invocation

Synopsis `service_return [`*`returnStatus`*`] ([`*`message`*`]) [CODE `*`returnCode`*`]`

Arguments *`returnStatus`*

Specifies the service return status with one of these values:

- `SUCCESS`—The service succeeded.
- `FAILURE`—The service failed.
- `EXIT`—The service failed and the server executable should terminate.

Under Oracle Tuxedo, the middleware uses the service's return status to determine whether a transaction is successful. If any service that participates in a transaction returns with `FAILURE` or `EXIT`, the transaction is marked as abort-only. The transaction cannot be committed and must be aborted explicitly by the user.

For more information about transactions, refer to `xa_begin` and `xa_end`

`message`

Specifies the message data to return to the invoking agent. This provides a mapping from Panther variables to the return arguments specified in the service call. Always enclose the data in message in parentheses even if no data is specified. The message format is determined by the transport method specified for the service in the JIF: `JAMFLEX`, `STRING` or an `FML` buffer.

For more information on message data, refer to “Service Messages and Data Types” on page 5-15 in *JetNet/Oracle Tuxedo Guide*.

`CODE` *`returnCode`*

An integer to associate with the return. This code can be examined by the client agent; it is ignored by the middleware. If omitted, the default return value is `0`.

Environment JetNet, Oracle Tuxedo

Scope Server

Description

The `service_return` command returns data from a service request and indicates to the middleware whether the service was performed successfully. `service_return` and `service_forward` are the only commands by which a service can be explicitly completed. If a service routine terminates without using either of these, Panther completes the service automatically as if the `service_return` command were invoked with an empty argument list, with a *`returnCode`* equal to the return value from the service routine, and with a *`returnStatus`* of either `FAILURE` (if the service return value is negative), or `SUCCESS` (if the return value is `0` or greater).

All properties are restored to default settings after execution of `service_return`, since the service routine will have terminated.

On the client side, the `tp_svc_outcome` property is set to the service's return status, where it can be inspected by the client. The `tp_svc_return` property is set to the return code.

The `tp_tran_status` property is set to `TP_WILL_ABORT` if the service returned with a `FAILURE` or `EXIT` status.

Appending Transaction Manager Data to Reply Messages

If the JIF defines a service to use the transaction manager, any data that the transaction manager returns is automatically appended to the message argument. For example, service `customer_s` specifies `Select` as its transaction type. When that service returns, the middleware appends the query results to the reply message.

If a transaction returns data to the client, the `service_return` call must specify a message argument. Thus, a service that specifies `Select` as its transaction type returns (if successful) with the results of the database query, so `service_return` must specify a message argument; conversely, a `Delete` transaction returns no data to the client, so a `service_return` command that returns from a service of that transaction type can omit its message argument.

If a transaction returns data and the `service_return` command has no message data of its own, the command must use the default mapping format for its message arguments: `{...}`. For example:

```
service_return SUCCESS ({...})
```

Exceptions

`service_return` can generate the following exceptions:

Exception	Severity	Cause
`TP_INVALID_ARGUMENT`	`TP_COMMAND`	Invalid arguments are supplied

Exception	Severity	Cause
`TP_INVALID_ARGUMENT_LIST`	`TP_COMMAND`	Invalid arguments are supplied
`TP_INVALID_CONTEXT`	`TP_COMMAND`	Command is used outside of the service, or out of context

If, during processing of a `service_return` command, an exception of severity `TP_ERROR` or greater occurs before control returns to the middleware, the service is completed with an error return status. Panther ensures an error return status by using a `FAILURE` status unless the `service_return` command explicitly specifies an `EXIT` status, in which case an `EXIT` status is used.

Example

The following code is the `DEPOSIT` service. The client side of this process is shown in the `xa_end` command.

```
proc deposit()
// service DEPOSIT
vars amount
receive arguments ({account_id, amount})
call sm_tm_command("SELECT")

if (!@dmrowcount)
{
    service_return failure ({message = "Invalid account."})
}
// need to check that overflow will not happen when
// depositing
if (account_balance + amount > max_balance)
{
    service_return failure \
        ({message = "Balance field overflow"})
}
account_balance = account_balance + amount
call sm_tm_command("SAVE")
service_return ({message = @tpi_null, \
    balance = account_balance})
```

See Also

`service_call`, `service_forward`, `xa_begin`, `xa_end`

subscribe

Subscribes to an event managed by the Oracle Tuxedo event broker

Synopsis

```
subscribe EVENT eventName NOTIFICATION notificationSpec
    ENQUEUE enqueueSpec [OUTSIDE_TRANSACTION] [PERSIST]
    [FILTER rule] [NOTIMEOUT]
```

Arguments

`EVENT` *`eventName`*

The name of an event. *`eventName`* is any regular expression containing up to 255 characters. For information about regular expressions, refer to `recomp()` in the *Oracle Tuxedo Reference Manual*.

`NOTIFICATION` *`notificationSpec`*

Method of notification to the subscriber when the event is posted, formatted as follows:

```
{SERVICE serviceName | ENQUEUE enqueueSpec}
```

`SERVICE` *`serviceName`*

Notification is done via a call to the service *`serviceName`*. The event broker calls *`serviceName`* to notify the agent of the event.

`ENQUEUE` *`enqueueSpec`*

Notification is done via enqueuing a message to a reliable queue. *`enqueueSpec`* has this format:

```
QSPACE queueSpace NAME queueName [queueOption
[queueOption]...]
```

`QSPACE` *`queueSpace`*

The name of the queue space to which the queue belongs.

`NAME` *`queueName`*

The name of the queue.

`queueOption`

One or more of the enqueuing options listed in the “Enqueue Options” section.

`OUTSIDE_TRANSACTION`

Specifies that event notifications are dispatched outside of the current transaction. If this option is not used, the default behavior is notification within the current transaction.

`PERSIST`

Maintains the event subscription regardless of any error situation. By default, subscriptions are deleted when a resource is not available to an event poster.

`FILTER` *`rule`*

A filter rule to apply when the event broker determines that the subscriber should be notified of an event. rule is a string expression of up to 255 characters. The rule is applied to the message data of the event posting. This option is available for `FML` and `STRING` types only.

`NOTIMEOUT`

Specify that the execution of this command is unaffected by the blocking timeout.

Enqueue Options

- `BEFORE_MSGID` *`msgId`*—Put the message ahead of the message with Oracle Tuxedo message identifier *`msgId`*.
- `CORRID` *`corrID`*—A correlation ID to associate with the enqueued message, a string of up to 32 characters. The value is maintained across all queues, so any reply or failure message associated with the queued message can be identified.
- `DQTIME` *`dequeueTime`*—Specifies when to make the message available for dequeuing. If you omit this option, the message can be dequeued immediately.

  The server dequeues the message and calls the appropriate service, if it is monitoring the queue. *`dequeueTime`* can be a relative time (time elapsed after the message is enqueued) or an absolute time. An absolute time must be greater than January 1 1970 00:00:00 UTC. In either case, Panther can dequeue the command only after the specified amount of time has elapsed.
  A relative *`dequeueTime`* can be specified in this format:

  `"[ +`*`days hours`*`::`*`minutes`*`::]`*`seconds`*`"`

  Seconds are required; minutes, hours, or days (space delimiter between days and hours) can also be specified. If more than seconds is specified, the + symbol and the quotation marks are mandatory. If only seconds are specified, both are optional.

  Note: JPL's colon preprocessor expands colon-prefixed variables. To prevent expansion of variables that contain colons, prefix literal colons with another colon (`::`) or a backslash (`\:`).

  An absolute *`dequeueTime`* can be specified in one of these ways:

  - The value from a widget having Date/Time property values.

- A date/time string in this format: *`"mm/dd/yy HH::MM"`*

  The `DQTIME` option is valid only if the queue has been configured for time-based ordering; that is, queue order is set to time. For more information, refer to the *Oracle Tuxedo /Q Guide* of the Oracle Tuxedo SDK in your Oracle Tuxedo documentation.

- `FAILUREQ` *`queue`* `| NOFAILUREQ`—Specify a failure queue to which failure responses can be enqueued, or use `NOFAILUREQ` if no failure message is necessary. If this option is not used, the JIF is checked for a failure queue.
- `FRONT`—Place the message at the head of the queue. This option can only be used if the queue has been configured (when it was created) for `out_of_order` enqueuing, with this attribute set to top.
- For further information on `out_of_order` enqueuing, refer to the *Oracle Tuxedo /Q Guide* of the Oracle Tuxedo SDK in your Oracle Tuxedo documentation.
- `PRIORITY` *`priority`*— Establish a priority value for the message. The valid range is 1 to 100; the default is 1. This option will only have effect if the queue was created using priority as a queue-ordering parameter. The larger the value of priority, the higher the priority.
- For further information on priority enqueuing, refer to the *Oracle Tuxedo /Q Guide* of the Oracle Tuxedo SDK in your Oracle Tuxedo documentation.
- `RCODE` *`returnCode`*—Specify an integer-value return code to be made available to the application. The return is handed to the reply queue from the service which replies to the message.
- `REPLYQ` *`queue`* `| NOREPLYQ`—Specify a reply queue to which replies can be enqueued, or use `NOREPLYQ` if no reply message is wanted. If this option is not used, the JIF is checked for a reply queue.

Environment Oracle Tuxedo

Scope Client, Server

Description The `subscribe` command permits agents to subscribe to events managed by the event broker. Once an event is posted via the `post` command, subscribers to the event are notified in the manner determined by the arguments to this command.

When the subscribing agent is a client, event notification is done via an unsolicited message. A client can receive unsolicited notifications only if it has appropriate message handling. Refer to the client_init and receive commands for information on how to permit clients to receive unsolicited messages.

For servers subscribing to events, there are two methods of notification: notification by service call and notification by message queuing.

Before notification is initiated, the event broker, after successfully matching the event to its potential subscribers via the `EVENT` *`eventName`*, applies the subscribers filter rule if one was used. If the data passes through the filter rule, the subscriber is notified via the method selected with *`notificationSpec`*.

Successful execution of the subscribe command results in a unique subscription ID, which can be accessed from the `tp_return` property. If the command fails, `tp_return` is set to `TP_FAILURE`.

For additional information on message queuing, refer to "Reliable Queues" on page 8-11 in *JetNet/Oracle Tuxedo Guide* and refer to your Oracle Tuxedo System /Q documentation.

Filter Rule Syntax The filter rule is contained in a string of up to 255 characters. The rule format is specific to the type of event message data—`FML` or `STRING`—of the event's data to which it is applied.

`FML` filters can be built from primary expressions, regular expressions, and operators. A `STRING` filter must be in the form of a regular expression. For information about regular expressions, refer to `recomp`() in the *Oracle Tuxedo Reference Manual*; for information about operators and primary expressions, refer to the Oracle Tuxedo FML Guide.

Exceptions

Because `subscribe` uses message queues, it can raise some of the same exceptions as the `enqueue` command.

Exception	**Severity**	**Cause**
`TP_INVALID_COMMAND_SYNTAX`	`TP_COMMAND`	Command syntax is invalid.
`TP_INVALID_OPTION_VALUE`	`TP_COMMAND`	Priority value is not between 1 and 100 or an invalid time value was specified
`TP_INVALID_QUEUE`	`TP_COMMAND`	Queue is not declared in the JIF

Exception	Severity	Cause
`TP_QUEUE_SPACE_NOT_IN_JIF`	`TP_COMMAND`	Queuespace not found in the JIF.

Example

```
// Client
// The client will receive an unsolicited message
// along with any data posted with the event.

subscribe EVENT "user*" FILTER "*something*"

// Server
// The server will receive notification via a call
// to the service "svc1"

subscribe EVENT "user*" FILTER "*something" \
   NOTIFICATION SERVICE "svc1" OUTSIDE_TRANSACTION

// Server
// The server will receive notification via enqueuing the
// message to the queue "queue1" in queue space "qspace1."

subscribe EVENT "user*" NOTIFICATION ENQUEUE QSPACE \
   "qspace1" NAME "queue1" PRIORITY 5 REPLYQ "rqueue1" NOTIMEOUT
```

See Also

`enqueue`, `dequeue`, `post`, `unsubscribe`

switch

Execute different statements based on the value of an expression

Synopsis

```
switch testExpression

case caseExpression[, caseExpression] ...
   statementBlock

[case caseExpression[, caseExpression] ...
   statementBlock]
...
[default
   defaultStatementBlock]
```

Arguments

`testExpression`
: Specifies a value to be tested by `case` statements that follow the `switch` statement. *`testExpression`* can be any expression. For more information on expressions, refer to “Expressions” on page 19-55 in *Application Development Guide*.

`caseExpression`
: Each *`caseExpression`* is evaluated and compared with *`testExpression`*. If they are equal, the *`statementBlock`* following the `case` statement is executed

`statementBlock`
: One or more statements that JPL executes if the preceding `case` statement finds a match. If `statementBlock` has more than one statement, enclose the block with open and close blocking characters `{}` on the lines before and after.

`defaultStatementBlock`
: *`defaultStatementBlock`* is or more statements that JPL executes if none of the preceding `case` statements find a match. If *`defaultStatementBlock`* has more than one statement, enclose the block with open and close blocking characters `{}` on the lines before and after it.

Description

The `switch` command does conditional execution of other JPL statements. It is new to Panther 5.40. Each `switch` can be followed by one or more `case` commands to create a chain of conditional processing. JPL checks the value of each *`caseExpression`* in the chain until finds a value that is equal to *`testExpression`*;

JPL then executes the statement block that follows the `case` command and exits the chain. If none of the `case` expressions in a `switch` chain match *`testExpression`* and if the chain ends with the `default` command, JPL executes the following statement block. If the `switch` chain omits a `default` command, JPL simply exits the chain and continues execution with the next statement after it.

Example

```
//Determine a person's sex, based on personal title.
switch title

case 'MR'
   sex = 'Male'

case 'MS', 'MRS', 'MISS'
   sex = 'Female'

default
{
   sex = 'Unknown'
   msg err_reset 'Please supply a title.'
}
```

unadvertise

Unadvertises services from a server

Synopsis

```
unadvertise {ALL | SERVICE service | GROUP serviceGroup}
```

Arguments

`ALL`

Unadvertise all advertised services.

`SERVICE` *`service`*

Unadvertise *`service`*. The service name can be up to 15 characters in length.

`GROUP` *`serviceGroup`*

Unadvertise all services advertised via *`serviceGroup`*. If services that are contained in a group are advertised individually or through the `ALL` option, this option has no effect on them and no exception event is raised.

Environment

JetNet, Oracle Tuxedo

Scope

Server

Description

The `unadvertise` command unadvertises services previously advertised via the `advertise` command. You can unadvertise individual services, services from a group, or all services. For example, this statement unadvertises all services:

```
unadvertise ALL
```

The following statement unadvertises service transfer:

```
unadvertise SERVICE "transfer"
```

The `tp_return` property is set to the number of services successfully unadvertised.

Exceptions

The `unadvertise` command can generate these exceptions:

Exception	Severity	Cause
`TP_INVALID_SERVICE`	`TP_COMMAND`	`SERVICE` is used and the service has not been advertised
`TP_NO_SERVICES_ADVERTISED`	`TP_WARNING`	`ALL` is used and no services have been advertised.

See Also `advertise`

unload

Frees the memory allocated for a public module

Synopsis `unload` *`moduleName`* `[` *`moduleName`*`]...`

Arguments *`moduleName`*

The name of the public module to remove from memory. This parameter is case-sensitive; the name must exactly match the name used in the public command.

Description The `unload` command releases the memory used to hold one or more JPL modules previously loaded into memory as public modules. After you unload *`moduleName`*, subsequent calls to that module read it from disk, the memory-resident list, or an open library. The named procedures in that module are inaccessible to the application except through its unnamed procedure.

Avoid unloading a module that is undergoing execution.

Unloading a module not currently in memory does not cause an error.

Example

```
// load a file, call it in a loop,
// then unload it after exiting the loop
public validname
for i = 1 while i < 11 step 1
  call validname (name[i])
unload validname
```

See Also `public`

unload_data

Writes data received remotely via the middleware to target Panther variables

Synopsis `unload_data [CLEAR_ONLY]`

Arguments `CLEAR_ONLY`

Clears target Panther variables and does not attempt to write any data to them.

Environment JetNet, Oracle Tuxedo

Scope Client, Server

Description The `unload_data` command writes to Panther variables data received from a re mote agent via middleware processing. The data is written into the variables according to the mapping established in a `service_call` or `receive` command invocation. This command is used in the default Panther unload handler. Use this command if you write your own unload handler.

An unload event is triggered whenever data is received from an external source whose contents can be written to a set of target Panther variables. For clients, external sources of data include `service_return` data from services requested and unsolicited messages. For servers, external sources of data are data sent from a client as part of a service request, such as arguments. The unload operation begins by first clearing all target Panther variables, even if there is actually no data to unload. The `CLEAR_ONLY` option ensures that the target Panther variables are cleared of old data even in the event of an error, and prevents new data from being written.

Example For example, the following unload handler `user_unload_handler` calls an unqualified `unload_data` command:

```
proc user_unload_handler (callid, msg_source, method)
if (method == TP_ASYNCHRONOUS)
{
   msg emsg "Data has arrived!"
}

// Do the unload
unload_data
return
```

The following client JPL installs this unload handler. The service reverse takes a string as input and reverses the characters for output:

```
// Client code:
vars receive_string
@app()->hdl_unload = "user_unload_handler
service_call "reverse" ("hello world", receive_string)
msg setbkstat "String unloaded", receive_string
```

Exceptions

If `unload_data` is called other than from an unload handler or subordinate procedures, a `TP_INVALID_CONTEXT` exception of severity `TP_COMMAND` is raised.

See Also

`receive`, `service_call`, `service_return`

unsubscribe

Unsubscribes to an event managed by the Oracle Tuxedo event broker

Synopsis

```
unsubscribe {ALL | SID subscriptionId} [NOTIMEOUT]
```

Arguments

`ALL`
: Removes all event subscriptions, except persistent subscriptions—that is, those whose subscribe command used the `PERSIST` option.

`SID` *`subscriptionId`*
: Unsubscribe from the event specified by *`subscriptionId`*, even if the subscription is persistent. You obtain *`subscriptionId`* at the time the event is subscribed to.

`NOTIMEOUT`
: Execution of the command disregards the blocking timeout; however, transaction timeouts remain in effect.

Environment Oracle Tuxedo

Scope Client, Server

Description The `unsubscribe` command removes event subscriptions from Oracle Tuxedo's event broker. The command lets you unsubscribe all events, or specific or persistent events.

The `tp_return` property is set to number of subscriptions canceled after successful execution of the command.

See Also `post`, `subscribe`

vars

Declares JPL variables

Synopsis `vars` *`varSpec`* `[,` *`varSpec`*`]...`

Arguments

`varSpec` Specifies the variable's name and properties as follows:

`varName` `[[`*`numOccurs`*`]] [(`*`size`*`)] [=` *`initValue`*`]`

`varName`

The name of the variable, where *`varName`* is a string that contains up to 31 characters. JPL variable names can use any combination of letters, digits, or underscores, where the first character is not a digit. Panther also allows usage of two special characters, the dollar sign ($) and period (.).

[*`numOccurs`*]

Optionally declares *`varName`* as an array of *`numOccurs`* occurrences. The default number of occurrences is 1. For example the following statement declares dependents as an array of ten occurrences:

```
vars dependents[10]
```

(*`size`*)

Optionally specifies the number of bytes allocated for this variable; Panther allocates an extra byte for the terminating null character. The default size is 255 bytes. For example, the following statement declares the variable zip with a size of 10 bytes:

```
vars zip (10)
```

= *`initValue`*

Optionally initializes the variable to *`initValue`*, where *`initValue`* can be any constant, variable, or string or numeric expression. For example:

```
vars workweek = 5
vars avg_sale = @sum(sale_amt) / sale_amt->num_occurrences
vars name = fname##lname
```

If the variable is declared as an array, you can initialize its occurrences. For example:

```
vars ratings[5] = {"G", "PG", "PG-13", "R", "NC-17"}
```

Occurrence values are comma-delimited.

If no value is assigned, Panther initializes the variable to an empty string ("").

Description

The `vars` command creates one or more JPL variables. Panther executes `vars` statements as it encounters them at runtime. JPL's ability to reference a variable depends on the variable's scope and lifetime:

- Variables declared in a named procedure are known only to that procedure and remain in memory until the procedure returns.
- Variables declared in an unnamed procedure are accessible to all procedures in the module. These remain in memory until the module returns. Two exceptions apply: variables declared in a screen module's unnamed procedure remain in memory until the screen exits; variables in a public module's unnamed procedure remain in memory until the module itself is removed from memory.

Example

```
vars name(50), flag(1)
vars address[3](50), abbrevs[10]
vars zip(5) = "02138"
```

See Also

`global`

wait

Waits for service calls to return before processing resumes

Synopsis `wait [`*`serviceCalls`*`] [ TIMEOUT `*`timeout`*`]`

Arguments *`serviceCalls`*

Specifies which service calls must return before processing resumes. If no service calls are specified, processing is suspended until the current service call returns. Use one of these keywords and its options:

`FOR` [*`callid`*]
Processing resumes when service call *`callid`* returns.

`FOR_ALL` [*`callSpec`*]
Processing resumes when all the specified services return. If you omit *`callSpec`*, all outstanding service calls must return in order to resume processing. *`callSpec`* can have one of these formats:

- (*`callid`*[, *`callid`*]...)—A comma-delimited list of service call IDs, enclosed in parentheses, that specifies the service calls that must return before processing resumes.
- `TRANSACTION`—Processing resumes when all services associated with the current transaction return.

`FOR_ANY` [*`callSpec`*]
Processing resumes when one of the specified services return. If you omit *`callSpec`*, processing resumes when any outstanding service call returns. *`callSpec`* can have one of these formats:

- (*`callid`*, [*`callid`*]...)—A comma-delimited list of service call IDs, enclosed in parentheses. Processing resumes when any of these calls returns.
- `TRANSACTION`—Processing resumes when any service associated with the current transaction returns.

`TIMEOUT` *`timeout`*

Resume processing if the specified service calls do not return before *`timeout`* elapses. *`timeout`* is a quoted string or variable whose format can specify a relative time (time elapsed after the wait command is issued) or an absolute time. If no *`timeout`* is specified, all specified service calls must return before processing resumes.

A relative `timeout` can be specified in this format:

```
"[ +days hours::minutes::]seconds"
```

Seconds are required; minutes, hours, or days (space delimiter between days and hours) can also be specified. If more than seconds is specified, the + symbol and the quotation marks are mandatory. If only seconds are specified, both are optional.

Note: JPL's colon preprocessor expands colon-prefixed variables. To prevent expansion of variables that contain colons, you must prefix literal colons with another colon (`::`) or a backslash (`\:`).

An absolute `timeout` can be specified in one of these ways:

- The value from a Panther date/time field.
- A date/time string in this format: *`"mm/dd/yy HH::MM"`*

For example, the following `wait` command suspends processing for 10 seconds or until any call in the list has completed:

```
WAIT FOR_ANY (call_save_emp, call_save_dept) TIMEOUT 10
```

Environment JetNet, Oracle Tuxedo

Scope Client, Server

Description The `wait` command suspends processing of its caller pending completion of the specified service calls or elapse of the specified *`timeout`*. If no service calls are specified, processing is suspended until the current service call returns. Processing can resume whether or not the specified calls return with the requested service. When `wait` returns, it sets the `tp_return` property to the number of service calls that returned while processing was suspended.

The `wait` command activates the `exception`, `unload` and `post_request` handlers associated with any call that returns while the wait command executes.

In the following example, the client code issues two service calls, `WITHDRAWAL` and `BAL_INFO`, both asynchronous. The `wait` command includes the `FOR_ALL` option, which prevents the same client from engaging in any other activity until both service calls return:

```
vars err_msg
service_call service "WITHDRAWAL"( \
    acct_id,            \
    {err_msg, transact_id, post_date}) ASYNC

service_call service "BAL_INFO"( \
    acct_id,            \
    {message, cur_bal, avail_bal, cur_date}) ASYNC

wait FOR_ALL
...
```

Exceptions

`wait` can generate these exceptions:

Exception	Severity	Cause
`TP_INVALID_CALL`	`TP_COMMAND`	Invalid *`callid`*.
`TP_INVALID_CONNECTION`	`TP_COMMAND`	No connection to middleware.
`TP_INVALID_OPTION_VALUE`	`TP_COMMAND`	Invalid `TIMEOUT` specification.
`TP_NO_OUTSTANDING_CALLS`	`TP_INFORMATION`	All specified service calls returned.
`TP_TIMEOUT`	`TP_WARNING`	The `wait` command returned because the specified *`timeout`* elapsed.

See Also

`service_call`, `receive`

while

Repeatedly executes a block while a condition is true

Synopsis

```
while logicalExpr
    statementBlock
```

Arguments

logicalExpr

Specifies the condition that JPL uses to determine whether to reiterate execution of the while block.

statementBlock

One or more statements that JPL executes if *logicalExpr* evaluates to true. If *statementBlock* has more than one statement, enclose the block with open and close blocking characters {0} on the lines before and after.

Description

The `while` statement repeatedly executes a block of one or more statements as long as the value of *logicalExpr* is true. JPL evaluates *logicalExpr* before each iteration of the loop.

Example

```
// do do_proc as often as user wants
vars ans
ans = sm_message_box \
    ("Start processing?","",SM_MB_YESNO, "")
while ans
{
  call do_proc
  ans = sm_message_box \
    ("Repeat processing?","",SM_MB_YESNO, "")
}
```

When you construct a logical expression, take into account that JPL, unlike C, always fully evaluates a boolean expression. For example, the following while loop traverses a screen's fields by field number (ct) until the last field or the first modified field is reached:

```
vars ct
vars n_flds = @screen("@current")->numflds

while ct <= n_flds && @field_num(ct)->mdt == PV_NO
{
   ...
   ct = ct + 1
}
```

If all fields are unmodified, `ct` increments to one greater than `n_flds` on the last pass through the while loop, so the first condition evaluates to false; however, JPL also evaluates the second condition `@field_num(ct)`, which is invalid. Consequently, JPL issues an error message and stops execution of the remaining code.

See Also `break`, `for`, `next`

xa_begin

Starts a middleware transaction

Synopsis

```
xa_begin [ EXCEPTION_HANDLER handler, UNLOAD_HANDLER handler,
    TIMEOUT timeout]
```

Arguments

`EXCEPTION_HANDLER` *`handler`*

Specifies an exception handler to be installed for the duration of the transaction; use `NULL` if none is to be specified. For further information on exception events and handlers, refer to “Exception Events” on page 6-11 in *JetNet/Oracle Tuxedo Guide*.

`UNLOAD_HANDLER` *`handler`*

Specifies an unload handler to be installed for the duration of the transaction. The handler should control all unloading of transaction data to Panther target variables; use `@NULL` if none is to be specified.

For example, this command specifies the unload handler `myhandler`:

```
xa_begin UNLOAD_HANDLER "myhandler"
```

For more information on unload events and handlers, refer to “Unload Events” on page 6-28 in *JetNet/Oracle Tuxedo Guide*.

`TIMEOUT` *`timeout`*

Resume processing if the transaction is not complete before *`timeout`* elapses. If you omit this option, transaction processing continues without a time limit. Specify *`timeout`* with this format:

```
"[ +days hours::minutes::]seconds"
```

Seconds are required; minutes, hours, or days (space delimiter between days and hours) can also be specified. If more than seconds is specified, the + symbol and the quotation marks are mandatory. If only seconds are specified, both are optional.

Note: JPL's colon preprocessor expands colon-prefixed variables. To prevent expansion of variables that contain colons, you must prefix literal colons with another colon (`::`) or a backslash (`\:`).

For example, this command specifies a time interval of `30` seconds:

```
xa_begin TIMEOUT 30
```

The following command specifies a time interval of 3 hours:

```
xa_begin TIMEOUT "+3::00::00"
```

Environment Oracle Tuxedo

Scope Client, Server

Description The xa_begin command initiates a transaction to be performed on XA-compliant resource managers. Once initiated, a transaction must be completed by a call to either xa_commit, xa_rollback or xa_end. When a transaction is in progress, any service requests made to XA-compliant resources can be processed on behalf of the current transaction.

Use the EXCEPTION_HANDLER option to specify an exception handler to be installed for the lifetime of this transaction. All exceptions generated within the scope of this transaction are passed to the associated handler, unless a more specific scope has specified its own handler, for example, by an individual request.

For example, this command starts a transaction with the exception handler

```
my_exc_handler:
xa_begin EXCEPTION_HANDLER "my_exc_handler"
```

Exceptions related to the parsing or execution of the xa_begin command do not cause the associated exception handler to be invoked, since the exception occurs before the transaction has begun.

For information about event scopes and handler properties, refer to “Handler Scope and Installation” on page 6-3 in *JetNet/Oracle Tuxedo Guide*.

The following application properties are affected by execution of xa_begin:

Property	Value
tp_return	TP_NOTRAN or TP_TOP_LEVEL—Indicates type of transaction started, if any.
tp_tran_status	TP_WILL_COMMIT—If a new transaction is started and becomes the active transaction context. Otherwise, unchanged.

Exceptions `xa_begin` can generate the following exceptions:

Exception	Severity	Cause
`TP_BEGIN_FAILED`	`TP_COMMAND` `TP_ERROR`	The middleware is unable to begin a transaction
`TP_INVALID_CONNECTION`	`TP_COMMAND`	There is no connection to the middleware
`TP_INVALID_OPTION_VALUE`	`TP_COMMAND`	An invalid ***`timeout`*** value is specified
`TP_MONITOR_ERROR`	`TP_ERROR`	Error is reported from the middleware
`TP_INVALID_MONITOR_OPTION`	`TP_WARNING`	***`timeout`*** is specified as `INFINITE` and the middleware does not support it. If the command continues, the maximum ***`timeout`*** allowed by the middleware is used
`TP_TRANSACTION_LIMIT`	`TP_COMMAND`	A transaction is already active; only one is allowed to exist at a time

Example

```
// Process a bank account withdrawal.
// FML buffers are used in a call to service WITHDRAWAL
proc withd ()
vars message
//******** Perform ATM Withdrawal ********
if (account_id == "")
{
    msg quiet "Account id is required"
    return 0
}

if (amount > 0)
{
    xa_begin
    service_call "WITHDRAWAL" ({account_id, amount}, \
        {message, balance = account_balance})
    xa_end
    if (@app()->tp_svc_outcome == TP_FAILURE)
    {
        msg quiet message
    }
}

else
{
   msg quiet "Invalid withdrawal amount"
}
return 0
```

See Also `xa_commit`, `xa_end`, `xa_rollback`

xa_commit

Commits an XA-compliant transaction

Synopsis

```
xa_commit
```

Environment Oracle Tuxedo

Scope Client, Server

Description The `xa_commit` command commits the current transaction, initiated by `xa_begin`. Once initiated, a transaction must be completed by a call to either `xa_commit`, `xa_rollback`, or `xa_end`.

For example, the following transaction calls service `DEPOSIT`. If successful, the transaction is committed; otherwise, it is rolled back:

```
xa_begin
service_call "DEPOSIT" ({ACCOUNT, AMOUNT})
if ((@app()->tp_severity > TP_WARNING) \
    || (@app()->tp_return < 0) || (@app()->tp_tran_status < 0))
{
    xa_rollback
    return 0
}
xa_commit
return 0
```

`xa_commit` can set the `tp_return` property to one of these values:

- `TP_COMMIT`—The transaction is committed.
- `TP_PARTIAL_COMMIT`—The transaction is partially committed.
- `TP_NOT_COMPLETED`—The attempt to commit failed; the transaction remains viable.
- `TP_ROLLBACK`—The attempt to commit failed, the transaction is rolled back.

Exceptions Execution of `xa_commit` can generate the following exceptions:

Exception	Severity	Cause
`TP_COMMIT_FAILED`	`TP_ERROR`	Attempt to commit the transaction failed
`TP_COMMIT_PARTIAL`	`TP_WARNING`	Transaction has or might have been partially rolled back
`TP_COMMIT_ROLLEDBACK`	`TP_WARNING`	Transaction cannot be committed because it has been rolled back
`TP_INVALID_CONTEXT`	`TP_ERROR`	Commit operation was attempted outside of a transaction
`TP_INVALID_TRANSACTION`	`TP_ERROR`	There is no current transaction
`TP_MONITOR_ERROR`	`TP_ERROR`	An error was reported from the middleware
`TP_WORK_OUTSTANDING`	`TP_COMMAND`	There is still service request work that has not been completed

See Also `xa_begin`, `xa_end`, `xa_rollback`

xa_end

Completes an XA-compliant transaction

Synopsis

```
xa_end
```

Environment Oracle Tuxedo

Scope Client, Server

Description The `xa_end` command terminates a middleware transaction. It checks the `tp_tran_status` property to determine whether the current transaction is successful. If no errors occurred, `xa_end` commits the transaction; otherwise, the transaction is rolled back. Exception handlers play a direct role in determining whether to commit or abort a transaction; the exception handler decides the actual severity of an exception, and thus determines the value set in the `tp_tran_status` application property. `xa_end` can set the `tp_return` property to one of these values:

- `TP_COMMIT`—The transaction is committed.
- `TP_PARTIAL_COMMIT`—The transaction is partially committed.
- `TP_NOT_COMPLETED`—The attempt to commit failed; the transaction remains viable.
- `TP_ROLLBACK`—The attempt to commit failed, the transaction is rolled back.

Example In the following example, the following client code makes a service call that performs an account deposit transaction. The server side of the deposit process is shown in the return command.

```
proc dep()
vars message
//******** Perform ATM Deposit ********
if (account_id == "")
{
   msg quiet "Account id is required"
   return 0
}
...

if (amount > 0)
{
    xa_begin
    service_call "DEPOSIT" ({account_id, amount}, \
       {message, balance = account_balance})
    xa_end
    if (@app()->tp_svc_outcome == TP_FAILURE)
    {
       msg quiet message
    }
}

else
{
    msg quiet "Invalid deposit amount"
}
return 0
```

Exceptions

Because xa_end implicitly performs either xa_commit or xa_rollback, refer to those commands for possible exceptions.

See Also

xa_begin, xa_commit, xa_rollback

xa_rollback

Aborts an XA-compliant transaction

Synopsis

```
xa_rollback
```

Environment Oracle Tuxedo

Scope Client, Server

Description The `xa_rollback` command aborts the current middleware transaction. If a transaction has unfinished work, this command cancels all outstanding requests and rolls back all child transactions. Unfinished work includes:

- A service request performed as part of the transaction that has not yet completed.
- Resource manager data manipulation that is not yet committed.

On return, `xa_rollback` sets the `tp_return` property to one of these values:

- `TP_PARTIAL_COMMIT`—The transaction is partially committed.
- `TP_NOT_COMPLETED`—The attempt to rollback failed; the transaction remains viable.
- `TP_ROLLBACK`—The transaction is rolled back.

Example In the following example, service transfer is called by a client to transfer money between accounts. transfer first calls service withdrawal to withdraw the amount from the debit account. If this fails, the entire transaction is rolled back.

```
// Client code:
...
service_call "transfer" \
   ({debit_acct_id, credit_acct_id, amount },\
   {message, debit_bal, credit_bal})
if (@app()->tp_svc_outcome ! = TP_SUCCESS
{
   msg emsg message
}
...
```

```
// Service: transfer
proc transfer
vars withdrawal_msg


receive arguments \
    ({debit_acct_id, credit_acct_id, amount})
xa_begin
service_call "withdrawal" \
    ({account_id = debit_acct_id, amount}, \
    {message=withdrawal_msg, balance = debit_bal})
if ((@app()->tp_severity > TP_WARNING) ||
     (@app()->tp_svc_outcome !=TP_SUCCESS))
{
    xa_rollback
    service_return FAILURE \
        ({message = "Debit account problem -- :withdrawal_msg"})
}
...
```

Exceptions The `xa_rollback` command can generate the following exceptions:

Exception	Severity	Cause
`TP_MONITOR_ERROR`	`TP_ERROR`	An error is reported from the middleware
`TP_ROLLBACK_COMMITTED`	`TP_ERROR`	Transaction cannot be rolled back because it has already been committed
`TP_ROLLBACK_FAILED`	`TP_ERROR`	Transaction could not be rolled back
`TP_INVALID_TRANSACTION`	`TP_COMMAND`	There is no current transaction

See Also `xa_begin`, `xa_commit`, `xa_end`

CHAPTER

3 Built-in Control Functions

This chapter describes control functions supplied with Panther. You can use these functions in control strings and in JPL call statements. Unlike other control hook functions, these functions are installed internally and cannot be deinstalled.

- `jm_exit`—ends processing and leaves the current screen
- `jm_gotop`—returns to form stack's base screen
- `jm_goform`—invokes a dialog box that prompts for the name of a screen to display
- `jm_keys`—simulates keyboard input
- `jm_system`—prompts for and executes an operating system command
- `jm_winsize`—lets users manipulate a screen's viewport from the keyboard

Notes: Built-in control functions are internally installed. Unlike Panther library functions, they can only be called from within Panther.

jm_exit

Ends processing and leaves the current screen

```
jm_exit
```

Description

`jm_exit` closes the current form or window and returns to the previous one. If the form is the application's base form and the setup variable `CLOSELAST_OPT` is set to `OK_CLOSELAST`, Panther asks the user whether to exit the application.

By default, `EXIT` invokes this function at runtime.

Example

```
/* The following control string invokes a function named
   process. If it returns 0, another function is
    invoked to reinitialize the screen. If it returns -1,
    the screen closes.
 */

^(-1=^jm_exit; 0=^reinit)process

/* This control string replaces a form or a window with another
   form or a window
 */

^(0=&w2)jm_exit
```

jm_gotop

Returns to form stack's base screen

Description

`jm_gotop` returns to the form stack's base screen–typically, the first screen that the application displays at startup. Panther closes all other forms and windows and removes them from their respective stacks.

By default, SPF1 invokes this function at runtime.

jm_goform

Invokes a dialog box that prompts for the name of a screen to display

```
jm_goform
```

Description

`jm_goform` invokes an Open Screen dialog box that asks the user to enter the name of a screen to open. By default, Panther opens the screen as a form; however, users can specify to open a screen as a a stacked or sibling window. If the screen opens as a form, Panther closes all previously open windows before it displays the specified screen.

By default, the SPF3 key invokes this function at runtime.

For example, the following line in your setup file causes PF10 to invoke `jm_goform`.

```
SMINICTRL= PF10=^jm_goform
```

jm_keys

Simulates keyboard input

```
jm_keys input[ , input...]
```

input

A logical key or string to push onto the input queue. Arguments can be space or comma-delimited. Strings are enclosed by single or double quote characters. Logical keys are defined in `smkeys.h`. For a complete list of Panther logical keys, refer to Table 6-1 on page 6-7 in *Configuration Guide*.

Because `jm_keys` passes its arguments to `sm_ungetkey` in reverse order, list them in their actual input sequence. You can specify up to 20 arguments.

Description

`jm_keys` queues the specified characters and function keys for input to the runtime system through successive calls to `sm_ungetkey`. The runtime system then be haves as though you had typed the keys or strings.

For a single call to `jm_keys`, list items in input order. Items in a single call are placed on the input queue in right to left order; the keyboard stack then processes items by last in, first out (LIFO) order.

For example, the following single call to `jm_keys` enters a string value into the current field, then tabs to the next field and enters a number value into it:

```
^jm_keys 'Steinway Brauhall', TAB, "104"
```

Successive calls to `jm_keys` place additional items on the input queue; the keyboard stack processes the last item first. For example, the following three calls:

```
jm_keys "One"
jm_keys "Two"
jm_keys "Three"
```

would output the following keyboard sequence:

```
ThreeTwoOne
```

jm_system

Prompts for and executes an operating system command

```
jm_system
```

Description

`jm_system` invokes a dialog box in which you can enter an operating system command. By default, the SPF2 key invokes this function at runtime.

For example, the following line in your setup file causes PF10 to invoke system.

```
SMINICTRL= PF10 = ^jm_system
```

See Also

`sm_shell`

jm_winsize

Lets users manipulate a screen's viewport from the keyboard

```
jm_winsize
```

Description

Valid only in character mode, `jm_winsize` invokes the system menu and selects the Move option. Cursor keys are enabled to change the window's position, size, and the offset of its contents. You can also change focus to a sibling window. `XMIT` accepts the changes; `EXIT` cancels them.

The initial mode is resize. You can change the mode through one of these function keys:

- F2: Move the screen.
- F3: Resize the screen.
- F4: Change offset of the screen's contents within its window.
- F5: Change focus to a sibling window.

See Also

`sm_winsize`

CHAPTER

4 Library Function Overview

This chapter summarizes the Panther library functions and lists them by category. Groups of closely related variant functions are listed under a single root name. The functions `sm_r_form`, `sm_d_form`, and `sm_l_form`, for example, are all grouped under the heading `sm_*form`.

Functions marked with § are not installed in the distribution and cannot be directly called from JPL. All other functions can be called from JPL.

Initialization/Reset

The following library functions are called in order to initialize or reset certain aspects of the Panther runtime environment. Those that are necessary for the proper operation of Panther are called from within the supplied main routine source modules `jmain.c` and `jxmain.c`.

Table 4-1 Initialization/Reset

`sm_cancel`	Resets the display and exits
`sm_do_uinstalls`§	Installs an application's event functions

Table 4-1 Initialization/Reset *(Continued)*

sm_inimsg§	Creates a displayable error message on failure of an init initialization function
sm_initcrt§	Initializes the display and Panther data structures
sm_install§	Installs application event functions
sm_jtop§	Starts Panther
sm_launch	Invokes a process without waiting for it to return
sm_leave§	Prepares to temporarily leave a Panther application
sm_resetcrt§	Resets the terminal to the operating system's default state
sm_return§	Prepares for return to Panther application
sm_shell	Executes a system call
sm_vinit§	Initializes the video translation table

§ *Cannot be called directly from JPL.*

Screen and Viewport Control

Control viewports, the display of screens, and the form and window stacks:

Table 4-2 Screen/Viewport Control

sm_*at_cur	Displays a window at the cursor location
sm_close_window	Closes the current window
sm_*form	Opens a screen as a form
sm_formlist§	Updates the list of memory-resident files
sm_issv	Checks whether a screen is in the saved list

Table 4-2 Screen/Viewport Control *(Continued)*

`sm_jclose`	Closes the current window or form
`sm_jform`	Displays a screen as a form
`sm_jwindow`	Displays a window at a given position
`sm_load_screen`	Preloads a screen into memory
`sm_rmformlist`§	Purges the memory-resident form list
`sm_setsibling`	Specifies to open the next screen as a sibling of the current window
`sm_shrink_to_fit`	Removes trailing empty array elements and shrinks the screen
`sm_svscreen`§	Registers a list of screens on the save list
`sm_unload_screen`	Unloads a screen, freeing the memory associated with it
`sm_unsvscreen`§	Removes screens from the save list
`sm_wcount`	Obtains the number of currently open windows
`sm_wdeselect`	Restores the previously active window
`sm_win_shrink`	Trims the current screen
`sm_*window`	Opens a window at a given position
`sm_winsize`	Lets users interactively move and resize a window
`sm_wrotate`	Rotates the display of sibling windows
`sm_wselect`	Activates a window

§ *Cannot be called directly from JPL.*

Interscreen Messaging

Send and receive data from one screen to another:

Table 4-3 Interscreen Messaging

sm_append_bundle_data	Sends data to a bundle item
sm_append_bundle_done	Optimizes memory allocated for a send bundle
sm_append_bundle_item	Adds a data item to a bundle
sm_create_bundle	Creates a send bundle
sm_free_bundle	Destroys a send bundle
sm_get_bundle_data	Reads an occurrence of bundle item data
sm_get_bundle_item_count	Counts the number of data items in a bundle
sm_get_bundle_occur_count	Counts the number of occurrences in a data item
sm_get_next_bundle_name	Gets the name of the bundle created before the one specified
sm_is_bundle	Checks whether a bundle exists
sm_receive	Executes a JPL receive command
sm_send	Executes a JPL send command

Widget Creation/Deletion

Table 4-4 Widget Creation/Deletion

sm_obj_copy	Copies a widget
sm_obj_delete*	Deletes a widget

Property Access

Set and get properties of Panther objects–for example, screens, widgets, and the application itself:

Table 4-5 Property Access

`sm_prop_error`	Gets the error code returned by the last properties API function call
`sm_prop_get`	Gets a property setting
`sm_prop_id`	Returns an integer handle for an application component
`sm_prop_name_to_id`	Gets the integer ID of a Panther property
`sm_prop_set`	Sets a property

Field/Array Data Access

Access data in fields and arrays:

Table 4-6 Field/Array Data Access

`sm_*amt_format`	Writes formatted data to a field
`sm_calc`	Executes a math expression
`sm_cl_all_mdts`	Clears the mdt property for all occurrence on current screen
`sm_cl_unprot`	Clears data from unprotected widgets
`sm_*clear_array`	Clears all data in an array

Table 4-6 Field/Array Data Access *(Continued)*

sm_*copyarray	Copies the contents of one array to another
sm_*dblval	Returns the value of a field as a double precision floating point number
sm_*dlength	Gets the length of a field's contents
sm_*doccur	Deletes occurrences from a field
sm_*dtofield§	Writes a real number to a field
sm_*fptr	Gets the contents of a field
sm_*getfield	Copies the contents of a field
sm_*intval	Gets the integer value of a field
sm_*ioccur	Inserts blank occurrences into an array
sm_*is_no	Tests a field for no
sm_*is_yes	Tests a field for yes
sm_*itofield	Writes an integer value to a field
sm_list_objects_count	Counts the widgets contained by an application object
sm_list_objects_end	Destroys an object contents list
sm_list_objects_next	Traverses the widgets contained by an application object
sm_list_objects_start	Constructs a list of widgets contained by an application object
sm_*lngval§	Gets the long integer value of a field
sm_*ltofield§	Writes a long integer value to a field
sm_*null	Tests whether a field is null
sm_obj_sort	Sorts an object's occurrences
sm_obj_sort_auto	Sorts an object's occurrences according to grid settings
sm_optmnu_id	Gets the ID of an option menu or combo box
sm_*putfield	Puts a string into a field
sm_sdtime	Gets the formatted system date and time

Table 4-6 Field/Array Data Access *(Continued)*

sm_*strip_amt_ptr	Strips non-digit characters from a string
sm_udtime§	Formats a user-supplied date and time
sm_upd_select§	Updates the contents of an option menu or combo box
sm_*ww_length	Gets the number of characters in a wordwrapped multiline text widget
sm_*ww_read§	Copies the contents of a wordwrapped text widget into a text buffer
sm_*ww_write	Writes text into a wordwrapped multiline text widget

§ *Cannot be called directly from JPL.*

Group Access

The following functions access groups. Groups are made up of fields that have attributes and data in them. The value of a group indicates the set of selected constituent fields, although it is not recommended that that value ever be accessed or modified directly with any of the field access functions discussed in the preceding sections.

Table 4-7 Group Access

sm_deselect	Deselects an occurrence in a selection widget group
sm_*ftog§	Converts field references to selection group references
sm_*gtof§	Converts a selection group name and occurrence into a field number and occurrence
sm_n_gval	Forces execution of a group's validation function
sm_next_sync	Finds the next synchronized array
sm_select	Selects an occurrence in a selection group

§ *Cannot be called directly from JPL.*

Local Data Block Access

The following functions access local data blocks, or LDBs. Note that if a field data access function references a field by name–for example, `sm_n` and `sm_i_variants`–and the name field does not exist on the active screen, it looks in an active LDB for an entry of the same name.

Table 4-8 Local Data Block Access

`sm_allget`	Loads data from the active LDBs to the current screen
`sm_dd_able`	Turns LDB write-through on or off for all LDBs
`sm_*ldb_fld_*get`	Copies data from LDBs to specific fields on the current screen
`sm_*ldb_fld_*store`	Copies data from specific fields to LDBs
`sm_*ldb_*getfield`§	Gets the contents of an LDB entry
`sm_ldb_get_active`	Gets the handle of the most recently loaded active LDB
`sm_ldb_get_inactive`	Gets the handle of the most recently loaded inactive LDB
`sm_ldb_get_next_active`	Gets the active LDB loaded before the one specified
`sm_ldb_get_next_inactive`	Gets the inactive LDB loaded before the one specified
`sm_ldb_handle`	Gets the handle of an LDB
`sm_ldb_init`§	Initializes or reinitializes LDBs
`sm_ldb_is_loaded`	Tests whether an LDB is loaded
`sm_ldb_load`	Loads an LDB into memory
`sm_ldb_name`	Gets the name of an LDB of the specified handle
`sm_ldb_next_handle`	Gets the handle of previously loaded LDB with the same name as the specified LDB
`sm_ldb_pop`	Pops LDBs off the LDB save stack

Table 4-8 Local Data Block Access *(Continued)*

sm_ldb_push	Pushes all LDBs onto a save stack
sm_*ldb_*putfield	Reads data into an LDB entry
sm_ldb_*state_get	Gets the current state of the LDB
sm_ldb_*state_set	Changes the state of the LDB
sm_ldb_*unload	Unloads LDBs from memory
sm_lstore	Copies everything from screen to LDB

§ *Cannot be called directly from JPL.*

Validation

The following functions provide an application interface to the field and group validation processes:

Table 4-9 Validation

dm_val_relative	Sets bits for validation after SELECT statements are executed
sm_ckdigit	Validates data with a check digit function
sm_cl_all_mdts	Clears mdt property for all occurrences
sm_fval	Forces field validation
sm_n_gval	Forces execution of a group's validation function
sm_s_val	Validates the current screen
sm_tst_all_mdts§	Finds the first modified occurrence on the current screen
sm_validate	Validates the specified object–screen, widget or widget group

§ *Cannot be called directly from JPL.*

Cursor Control

Control the positioning and display of the cursor on the active screen:

Table 4-10 Cursor Control

`sm_backtab`	Backtabs to the previous unprotected field
`sm_c_off`	Turns the cursor off
`sm_c_on`	Turns the cursor on
`sm_c_vis`	Turns the cursor position display on or off
`sm_disp_off`	Gets the cursor's offset in the current field
`sm_gofield`	Moves the cursor into a field
`sm_home`	Homes the cursor
`sm_last`	Positions the cursor in the last field
`sm_nl`	Positions the cursor to the first unprotected field beyond the current line
`sm_off_gofield`	Moves the cursor into a field, offset from the left
`sm_sh_off`	Gets the cursor location relative to the start of a shifting field
`sm_tab`	Moves the cursor to the next unprotected field

Display Terminal I/O

Set the interface to Panther terminal I/O:

Table 4-11 Display Terminal I/O

`sm_bel`	Issues a beep from the terminal
`sm_bkrect`	Sets the background color of a rectangle
`sm_flush`	Flushes delayed writes to the display
`sm_getkey`	Gets the logical value of the key hit
`sm_input`	Opens the keyboard for data entry and menu selection
`sm_key_integer`	Gets the integer value of a logical key mnemonic
`sm_keyfilter`	Controls keystroke record/playback filtering
`sm_keyhit`	Tests whether a key is typed ahead
`sm_keyinit`	Initializes a key translation table
`sm_keylabel`	Gets the printable name of a logical key
`sm_keyoption`	Sets cursor control key options
`sm_m_flush`	Flushes the status line
`sm_rescreen`	Refreshes the data displayed on the screen
`sm_resize`	Notifies Panther of a change in the display size
`sm_ungetkey`	Pushes a translated key onto the input queue
`sm_xlate_table`§	Installs or deinstalls an 8-bit character translation table

§ *Cannot be called directly from JPL.*

Message Display

Access and display runtime application messages:

Table 4-12 Message Display

`sm_d_msg_line`	Displays a message on the status line
`sm_femsg`§§	Displays an error message and awaits user acknowledgement
`sm_ferr_reset`§§	Displays an error message and awaits user acknowledgement
`sm_fqui_msg`§§	Displays an error message preceded by a constant tag
`sm_fquiet_err`§§	Displays an error message preceded by a constant tag
`sm_hlp_by_name`	Displays a help window
`sm_message_box`	Displays a message in a dialog box
`sm_msg`	Displays a message at a given column on the status line
`sm_msg_del`	Removes a class of messages from memory
`sm_msg_get`	Finds a message
`sm_msg_read`§	Reads messages from a message file
`sm_msg_set`	Replace a message
`sm_msgfind`	Finds a message given its number
`sm_sb_delete`	Deletes a status bar section
`sm_sb_format`	Sets a format string for a status bar section
`sm_sb_gettext`	Gets contents of a status bar section
`sm_sb_insert`	Inserts a status bar section
`sm_sb_settext`	Sets contents of a status bar section
`sm_setbkstat`	Sets background text for status line
`sm_setstatus`	Turns alternating background status message on or off

§ *Cannot be called directly from JPL.*
§§ *In JPL, error messages are handled by the* `msg` *command.*

Mass Storage and Retrieval

Move data to or from sets of fields in the screen or LDB:

Table 4-13 Mass Storage and Retrieval

`sm_restore_data`§	Restores previously saved data to the screen
`sm_rs_data`§	Restores saved data to some of the screen
`sm_save_data`§	Saves screen contents
`sm_sv_data`§	Saves partial screen contents
`sm_sv_free`§	Frees a buffer that contains saved screen data
`sm_svscreen`§	Registers a list of screens on the save list

§ *Cannot be called directly from JPL.*

Global Data and Changing Panther Behavior

Get access to global data and manipulate their settings:

Table 4-14 Global Data and Changing Application Behavior

`sm_inquire`	Gets the value of a global integer variable
`sm_iset`	Changes the value of a global integer variable
`sm_ms_inquire`	Gets information about the mouse's current state

Table 4-14 Global Data and Changing Application Behavior *(Continued)*

sm_mus_time§	Gets the system time of the last mouse click
sm_occur_no	Gets the current occurrence number
sm_option	Sets a behavior variable
sm_pinquire	Gets the value of a global string
sm_pset	Modifies the value of a global string
sm_soption	Sets a string option

§ *Cannot be called directly from JPL.*

Menus

Get and change properties of menus and menu items:

Table 4-15 Menus

sm_menu_bar_error	Returns the last error returned by a menu function
sm_menu_change	Sets a menu's properties
sm_menu_create	Defines a menu at runtime
sm_menu_delete	Removes a menu from the specified script
sm_menu_get_*	Gets a menu's property
sm_menu_install	Makes a menu available for display
sm_menu_remove	Removes a menu from display
sm_mncrinit6§	Initializes support for the menu subsystem
sm_mnitem_change§§	Sets a menu item's property

Table 4-15 Menus *(Continued)*

sm_mnitem_create	Inserts a new item into a menu
sm_mnitem_delete	Removes an item from a menu
sm_*mnitem_get_*	Gets a menu item's property
sm_mnscript_load	Loads a menu script into memory and makes its menus available for installation
sm_mnscript_unload	Removes a script from memory and destroys all menus installed from it
sm_popup_at_cur	Invokes the current widget's popup menu

§ *Cannot be called directly from JPL.*
§§ *Wrapper functions for sm_mnitem_change are prototyped in funclist.c and callable from JPL. For a list of these functions and their parameter declarations, refer to Table 5-15 on page 5-343.*

Database Interaction

Table 4-16 Database Initialization

dm_dbi_init§	Initializes for database interaction
dm_init§	Initializes access to a specific database engine
dm_reset§	Disables support for a named database engine

§ *Cannot be called directly from JPL.*

Table 4-17 Database Access

dm_cursor_connection	Gets the connection name for a database cursor
dm_cursor_consistent	Determines if a cursor is on the default connection
dm_cursor_engine	Determines the database engine of a cursor

Table 4-17 Database Access *(Continued)*

`dm_dbms`	Executes a DBMS command directly from C
`dm_dbms_noexp`	Executes a DBMS command without colon preprocessing
`dm_expand`§	Formats a string for an engine
`dm_get_connection_option`	Gets a connection option
`dm_get_db_conn_handle`§	Gets a handle to a database connection logon structure
`dm_get_db_cursor_handle`§	Gets a handle to a database cursor's structure
`dm_get_driver_option`	Gets a database driver option
`dm_getdbitext`§	Gets the text of the last-executed DBMS command
`dm_is_connection`	Verifies that a connection is open
`dm_is_cursor`	Checks to see if a cursor exists
`dm_is_engine`	Verifies that a database engine is initialized
`dm_set_connection_option`	Sets a database connection option
`dm_set_driver_option`	Sets a database driver option
`dm_set_max_fetches`	Sets the maximum number of rows in a select set
`dm_set_max_rows_per_fetch`	Sets the maximum number of rows per fetch
`dm_set_onevent`	Install a C DBi event hook function

§ *Cannot be called directly from JPL.*

Table 4-18 Database Binary Variables

`dm_bin_create_occur`§	Gets or allocates an occurrence in a binary variable
`dm_bin_delete_occur`§	Deletes an occurrence in a binary variable
`dm_bin_get_dlength`§	Gets the length of an occurrence in a binary variable
`dm_bin_get_occur`§	Gets the data in an occurrence of a binary variable
`dm_bin_length`§	Gets the maximum length of an occurrence in a binary variable

Table 4-18 Database Binary Variables *(Continued)*

`dm_bin_max_occur`§	Gets the maximum number of occurrences in a binary variable
`dm_bin_set_dlength`§	Sets the length of an occurrence in a binary variable

§ *Cannot be called directly from JPL.*

Table 4-19 SQL Generation

`dm_exec_sql`	Generates and executes SQL statements
`dm_free_sql_info`	Frees memory associated with an SQL statement
`dm_gen_change_execute_using`	Adds or replaces a bind value in a DBMS `EXECUTE` statement for SQL generation
`dm_gen_change_select_from`	Edits the `FROM` clause in a `SELECT` statement for automatic SQL generation
`dm_gen_change_select_group_by`	Edits the `GROUP BY` clause in a `SELECT` statement for automatic SQL generation
`dm_gen_change_select_having`	Edits the `HAVING` clause in a `SELECT` statement for automatic SQL generation
`dm_gen_change_select_list`	Edits the select list for automatic SQL generation
`dm_gen_change_select_order_by`	Edits the `ORDER BY` clause in a `SELECT` statement for automatic SQL generation
`dm_gen_change_select_suffix`	Appends text to the end of a `SELECT` statement for automatic SQL generation
`dm_gen_change_select_where`	Edits the `WHERE` clause in a `SELECT` statement for automatic SQL generation
`dm_gen_get_tv_alias`	Gets the correlation name, or alias, for a table view
`dm_gen_sql_info`	Generates a data structure used in `SELECT` statements

Transaction Manager

Table 4-20 Transaction Manager Access

dm_disable_styles	Suppresses the application of transaction manager styles
dm_enable_styles	Enables the application of transaction manager styles
dm_set_tm_clear_fast	Determines the behavior of the CLEAR command
sm_tm_clear	Clears all fields in the table view
sm_tm_command	Executes a transaction command
sm_tm_event	Returns the event number for the specified transaction manager event name
sm_tm_event_name	Returns the transaction manager event name for the specified event number
sm_tm_handling	Processes a handling method property
sm_tm_inquire	Gets an integer attribute of the current transaction
sm_tm_iset	Sets the value of an integer transaction attribute
sm_tm_old_bi_context	Sets a backward compatibility flag
sm_tm_pcopy§	Gets a string attribute of the current transaction and stores a copy
sm_tm_pinquire	Gets a string attribute of the current transaction for immediate use
sm_tm_pset	Sets the value of a string transaction attribute

§ *Cannot be called directly from JPL.*

T

Table 4-21 Transaction Manager Event Processing

sm_tm_clear_model_events	Empties the transaction event stack

Table 4-21 Transaction Manager Event Processing *(Continued)*

`sm_tm_continuation_validity`	Checks whether `CONTINUE` events are permitted for the current table view
`sm_tm_pop_model_event`	Pops an event off the transaction event stack
`sm_tm_push_model_event`	Pushes an event onto the transaction event stack

Table 4-22 Transaction Manager Error and Message Handling

`sm_tm_command_emsgset`	Initiates error message processing for a transaction manager error code
`sm_tm_command_errset`	Initiates error processing for a transaction manager error code
`sm_tm_dbi_checker`	Tests for common database errors during transaction manager processing
`sm_tm_error`	Reports an error condition
`sm_tm_errorlog`	Controls error log processing
`sm_tm_failure_message`	Specifies an error message for a failed event
`sm_tm_msg_count_error`	Reports a transaction manager error
`sm_tm_msg_emsg`	Reports an error of `emsg` severity
`sm_tm_msg_error`	Reports an error

Table 4-23 Before-image Access in the Transaction Manager

`sm_bi_compare`	Compares widgets in the current table view with their before-image values
`sm_bi_copy`	Copies current values of a range of occurrences to the before-images
`sm_bi_initialize`	Initializes before-image data for widgets in the current table view
`sm_get_bi_data`	Returns the specified before-image data
`sm_get_tv_bi_data`	Gets before-image data

GUI Access

The following functions are applicable for GUI Panther applications. Those that contain _mw_ or _xm_ are specific to Windows or Motif only.

Table 4-24 GUI Access

`sm_adjust_area`	Recalculates widget positions
`sm_*attach_drawing_func`§	Associates a drawing function with a widget
`sm_delay_cursor`	Changes the state of the mouse pointer
`sm_*drawingarea`§	Gets a handle to the current screen that can be passed to the window manager
`sm_mw_DismissIntroPixmap`	Close the window containing the image selected by`IntroPixmap` in the application's .ini file
`sm_mw_get_client_wnd`	Gets a handle to the client area of the MDI frame
`sm_mw_get_cmd_show`	Returns the initial state of an application
`sm_mw_get_frame_wnd`	Gets a handle to the MDI frame
`sm_mw_get_instance`§	Gets a handle to the current instance of a Windows program
`sm_mw_get_prev_instance`	Gets a handle to the previous instance of a Windows program
`sm_mw_install_msg_callback`	Install a message handler in Panther's Windows message loop
`sm_mw_PrintScreen`	Prints a Panther screen
`sm_*PiMwCopyToClipboard`	Copy data from fields to the Windows clipboard
`sm_*PiMwPasteFromClipboard`	Paste data from the Windows clipboard to fields
`sm_translatecoords`§	Translates screen coordinates to display coordinates
`sm_*widget`§	Gets a handle to a widget

Table 4-24 GUI Access *(Continued)*

`sm_win_shrink`	Trims the current screen
`sm_xm_get_base_window`§	Gets a widget ID to the base window
`sm_xm_get_display`§	Gets a pointer to the current display

§ *Cannot be called directly from JPL.*

DDE (Dynamic Data Exchange)

Exchange data between Panther Windows applications and other Windows applications.

Table 4-25 DDE

`sm_dde_client_connect_cold`	Creates a cold DDE link to a server
`sm_dde_client_connect_hot`	Creates a hot DDE link to a server
`sm_dde_client_connect_warm`	Creates a warm DDE link to a server
`sm_dde_client_disconnect`	Destroys a DDE link to a server
`sm_dde_client_off`	Disables DDE client activity
`sm_dde_client_on`	Enables DDE client activity
`sm_dde_client_paste_link_cold`	Creates a cold DDE paste link between a widget and a DDE server
`sm_dde_client_paste_link_hot`	Creates a hot DDE paste link between a widget and a DDE server
`sm_dde_client_paste_link_warm`	Creates a warm DDE paste link between a widget and a DDE server
`sm_dde_client_request`	Requests data from a DDE server
`sm_dde_execute`	Sends a command to a DDE server

Table 4-25 DDE *(Continued)*

sm_dde_install_notify§	Installs a callback function that executes on changes in warm link data
sm_dde_poke	Pokes data into a DDE server
sm_dde_server_off	Disables DDE server activity
sm_dde_server_on	Enables DDE server activity

§ *Cannot be called directly from JPL.*

File Access

Table 4-26 File Access

sm_fi_path	Returns the full path name of a file
sm_file_copy	Copies a file
sm_file_exists	Checks whether a file exists
sm_file_move	Copies a file and deletes its source
sm_file_remove	Deletes a file
sm_filebox§	Opens a file selection dialog box
sm_filetypes	Adds an option to the file type option menu
sm_fio_a2f	Writes the contents of an array to a file
sm_fio_close	Closes an open file stream
sm_fio_editor	Invokes an external text editor for an array
sm_fio_error	Gets the error returned by the last call to a file I/O function
sm_fio_error_set	Sets the file I/O error

Table 4-26 File Access *(Continued)*

sm_fio_f2a	Writes a file's contents to an array
sm_fio_getc	Reads the next byte from the specified file stream
sm_fio_gets	Reads a line from a file
sm_fio_handle§	Gets a handle to an open file
sm_fio_open	Opens the specified file and returns a handle to the JPL caller
sm_fio_putc	Writes a single byte to an open file
sm_fio_puts	Writes a line of text to an open file
sm_fio_rewind	Resets the file stream to the beginning of a file
sm_jfilebox	Opens a file selection dialog box
sm_tmpnam	Creates a unique file name

§ *Cannot be called directly from JPL.*

Library Access

Table 4-27 Library Access

sm_l_close	Closes a library and frees all memory associated with it
sm_l_open	Opens a library
sm_l_open_syslib	Opens a library as a system library
sm_slib_error	Gets the system return for the last call to `sm_slib_load`
sm_slib_install	Installs a function from a DLL into a Panther application
sm_slib_load	Loads a dynamic link library (DLL) and makes its functions available for installation

JPL

Table 4-28 JPL

`sm_jplcall`	Executes a JPL procedure
`sm_jplpublic`	Executes JPL's public command
`sm_jplunload`	Executes JPL's unload command

JetNet/Oracle Tuxedo Processing

Table 4-29 JetNet/Oracle Tuxedo Processing

`sm_tp_exec`§	Executes a middleware API JPL command
`sm_tp_free_arg_buf`§	Frees memory allocated by argument list generation functions
`sm_tp_gen_insert`§	Generates an argument list of fields for an `INSERT` operation
`sm_tp_gen_sel_return`§	Generates a list of fields for the returned select set of a `SELECT` or `VIEW` operation
`sm_tp_gen_sel_where`§	Generates a list of fields for the `WHERE` clause of a `SELECT` or `VIEW` operation
`sm_tp_gen_val_link`§	Generates a list of fields to be validated in a validation link operation
`sm_tp_gen_val_return`§	Generates a list of fields for the returned select set of a validation link operation

Table 4-29 JetNet/Oracle Tuxedo Processing *(Continued)*

`sm_tp_get_svc_alias`§	Returns the alias assigned to a server
`sm_tp_get_tux_callid`§	Returns the Oracle Tuxedo identifier for a service call

§ *Cannot be called directly from JPL.*

Open Middleware Connectivity

The following functions can be used with all service components, regardless of the technology used to deploy the components–COM or EJB.

Table 4-30 Open Middleware Connectivity

`sm_log`	Writes a message to a log file
`sm_obj_call`	Calls a service component's method
`sm_obj_create`	Instantiates a service component
`sm_obj_get_property`	Gets a service component's property setting
`sm_obj_onerror`	Installs an error handler
`sm_obj_set_property`	Sets a property for a service component
`sm_raise_exception`	Sends an error code back to the client
`sm_receive_args`	Receives a list of in and in/out parameters for a method
`sm_return_args`	Returns a list of in/out and out parameters for a method

COM/MTS Processing

Table 4-31 COM/MTS Processing

`sm_obj_create_licensed`	Instantiates a licensed COM component
`sm_obj_create_server`	Instantiates a COM component
`sm_com_load_picture`	Returns the object ID for a graphics file in COM format
`sm_com_QueryInterface`§	Accesses an interface of a COM component
`sm_com_result`	Gets the error code returned by the last call to a COM component
`sm_com_result_msg`	Gets the error message returned by the last call to a COM component
`sm_com_set_handler`	Sets an event handler for the specified event on a COM component

§ *Cannot be called directly from JPL.*

Table 4-32 MTS Database Transactions

`sm_mts_CreateInstance`	Creates a new object for use in the existing transaction
`sm_mts_DisableCommit`	Prohibits the transaction from being committed
`sm_mts_EnableCommit`	Allows the transaction to be committed
`sm_mts_IsInTransaction`	Determines if the object is participating in a transaction
`sm_mts_SetAbort`	Tells MTS to abort the transaction
`sm_mts_SetComplete`	Tells MTS the work is complete (and ready to be committed)

Table 4-33 MTS Property Access

`sm_mts_CreateProperty`	Creates a named property

Table 4-33 MTS Property Access *(Continued)*

`sm_mts_CreatePropertyGroup`	Creates a named property group
`sm_mts_GetPropertyValue`	Gets the value of a property
`sm_mts_PutPropertyValue`	Sets a property's value

Table 4-34 MTS Security Checking

`sm_mts_IsCallerInRole`	Determines if the caller of the component is in a role
`sm_mts_IsSecurityEnabled`	Determines if security is currently enabled

Reports

Table 4-35 Reports

`sm_rw_error_message`	Returns the last error message generated by report processing
`sm_rw_play_metafile`	Displays or prints a report that is in metafile format
`sm_rw_runreport`	Invokes the report generator from a user-written function

Web Applications

Table 4-36 Web Applications

`sm_web_get_cookie`	Returns the value of a specified cookie
`sm_web_invoke_url`	Invokes a URL on the Web
`sm_web_log_error`	Writes Web application errors to a log file
`sm_web_save_global`	Creates a context global variable
`sm_web_set_cookie`	Sets HTML cookies on a client
`sm_web_set_onevent`	Install a C function as the Web event hook
`sm_web_unsave_all_globals`	Redesignates all context global variables as transient globals
`sm_web_unsave_global`	Redesignates a context global variable as a transient global

Mail

Table 4-37 Mail

`sm_*mail_attach`	Attaches a file to the email message
`sm_*mail_file_text`	Specifies the file containing the text of the email message
`sm_mail_message`	Send a simple email message
`sm_mail_new`	Creates a new mail object

Table 4-37 Mail *(Continued)*

`sm_*mail_send`	Sends a mail object
`sm_*mail_text`	Specifies the widget containing the text of the email message
`sm_mail_widget`	Specifies a widget to be included as an image attachment to an email message

XML

Table 4-38 XML

`sm_*xml_export`§	Generate XML from Panther screens and widgets
`sm_*xml_export_file`	Export Panther-generated XML to a file
`sm_*xml_import`	Import XML to a Panther screen
`sm_*xml_import_file`	Import XML to a Panther screen from a file

§ *Cannot be called directly from JPL.*

Miscellaneous

Table 4-39 Miscellaneous

`sm_dicname`§	Sets the repository name
`sm_ffree`§	Free allocated memory

Table 4-39 Miscellaneous *(Continued)*

`sm_fmalloc`§	Allocate memory
`sm_getenv`	Get system environment variable value
`sm_isabort`	Tests and sets the abort control flag
`sm_set_help`	Puts an application into help mode
`sm_strdup`§	Copy a string to newly allocated memory
`sm_trace`	Tracing and dumping of Panther events

§ *Cannot be called directly from JPL.*

CHAPTER

5 Library Functions

This chapter contains descriptions of Panther library functions arranged alphabetically. Each function description tells what the function does, and where and how to use it. Information about each function is organized into the following sections:

- Syntax lines that are patterned after C function declarations. A syntax line is given for each variant of this function. Syntax lines are preceded by `include` statements that are specific to the function.
- Parameter descriptions.
- Platforms on which the function is valid. If the function is available on all platforms, this section is omitted.
- Return values, if any. If the function returns no meaningful value, this section is omitted.
- Description of the function—typical usage, prerequisites, results, and potential side-effects.
- An example that shows how to use the function.
- Listing of related functions.

Note: Because all routines that call Panther library functions must include `smdefs.h`, syntax sections omit an `include` statement for this file. If the function requires inclusion of other header files, the syntax section contains `include` statements for them.

dm_bin_create_occur

Gets or allocates an occurrence in a binary variable

```
#include <dmuproto.h>
char *dm_bin_create_occur(char *variable, int occurrence);
```

`variable`
    The binary variable that contains the occurrence to get.

`occurrence`
    The occurrence in `variable` to get.

Environment
C only

Returns
- 0: The variable is not found or the occurrence number is not valid.
- A pointer to an occurrence in a binary variable.

Description
`dm_bin_create_occur` gets the specified occurrence from the variable if the application has created a binary variable with `DBMS BINARY`. If the occurrence has not been allocated, this function will allocate it. Note that `occurrence` must be less than or equal to the number of occurrences specified in the `DBMS BINARY` statement.

See Also
`DBMS BINARY`

dm_bin_delete_occur

Deletes an occurrence in a binary variable

```
#include <dmuproto.h>
void dm_bin_delete_occur(char *variable, int occurrence);
```

`variable`
: The binary variable that contains the occurrence to delete.

`occurrence`
: The occurrence in `variable` to delete.

Environment C only

Description `dm_bin_delete_occur` frees the specified occurrence and sets the pointer to the occurrence to 0 if the application has created a binary variable with `DBMS BINARY` and the occurrence has been allocated. If the occurrence has not been allocated, the function does nothing.

See Also `DBMS BINARY`

dm_bin_get_dlength

Gets the length of an occurrence in a binary variable

```
#include <dmuproto.h>
unsigned int dm_bin_get_dlength(char *variable, int occurrence);
```

`variable`
: The binary variable that contains the occurrence to measure.

`occurrence`
: The occurrence in `variable` whose length you want to get.

Environment C only

Returns

- 0: The variable or occurrence is not found.
- The length of the occurrence.

Description If the application has created a binary variable with `DBMS BINARY` and the occurrence has been allocated, this function returns the length of the contents in the specified occurrence.

See Also `DBMS BINARY`, `dm_bin_set_dlength`

dm_bin_get_occur

Gets the data in an occurrence of a binary variable

```
#include <dmuproto.h>
char *dm_bin_get_occur(char *variable, int occurrence);
```

`variable`
: The binary variable that contains the occurrence to get.

`occurrence`
: The occurrence in `variable` whose data you want to get.

Environment
: C only

Returns
- 0: The variable or occurrence is not found.
- A pointer to an occurrence in the variable.

Description
: If the application has created a binary variable with `DBMS BINARY` and the occurrence has been allocated, this function gets the specified occurrence from the variable.

See Also
: `DBMS BINARY`

dm_bin_length

Gets the maximum length of an occurrence in a binary variable

```
#include <dmuproto.h>
unsigned int dm_bin_length(char *variable);
```

`variable`
The variable whose maximum occurrence length you want to ascertain.

Environment
C only

Returns
- 0: The variable is not found.
- The length of the variable.

Description
If the application has created a binary variable with `DBMS BINARY`, this function gets the maximum length of a single occurrence in the variable. To get the length of an occurrence's contents, use `dm_bin_get_dlength`.

See Also
`DBMS BINARY`

dm_bin_max_occur

Gets the maximum number of occurrences in a binary variable

```
#include <dmuproto.h>
int dm_bin_max_occur(char *variable);
```

`variable`
The variable whose maximum number of occurrences you want to ascertain.

Environment C only

Returns

- 0: The variable is not found.
- The number of occurrences in the variable.

Description If the application has created a binary variable with `DBMS BINARY`, this function gets the maximum number of occurrences in the variable.

See Also `DBMS BINARY`

dm_bin_set_dlength

Sets the length of an occurrence in a binary variable

```
#include <dmuproto.h>
void dm_bin_set_dlength(char *variable, int occurrence,
    unsigned int length);
```

`variable`
: The variable that contains the occurrence to set.

`occurrence`
: The occurrence in `variable` whose length is to be set.

`length`
: The length to set for `occurrence`.

Environment C only

Description If the application has created a binary variable with `DBMS BINARY`, this function sets the length of a single occurrence in the binary variable. `length` can be less than or equal to the variable's declared length. If `length` is greater than the variable's declared length, the variable's length is used.

See Also `DBMS BINARY`, `dm_bin_get_dlength`

dm_convert_empty

Specifies the format of empty numeric fields

```
#include <dmuproto.h>
int dm_convert_empty(int flag);
```

`flag`

One of the following values:

0	Empty numeric fields are entered as "".
>0	Empty numeric fields are entered as 0.

Returns The previous value of the flag.

Description `dm_convert_empty` determines whether empty numeric fields ("") should be replaced with a `0`. This setting is database-specific since some databases do not allow NULL values in numeric columns.

dm_cursor_connection

Gets the connection name for a database cursor

```
#include <dmuproto.h>
char *dm_cursor_connection(char *cursor_name);
```

`cursor_name`
: Specifies a cursor name. For a named cursor, use the name specified in a `DBMS DECLARE CURSOR` command. To refer to the default connection, specify as a null pointer or an empty string.

Returns

- Name of the database connection name for the named cursor. If `cursor_name` is a null pointer or empty string, the name of the default connection is returned.
- An empty string if there is no such cursor as `cursor_name`, or if `cursor_name` is a null pointer or empty string, and there is no default connection.

Description

`dm_cursor_connection` returns the name of the connection for the named cursor, or returns the name of the default connection if `cursor_name` is a null pointer or an empty string.

See Also

`dm_cursor_consistent`, `dm_cursor_engine`

dm_cursor_consistent

Determines if a cursor is on the default connection

```
#include <dmuproto.h>
int dm_cursor_consistent(char *cursor_name);
```

`cursor_name`
: Specifies a cursor name. For a named cursor, use the name specified in a `DBMS DECLARE CURSOR` command. For a default cursor, specify as a null pointer or an empty string.

Returns

1 The cursor (named or default) exists and is on the default connection.
0 The cursor (named or default) is on a connection other than the default, or is not found.

Description

`dm_cursor_consistent` determines whether a database cursor is on the default connection. The cursor may be named, or if `cursor_name` is a null pointer or an empty string, the default cursor.

See Also

`dm_cursor_connection`, `dm_cursor_engine`

dm_cursor_engine

Determines the database engine of a cursor

```
#include <dmuproto.h>
char *dm_cursor_engine(char *cursor_name);
```

`cursor_name`
: Specifies a cursor name. For a named cursor, use the name specified in a `DBMS DECLARE CURSOR` command. To refer to the default engine, specify as a null pointer or an empty string.

Returns

- Name of the engine of the cursor's connection. If `cursor_name` is a null pointer or an empty string, the name of the default engine is returned.
- An empty string if the named cursor does not exist; or, if `cursor_name` is a null pointer or empty string and there is no default engine.

Description

`dm_cursor_engine` returns the name of the database engine for the database connection of the named cursor, or returns the name of the default engine if `cursor_name` is a null pointer or an empty string.

See Also

`dm_cursor_connection`, `dm_cursor_consistent`, `dm_init`

dm_dbi_init

Initializes for database interaction

```
#include <dmuproto.h>
void dm_dbi_init(void);
```

Environment C only

Description Panther must be initialized for use with the database drivers. `dm_dbi_init` tells Panther the class of error messages used with the database drivers and how to handle the JPL command `dbms`.

Panther calls this function in the source files `jmain.c` and `jxmain.c`. If you modify these files or if you write your own executive, you can call this function at another time. However, it should be called before `sm_initcrt` so that the message file loads properly.

dm_dbms

Executes a DBMS command directly from C

```
#include <dmuproto.h>
int dm_dbms(char *dbms_cmd);
```

`dbms_cmd`
: Points to a buffer with the `DBMS` command to execute. Refer to Chapter 11, "DBMS Statements and Commands," for detailed descriptions of each `DBMS` command.

Returns

- 0: Success.
- An error code from the default or installed error handler.

Description

`dm_dbms` lets you execute any `DBMS` command directly from C. This function executes in the following steps:

1. `dbms_cmd` is examined for the `WITH ENGINE` or `WITH CONNECTION` clause. If it is not used, `dm_dbms` assumes the default engine and connection.

2. The colon preprocessor examines `dbms_cmd` for colon variables and performs the indicated expansion.

3. `dbms_cmd` is passed to the appropriate function for handing `DBMS` commands. After executing the requested command, Panther updates all global status and error variables (`@dm`).

If the application has installed an entry function with `DBMS ONENTRY`, an exit function with `DBMS ONEXIT`, or an error handler with `DBMS ONERROR`, the installed function is called for commands executed through `dm_dbms`.

Example

```
int start_up ()
   {
      int retcode;
      retcode = dm_dbms ("ONERROR CALL do_error");
      if (retcode)
      {
         sm_emsg("Cannot install application error handler.")
         return 0;
      }
      dm_dbms ("DECLARE c1 CONNECTION FOR USER ':user' PASSWORD
      ':password'");
      return 0;
   }
```

See Also dm_dbms_noexp

dm_dbms_noexp

Executes a DBMS command without colon preprocessing

```
#include <dmuproto.h>
int dm_dbms_noexp(char *dbms_cmd);
```

`dbms_cmd`
: Points to a buffer that contains the `DBMS` command to execute.

Returns

- 0: Success.
- A return code from an installed or default error handler.

Description

`dm_dbms_noexp` is identical to `dm_dbms` except that no colon preprocessing is performed on `dbms_cmd`.

See Also

`dm_dbms`, `dm_expand`

dm_disable_styles

Suppresses application of transaction manager styles

```
#include <tmusubs.h>
int dm_disable_styles(void)
```

Returns

0	Transaction manager styles were previously not applied.
1	Transaction manager styles were previously applied.

Description

`dm_disable_styles` suppresses application of transaction manager styles. This can increase the efficiency of a transaction server, where user interface considerations don't apply. Styles are enabled by default, in accordance with the contents of `styles.sty`. If transaction manager processing occurs in batch mode, styles are disabled automatically.

See Also

`dm_enable_styles`

dm_enable_styles

Enables application of transaction manager styles

```
#include <tmusubs.h>
int dm_enable_styles(void)
```

Returns

0	Transaction manager styles were previously not applied.
1	Transaction manager styles were previously applied.

Description `dm_enable_styles` enables application of transaction manager styles. Styles are enabled by default, in accordance with the contents of `styles.sty`. If transaction manager processing occurs in batch mode, styles are disabled automatically. Disabling styles can speed up processing on a transaction server, where user interface considerations don’t apply.

See Also `dm_disable_styles`

dm_exec_sql

Generates and executes SQL statements

```
#include <tmusubs.h>
int dm_exec_sql(int type, char *cursor_name);
```

`type`
Type of SQL statement specified by one of the constants listed in Table 5-1.

`cursor_name`
Name of the cursor associated with the SQL statement.

Returns
- 0: Success.
- A non-zero value returned from an `ONENTRY`, `ONEXIT` or `ONERROR` function resulting from a generated SQL statement having executed.
- One of the `DM_TM_ERR_xxx` return values listed in `tmusubs.h`.

Description
`dm_exec_sql` is called from a transaction model or a user event function to generate and execute SQL statements according to one of the following constants supplied for the `type` parameter:

Table 5-1 SQL statement types

Argument	**Description**
`BUILD_SELECT`	Examines screen properties and builds structures for a `SELECT` statement including a distinct string, if specified, a select list (column names and/or expressions), and a `WHERE` clause.
`BUILD_VALIDATE`	Examines screen edits and builds structures for a `SELECT` statement used to process a validation link.

`DECLARE_DELETE_NBR`
`DECLARE_DELETE_OCC`
	Builds and executes the following statement for database deletions:
	
	`DBMS DECLARE` *`cursor`* `CURSOR FOR DELETE FROM`
	`current-table-view`
	  `WHERE` *`primary-key-column`* `= :w_`*`primary-key-column`* `...`

Table 5-1 SQL statement types *(Continued)*

Argument	Description
`DECLARE_INSERT`	Builds and executes the following statement for database insertions: `DBMS DECLARE` *cursor* `CURSOR FOR INSERT INTO` *current-table-view* `(`*column-name* `...)` `VALUES (:v_`*column-name*`...)`
`DECLARE_UPDATE`	Builds and executes the following statement for database updates: `DBMS DECLARE` *cursor* `CURSOR FOR UPDATE` *current-table-view* `SET` *column-name* `= :s_`*widget-name* `...` `WHERE` *primary-key-column* `= :w_`*primary-key-column* `...`
`EXEC_DELETE_NBR` `EXEC_DELETE_OCC`	Builds and executes the following statement for database deletions: `DBMS WITH CURSOR` *cursor* `EXECUTE USING` `w_`*primary-key-column* `= @bi(`*primary-key-widget*`)[`*occ*`] ...`
`EXEC_INSERT`	Builds and executes the following statement for database insertions: `DBMS WITH CURSOR` *cursor* `EXECUTE USING` `v_`*column-name* `=` *widget-name*`[`*occ*`] ...`
`EXEC_UPDATE`	Builds and executes the following statement for database updates: `DBMS WITH CURSOR` *cursor* `EXECUTE USING` `s_`*column-name* `=` *widget-name*`[`*occ*`] ...` `w_`*primary-key-column* `= @bi(`*primary-key-widget*`)[`*occ*`] ...`

Table 5-1 SQL statement types *(Continued)*

Argument	Description

`PERFORM_SELECT`	Executes the following statements for database queries:
	
	`DBMS DECLARE` *`cursor`* `CURSOR FOR SELECT [ DISTINCT ]` *`select-list`*
	`  FROM` *`tables-in-current-server-view`*
	`  [ WHERE [` *`join-clause`* `] [ AND` *`search-condition`* `] ]`
	`  [ GROUP BY` *`column-list`* `]`
	`  [ HAVING` *`search-condition`* `]`
	`  [ ORDER BY` *`column-position`* `{ ASC|DESC }, ... ]`
	
	`DBMS WITH CURSOR` *`cursor`* `ALIAS` *`widget-list`*
	
	`DBMS WITH CURSOR` *`cursor`* `<EXECUTE`
	`   [ USING [` *`join-values`* `] [` *`where-values`* `] [` *`having-values`* `] ]`

`PERFORM_VALIDATE`	Executes the following statements for validation links:
	
	`DBMS DECLARE` *`cursor`* `CURSOR FOR`
	`   SELECT {1 | look-up list} FROM` *`child-table-view`* `WHERE ...`
	`DBMS WITH CURSOR` *`cursor`* `ALIAS ...`
	`DBMS OCCUR`
	`DBMS WITH CURSOR` *`cursor`* `EXECUTE`
	`DBMS CLOSE CURSOR` *`cursor`*

Selecting Data

`dm_exec_sql(BUILD_SELECT)` and `dm_exec_sql(BUILD_VALIDATE)` should not be called without a prior call to `dm_gen_sql_info` to initialize the statement structures. In the standard transaction models, `dm_exec_sql` and other related functions are called by the following requests:

Table 5-2

Request	dm_exec_sql (and related) calls
`TM_SEL_GEN`	`dm_gen_sql_info(SELECT,` *`cursor`*`)`

`TM_SEL_BUILD_PERFORM`	`dm_exec_sql(BUILD_SELECT,` *`cursor`*`)`
	`dm_exec_sql(PERFORM_SELECT,` *`cursor`*`)`
	`dm_free_sql_info(SELECT)`

Table 5-2

Request	dm_exec_sql (and related) calls
`TM_VAL_GEN`	`dm_gen_sql_info(VALIDATE,` *`cursor`*`)`
`TM_VAL_BUILD_PERFORM`	`dm_exec_sql(BUILD_VALIDATE,` *`cursor`*`)` `dm_exec_sql(PERFORM_VALIDATE,` *`cursor`*`)`
`TM_VAL_CHECK`	`dm_free_sql_info(VALIDATE)`

Modifying Data

`dm_exec_sql(DECLARE_`*`xxx`*`)` should not be called without a prior call to `sm_bi_initialize`. The transaction manager calls `sm_bi_initialize` automatically when `sm_tm_command("NEW")` or `sm_tm_command("SELECT")` is executed. In the standard transaction models, `dm_exec_sql` and other related functions are called by the following requests:

Table 5-3

Request	dm_exec_sql calls
`TM_DELETE_DECLARE`	`dm_exec_sql(DECLARE_DELETE_NBR)` `dm_exec_sql(DECLARE_DELETE_OCC)`
`TM_DELETE_EXEC`	`dm_exec_sql(EXEC_DELETE_NBR)` `dm_exec_sql(EXEC_DELETE_OCC)`
`TM_INSERT_DECLARE`	`dm_exec_sql(DECLARE_INSERT)`
`TM_INSERT_EXEC`	`dm_exec_sql(EXEC_INSERT)`
`TM_UPDATE_DECLARE`	`dm_exec_sql(DECLARE_UPDATE)`
`TM_UPDATE_EXEC`	`dm_exec_sql(EXEC_UPDATE)`

dm_expand

Formats a string for an engine

```
#include <dmuproto.h>
int dm_expand(char *engine, char *data, int type, char *buf,
   int buflen, char *edit);
```

`engine`

The name of an initialized engine. If this argument is null, Panther uses the default engine.

`data`

The string to format. Use Panther library functions such as `sm_getfield` to get the value of a field or LDB entry.

`type`

A Panther data type, specified by one of the following constants defined in `smedits.h`:

`FT_CHAR`	`FT_DOUBLE`	`FT_LONG`
`DT_CURRENCY`	`FT_FLOAT`	`FT_SHORT`
`DT_DATETIME`	`FT_INT`	`DT_YESNO`

`buf`

A buffer provided by the program. The program is responsible for allocating a buffer large enough for the formatted string.

`buflen`

Points to the size of the buffer. Upon return from `dm_expand`, the value contained in the integer will be the length of the formatted text. The program can compare this value with the allocated length to ensure that truncation did not occur.

`edit`

A date-time edit string describing `data`. It is required when type is `DT_DATETIME`.

Environment C only

Returns

0	Success.
-1	`engine` is invalid.
-2	Arguments are invalid—illegal Panther type, `buflen` ≤ 0, `buf` not allocated, or `DT_DATETIME` was used without a datetime edit.
-3	Formatting function failed.

Description

`dm_expand` lets you format a string for a particular engine and Panther type. The function copies the formatted string to a buffer provided by the program.

Example

```
#include <smdefs.h>
#include <smedits.h>
#include >dmuproto.h>

char *
formatter (src_name, prolfxtype)
char *src_name;
int prolfxtype;
{
char src_buf[256];   /* For widget contents */
char *edit=0;        /* For datetime edit */
char dst_buf[256]; int dst_len=256;/* For formatted string*/

strcat (dst_buf, "");

       /* Get contents of non-null widget. */
    if ((sm_n_null (src_name) == 0) &&
       (sm_n_getfield (src_buf, src_name) > 0))
    {
    /* If no type was supplied, get it from the source
       field.*/
       if (prolfxtype == 0)
       {
           prolfxtype =
           sm_n_ftype(src_name, (int*)0) & DT_DTYPE;
       }

       /* If type is DT_DATETIME get format from source field. */
       if (prolfxtype == DT_DATETIME)
       {
           edit = sm_n_edit_ptr (src_name, UDATETIME);
           /* If there is no user format, check for
              system format. */
           if (edit == 0)
           {
              edit = sm_n_edit_ptr(src_name, SDATETIME);
           }
```

```
                edit = edit + 2;
            }

            /* Format text for the current engine. */
            dm_expand("", src_buf, prolfxtype, dst_buf, &dst_len, edit);
        }
        return dst_buf;
    }
```

See Also `dm_dbms_noexp`

dm_free_sql_info

Frees memory associated with a SELECT statement

```
#include <tmusubs.h>
int dm_free_sql_info(int type);
```

`type`

The type of `SELECT` statement, either `SELECT` or `VALIDATE`. When this function is called by the standard transaction models, the type is set to `SELECT` for the transaction commands `SELECT` and `VIEW`, and the type is set to `VALIDATE` for the transaction command `VALIDATE_LINK`.

Returns 0

Description `dm_free_sql_info` is used to free data that is associated with `SELECT` or `VALIDATE` statements. If the type is `SELECT`, it should follow the `BUILD_SELECT` or `PERFORM_SELECT` processing performed in `dm_exec_sql`.

If the type is `VALIDATE`, it should follow the `BUILD_VALIDATE` and `PERFORM_VALIDATE` processing performed in `dm_exec_sql` as well as any call to `dm_val_relative`.

Example

```
int retcode;
    char *sel_cursor;
    ...
    retcode = dm_exec_sql(BUILD_SELECT, sel_cursor);
    if (!retcode)
      retcode = dm_exec_sql(PERFORM_SELECT, sel_cursor);
    dm_free_sql_info(SELECT);
```

See Also `dm_gen_sql_info`

dm_gen_change_execute_using

Adds or replaces a bind value in a DBMS EXECUTE statement for SQL generation

```
#include <tmusubs.h>
int dm_gen_change_execute_using(char *arg, char *bind_parm,
   char *bind_val, int occ, int relative, int flag);
```

`arg`

Reserved for future use.

`bind_parm`

Specifies the bind parameter.; if a null pointer or empty string, the clause is not built.

`bind_val`

Specifies the bind value; if a null pointer or an empty string, the clause is not built.

`occ`

Specifies the occurrence number.

`relative`

Specifies how to use the occurrence number with one of the following values:

```
DM_GEN_ABSOLUTE_OCCUR
DM_GEN_RELATIVE_TO_PARENT
DM_GEN_RELATIVE TO CHILD
```

`flag`

Specifies the type of change to make with one of the following constants:

`DM_GEN_APPEND`

When `flag` is set to this value, `bind_val` is added to end of the `USING` clause. This produces the following statement:

```
DBMS WITH CURSOR cursor EXECUTE USING
 existing_parentTV_binds,
 existing_childTV_binds,
 bind_parm = bind_val[occ]
```

`DM_GEN_PREPEND`

When `flag` is set to this value, `bind_val` is added to the beginning of the `USING` clause. This produces the following statement:

```
DBMS WITH CURSOR cursor EXECUTE USING
 bind_parm = bind_val[occ],
 existing_parentTV_binds,
 existing_childTV_binds
```

`DM_GEN_REPLACE_ALL`
: When `flag` is set to this value, `bind_val` replaces the previous `USING` clause. This produces the following statement:

```
DBMS WITH CURSOR cursor EXECUTE USING
 bind_parm = bind_val[occ]
```

If `flag` is set to this value and the other arguments are empty strings, the `USING` clause is removed.

Returns

0	Success.
-1	Error: `dm_gen_sql_info` was not called.
-2	Error: Invalid flag.

Description

`dm_gen_change_execute_using` lets you edit the `USING` clause of a `DBMS EXECUTE` statement. The data structure for the `SELECT` statement, which is built by a call to `dm_gen_sql_info` (generally in the `TM_SEL_GEN` event), must already exist before this function is called. Note that this function must be called once for each bind value you wish to change.

Often, a call to `dm_gen_change_execute_using` follows a call to the function `dm_gen_change_select_where`. If new parameters are added to the `WHERE` clause's search conditions, those parameters must also be added to the `EXECUTE USING` statement.

This function can be implemented as part of a transaction manager event function that processes the `TM_SEL_BUILD_PERFORM` event. If you are modifying the select processing for a server view, call the `dm_gen_change_execute_using` function from an event function attached to the first parent table view in the server view.

To view a sample event function written in JPL, refer to the example in this section. For more information on writing transaction event functions, refer to Chapter 32, "Writing Transaction Event Functions," in *Application Development Guide*.

The settings for `relative` and `occurrence` determine the value for *`occ`*, the occurrence number used in the statement.

If `relative` is set to `DM_GEN_RELATIVE_TO_PARENT` or `DM_GEN_RELATIVE_TO_CHILD`, the current occurrence in the parent or child table view is used as the basis for the occurrence number. Then, the setting for `occurrence` is checked. If `occurrence` is 0, the current occurrence in that table view is used in the statement. If `occurrence` is greater than 0, the occurrence is calculated by adding the specified occurrence to the current occurrence.

If you only need to substitute an occurrence number in the statement processing, set `relative` to `DM_GEN_ABSOLUTE_OCCUR` and set `occurrence` to be greater than 0.

Example

```
# JPL Procedure:
   # Generate IN clause using binding parameters.
   # Function property is set to titles_exec.

proc titles_exec (event)
   if (event == TM_SEL_BUILD_PERFORM)
   {
     vars retval(5), occ(3), i(3), in_buffer(255), comma(1)

  occ = @widget("qbe_titleid")->num_occurrences

# If the array "qbe_titleid" contains data,
   # build a SQL "in" clause.

  if (occ > 0)

# First loop through qbe_titleid and build an IN clause
# in the form "title_id" in (::p1, ::p2, ::p3).
  {
    for i=1 while i <= occ
    {
      if (qbe_titleid[i] != "")
      {
        %.0 i = i
        in_buffer = in_buffer ## comma ## "\:\:p" ## i \
          comma = ","
      }
    }
    in_buffer = "title_id in (" ## in_buffer ## ")"
    retval = dm_gen_change_select_where \
      ("", in_buffer, DM_GEN_APPEND)

# Now loop through qbe_titleid and change the EXECUTE
# USING statement. This could be done in the previous loop.
# It is separated for clarity.

    for i=1 while i <= occ
    {
      if (qbe_titleid[i] != "")
      {
        %.0 i = i
```

```
            retval=dm_gen_change_execute_using \
              ('', "p:i", "qbe_titleid", i, \
              DM_GEN_ABSOLUTE_OCCUR, DM_GEN_APPEND)
          }
        }

        if (retval != 0)
        return TM_FAILURE
      }
    }
    return TM_PROCEED
```

Example The following example uses the current occurrence in the parent table view to specify the occurrence number. The parent table view in this sequential link is a list of customers. When you enter one of the `rental_status` codes for a customer in the `qbe_status` field, the rentals for that customer which match that status populate the child table view.

```
# JPL Procedure:
# Generate WHERE and EXECUTE USING clause using occurrence
# in parent table view. The Function property for rentals
# table view is set to rentals_hook.

proc rentals_hook(event)
{
  vars whexp(100) retval(5)
  if (event==TM_SEL_BUILD_PERFORM)
  {
  # Build the following: correlation.rental_status = ::qbe1
    whexp=dm_gen_get_tv_alias(sm_tm_pinquire(TM_TV_NAME)) \
      ## ".rental_status" \
      ## "=" \
      ## "::::qbe1"

  # Add it to the WHERE clause.
    retval = dm_gen_change_select_where("", whexp,\
      DM_GEN_APPEND)

  # Append to the EXECUTE USING clause in the form:
  # qbe1 = qbe_stat[<occ>]
  # where occ is the same occurrence number as the current
  # occurrence in parent table view.
    retval = dm_gen_change_execute_using\
      ("", "qbe1", "qbe_stat", \
      0, DM_GEN_RELATIVE_TO_PARENT, DM_GEN_APPEND)
  }
   return TM_PROCEED
}
```

See Also `dm_gen_sql_info`

dm_gen_change_select_from

Edits the FROM clause in a SELECT statement for automatic SQL generation

```
#include <tmusubs.h>
int dm_gen_change_select_from(char *arg, char *table,
    char *corr_name, int flag);
```

`arg`

Reserved for future use.

`table`

The name of the database table. For some database engines, you may need to include the owner name in the format:

`owner.table_name`

`corr_name`

The correlation name for the database table.

`flag`

Specifies the type of change to make with one of the following constants:

`DM_GEN_APPEND`

Adds the name of the database table and its associated correlation name to the end of the `FROM` clause. This produces the following statement:

```
DBMS DECLARE cursor FOR SELECT select_list FROM
 existing_from_clause,
 table corr_name
```

`DM_GEN_PREPEND`

Adds the name of the database table and its associated correlation name to the beginning of the `FROM` clause. This produces the following statement:

```
DBMS DECLARE cursor FOR SELECT select_list FROM
 table corr_name,
 existing_from_clause
```

`DM_GEN_REPLACE_ALL`

The name of the database table and its associated correlation name replace the previous `FROM` clause. This produces the following statement:

```
DBMS DECLARE cursor FOR SELECT select_list FROM
 table corr_name
```

Returns

0	Success.
-1	Error: `dm_gen_sql_info` was not called.
-2	Error: Invalid flag.

Description

`dm_gen_change_select_from` allows you to edit the tables listed in the `FROM` clause of a `SELECT` statement built with the SQL generator. The data structure for the `SELECT` statement, which is built by a call to `dm_gen_sql_info` (generally in the `TM_SEL_GEN` event), must already exist before this function is called. Note that this function must be called once for each table name you wish to change.

By default, the SQL generator builds the table list based on the `table` property of each table view in the server view. For more information on the SQL generator, refer to Chapter 33, “Using Automated SQL Generation,” in *Application Development Guide*.

This function can be implemented as part of a transaction manager event function which processes the `TM_SEL_BUILD_PERFORM` event. If you are modifying the select processing for a server view, call the `dm_gen_change_select_from` function from an event function attached to the first parent table view in the server view.

To view a sample event function written in JPL, refer to the example in this section. For more information on writing transaction event functions, refer to Chapter 32, “Writing Transaction Event Functions,” in *Application Development Guide*.

Example

```
# JPL Procedure:
   # Fetch data from titles which is an unlinked table view.
   # Function property is set to titles_join.

proc titles_join (event)

vars retval(5)

  if (event == TM_SEL_BUILD_PERFORM)
     {
     retval = dm_gen_change_select_list("", "name", "name", \
       DM_GEN_APPEND)

  retval = dm_gen_change_select_from \
       ("", "titles", "titles", DM_GEN_APPEND)

  retval = dm_gen_change_select_where ("", \
       "rentals.title_id = titles.title_id", DM_GEN_APPEND)
   if (retval != 0)
         return TM_FAILURE
       }

return TM_PROCEED
```

See Also `dm_gen_sql_info`

dm_gen_change_select_group_by

Edits the GROUP BY clause in a SELECT statement for automatic SQL generation

```
#include <tmusubs.h>
int dm_gen_change_select_group_by(char *arg, char *column,
   int flag);
```

`arg`

Reserved for future use.

`column`

The name of the column to be used in the `GROUP BY` clause.

`flag`

Specifies the type of change to make with one of the following constants:

`DM_GEN_APPEND`

Adds `column` to the end of the `GROUP BY` clause. This produces the following statement:

```
DBMS DECLARE cursor FOR SELECT select_list FROM
 tables GROUP BY existing_group_by_list, column
```

`DM_GEN_PREPEND`

Adds `column` to the beginning of the `GROUP BY` clause. This produces the following statement:

```
DBMS DECLARE cursor FOR SELECT select_list FROM
 tables GROUP BY column, existing_group_by_list
```

`DM_GEN_REPLACE_ALL`

`column` replaces the previous `GROUP BY` clause. This produces the following statement:

```
DBMS DECLARE cursor FOR SELECT select_list FROM
 tables GROUP BY column
```

If `flag` is set to this value and `column` is set to an empty string, the `GROUP BY` clause is removed. For example:

```
x = dm_gen_change_select_group_by
 ("", "", DM_GEN_REPLACE_ALL)
```

Returns

0	Success.
-1	Error: `dm_gen_sql_info` was not called.

-2 Error: Invalid flag.

Description

dm_gen_change_select_group_by allows you to edit the GROUP BY clause built with the SQL generator. The data structure for the SELECT statement, which is built by a call to dm_gen_sql_info (generally in the TM_SEL_GEN event), must already exist before this function is called. Note that this function must be called once for each change you wish to make.

By default, the SQL generator builds a GROUP BY clause automatically when one of the select expressions is an aggregate function. For more information on how the SQL generator builds statements, refer to Chapter 33, "Using Automated SQL Generation," in *Application Development Guide*.

This function can be implemented as part of a transaction manager event function which processes the TM_SEL_BUILD_PERFORM event. If you are modifying the select processing for a server view, call the dm_gen_change_select_group_by function from an event function attached to the first parent table view in the server view.

To view a sample event function written in JPL, refer to the example in this section. For more information on writing transaction event functions, refer to Chapter 32, "Writing Transaction Event Functions," in *Application Development Guide*.

Example

```
# JPL Procedure:
   # Append column not part of table view to automatically
   # generated group by clause.
   # Function property set to titles_group.

proc titles_group (event)

vars retval(5)

  if (event == TM_SEL_BUILD_PERFORM)
     {
       retval = dm_gen_change_select_list \
         ("", "rating_code", "rc", DM_GEN_APPEND)
       retval = dm_gen_change_select_group_by \
         ("", "rating_code", DM_GEN_APPEND)

    if (retval != 0)
         return TM_FAILURE
     }

return TM_PROCEED
```

See Also

dm_gen_sql_info

dm_gen_change_select_having

Edits the HAVING clause in a SELECT statement for automatic SQL generation

```
#include <tmusubs.h>
int dm_gen_change_select_having(char *arg, char *search_cond,
    int flag);
```

`arg`
: Reserved for future use.

`search_cond`
: The search condition to include in the `HAVING` clause.

`flag`
: Specifies the type of change to make with one of the following constants:

  `DM_GEN_APPEND`
  : Adds `search_cond` to the end of the `HAVING` clause. This produces the following statement:

    `DBMS DECLARE` *`cursor`* `FOR SELECT` *`select_list`* `FROM` *`tables`* `HAVING` *`existing_having_clause`* `AND` *`search_cond`*

  `DM_GEN_PREPEND`
  : Adds `search_cond` to the beginning of the `HAVING` clause. This produces the following statement:

    `DBMS DECLARE` *`cursor`* `FOR SELECT` *`select_list`* `FROM` *`tables`* `HAVING` *`search_cond`* `AND` *`existing_having_clause`*

  `DM_GEN_REPLACE_ALL`
  : `search_cond` replaces the existing `HAVING` clause. This produces the following statement:

    `DBMS DECLARE` *`cursor`* `FOR SELECT` *`select_list`* `FROM` *`tables`* `HAVING` *`search_cond`*

    If `flag` is set to this value and `search_cond` is set to an empty string, the `HAVING` clause is removed. For example:

    ```
    x = dm_gen_change_select_having
     ("", "", DM_GEN_REPLACE_ALL)
    ```

Returns

0	Success.
-1	Error: `dm_gen_sql_info` was not called.

-2 Error: Invalid flag.

Description

`dm_gen_change_select_having` lets you edit the `HAVING` clause built with the SQL generator. The data structure for the `SELECT` statement, which is built by a call to `dm_gen_sql_info` (generally in the `TM_SEL_GEN` event), must already exist before this function is called.

Generally, a `HAVING` clause sets search conditions for the preceding `GROUP BY` clause. The SQL generator creates `GROUP BY` clauses automatically for aggregate functions. `GROUP BY` clauses can also be generated using the function `dm_gen_change_select_group_by`. `HAVING` clauses can be generated with the Having property or by using this function. For more information on automatic SQL generation, refer to Chapter 33, "Using Automated SQL Generation," in *Application Development Guide*.

This function can be implemented as part of a transaction manager event function which processes the `TM_SEL_BUILD_PERFORM` event. If you are modifying the select processing for a server view, call `dm_gen_change_select_having` from an event function attached to the first parent table view in the server view.

To view a sample event function written in JPL, refer to the example in this section. For more information on writing transaction event functions, refer to Chapter 32, "Writing Transaction Event Functions," in *Application Development Guide*.

Example

```
# JPL Procedure:
   # Generate a having clause.
   # Function property is set to titles_having.

proc titles_having (event)

vars retval(5)

  if (event == TM_SEL_BUILD_PERFORM)
     {
     retval = dm_gen_change_select_having\
       ("", "count(*) > 2", DM_GEN_APPEND)

  retval = dm_gen_change_select_having\
       ("", "dir_last_name like 'W%'", DM_GEN_APPEND)

  if (retval != 0)
       return TM_FAILURE
     }

return TM_PROCEED
```

See Also `dm_gen_sql_info`

dm_gen_change_select_list

Edits the select list for automatic SQL generation

```
#include <tmusubs.h>
int dm_gen_change_select_list(char *arg, char *sel_expr,
   char *prolfx_alias, int flag);
```

`arg`

Reserved for future use.

`sel_expr`

The select expression. If the expression is invalid, the engine returns an error.

`prolfx_alias`

Name of the Panther variable to use in the `DBMS ALIAS` statement. This variable should not be a local JPL variable. If this variable does not exist or is blank, the `SELECT` statement fetches the expression's values, but they are ignored. This is not considered an error.

`flag`

Specifies the type of change to make with one of the following constants:

`DM_GEN_APPEND`

Adds `sel_expr` to the end of the select list. `prolfx_alias` is added after the existing aliases. This produces the following statements:

```
DBMS DECLARE cursor FOR SELECT existing_select_list,
 sel_expr FROM ...
DBMS WITH CURSOR cursor ALIAS existing_aliases,
 prolfx_alias
```

`DM_GEN_PREPEND`

Adds `sel_expr` to the beginning of the select list, and `prolfx_alias` is added before the existing aliases. This produces the following statements:

```
DBMS DECLARE cursor FOR SELECT sel_expr,
 existing_select_list FROM ...
DBMS WITH CURSOR cursor ALIAS prolfx_alias,
 existing_aliases
```

`DM_GEN_REPLACE_ALL`

`sel_expr` replaces the previous select list, and `prolfx_alias` replaces the existing aliases. This produces the following statements:

```
DBMS DECLARE cursor FOR SELECT sel_expr FROM ...
DBMS WITH CURSOR cursor ALIAS prolfx_alias
```

Returns

0	Success.
-1	Error: `dm_gen_sql_info` was not called.
-2	Error: Invalid flag.

Description

`dm_gen_change_select_list` allows you to edit the select list built using the SQL generator. The data structure for the `SELECT` statement, which is built by a call to `dm_gen_sql_info` (generally in the `TM_SEL_GEN` event), must already exist before this function is called. You must call this function once for each change you wish to make.

By default, the SQL generator builds the select list from the widgets whose `use_in_select` property is set to `PV_YES`. For more information on the SQL generator, refer to Chapter 33, "Using Automated SQL Generation," in *Application Development Guide*.

This function can be implemented as part of a transaction manager event function that processes the `TM_SEL_BUILD_PERFORM` event. If you are modifying the select processing for a server view, call `dm_gen_change_select_list` from an event function attached to the first parent table view in the server view.

For more information on transaction event functions, refer to Chapter 32, "Writing Transaction Event Functions," in *Application Development Guide*.

Example

```
# JPL Procedure:
   # Adds pic1, a binary column, to the select list for the
   # current server view and sets bin_col1 as the target.
   # The Function property is set to binary_hook.

proc binary_hook (event)
   {
   vars retval(5) colexp(64)

if (event==TM_SEL_BUILD_PERFORM)
     {
       colexp=dm_gen_get_tv_alias\
         (sm_tm_pinquire(TM_TV_NAME) ## ".pic1")
       retval=dm_gen_change_select_list\
         ("", colexp, "bin_col1", DM_GEN_APPEND)

# The number of occurrences for bin_col1 is set to match the
   # number of occurrences of another column in the table.
```

```
         if (retval == 0)
         {
           retval=sm_n_max_occur("name")
           dbms binary bin_col1[:retval](1024)
         }
       }
       return TM_PROCEED}
```

See Also `dm_gen_sql_info`

dm_gen_change_select_order_by

Edits the ORDER BY clause in a SELECT statement for automatic SQL generation

```
#include <tmusubs.h>
int dm_gen_change_select_order_by(char *arg, char *widget_name,
   int sort_ind, int flag);
```

`arg`

Reserved for future use.

`widget_name`

The name of the widget whose `column_name` property is referenced in the `ORDER BY` clause. If the name of the database column is entered, it is ignored.

`sort_ind`

Specifies whether the sort is ascending (`DM_GEN_ASC_SORTED`) or descending (`DM_GEN_DESC_SORTED`). If set to an invalid value, an error is generated.

`flag`

Specifies the type of change to make with one of the following constants:

`DM_GEN_APPEND`

Adds the specified information to the end of the `ORDER BY` clause. This produces the following statement:

```
DBMS DECLARE cursor FOR SELECT select_list
 FROM tables ORDER BY existing_order_by_list,
 column_position sort_ind
```

`DM_GEN_PREPEND`

Adds the specified information to the beginning of the `ORDER BY` clause. This produces the following statement:

```
DBMS DECLARE cursor FOR SELECT select_list
 FROM tables ORDER BY column_position sort_ind,
 existing_order_by_list
```

`DM_GEN_REPLACE_ALL`

The specified information replaces the previous `ORDER BY` clause. This produces the following statement:

```
DBMS DECLARE cursor FOR SELECT select_list
 FROM tables ORDER BY column_position sort_ind
```

If `flag` is set to this value and `widget_name` is set to an empty string, the `ORDER BY` clause is removed. For example:

```
x = dm_gen_change_select_order_by
  ("", "", "", DM_GEN_REPLACE_ALL)
```

Returns

0	Success.
-1	Error: `dm_gen_sql_info` was not called.
-2	Error: Invalid flag.

Description

`dm_gen_change_select_order_by` lets you edit the `ORDER BY` clause built with the SQL generator. The structure for the `SELECT` statement, which is built by a call to `dm_gen_sql_info` (generally in the `TM_SEL_GEN` event), must already exist before this function is called. Note that this function must be called once for each change you wish to make.

By default, the SQL generator builds the `ORDER BY` clause from values of the table view's Sort Widgets (`sort_widgets`) property. For more information on how the SQL generator builds statements, refer to Chapter 33, "Using Automated SQL Generation," in *Application Development Guide*.

This function can be implemented as part of a transaction manager event function which processes the `TM_SEL_BUILD_PERFORM` event. If you are modifying the select processing for a server view, call `dm_gen_change_select_order_by` from an event function attached to the first parent table view in the server view.

To view a sample event function written in JPL, refer to the example in this section. For more information on writing transaction event functions, refer to Chapter 32, "Writing Transaction Event Functions," in *Application Development Guide*.

Example

```
# Appends the order by list for titles table.
   # The Function property is set to titles_orderby.

proc titles_orderby (event)

vars retval(5)

if (event == TM_SEL_BUILD_PERFORM)
     {
     retval = dm_gen_change_select_order_by \
       ("", "film_minutes", DM_GEN_ASC_SORTED, DM_GEN_APPEND)

  if (retval != 0)
  return TM_FAILURE
  }

return TM_PROCEED
```

See Also `dm_gen_sql_info`

dm_gen_change_select_suffix

Appends text to the end of a SELECT statement for automatic SQL generation

```
#include <tmusubs.h>
int dm_gen_change_select_suffix(char *arg, char *suffix);
```

`arg`
: Reserved for future use.

`suffix`
: The suffix to append to the generated `SELECT` statement.

Returns

0	Success.
-1	`dm_gen_sql_info` was not called.
-2	Invalid flag.

Description

`dm_gen_change_select_suffix` lets you append text to the end of a generated `SELECT` statement built with the SQL generator. For example, you can use this function to add a `FOR UPDATE` clause to the end of a `SELECT` statement. The data structure for the `SELECT` statement, built by an earlier call to `dm_gen_sql_info` (generally in the `TM_SEL_GEN` event), must already exist before this function is called.

By default, the SQL generator builds the statement based on the widgets' and table view's property settings. For more information on the SQL generator, refer to Chapter 33, "Using Automated SQL Generation," in *Application Development Guide*.

You can use this function in a transaction manager event function that processes the `TM_SEL_BUILD_PERFORM` event. To modify the select processing for a server view, call `dm_gen_change_select_suffix` from an event function that is attached to the first parent table view in the server view.

For more information on writing transaction event functions, refer to Chapter 32, "Writing Transaction Event Functions," in *Application Development Guide*.

Example

```
# JPL Procedure:
    # Fetch data from titles for possible update.
    # Function property is set to titles_select.

proc titles_select (event)

vars retval(5)
```

```
  if (event == TM_SEL_BUILD_PERFORM)
      {
      retval = dm_gen_change_select_suffix("", "for update")

  if (retval != 0)
        return TM_FAILURE
      }

return TM_PROCEED
```

See Also `dm_gen_sql_info`

dm_gen_change_select_where

Edits the WHERE clause in a SELECT statement used in automatic SQL generation

```
#include <tmusubs.h>
int dm_gen_change_select_where(char *arg, char *where_expr,
   int flag);
```

`arg`
: Reserved for future use.

`where_expr`
: Text of the expression to include in the `WHERE` clause. If the expression includes a parameter and the function is called within a JPL procedure, the parameter name must be declared with four colons because of colon expansion (`::::parm1`).

`flag`
: Specifies the type of change to make with one of these constants:

`DM_GEN_APPEND`
: When `flag` is set to this value, `where_expr` is added to end of the `WHERE` clause. This produces the following statement:

```
DBMS DECLARE cursor FOR SELECT select_list
 FROM table_list WHERE link_expression
 AND existing_where_expr AND where_expr
```

`DM_GEN_PREPEND`
: Adds `where_expr` to the beginning of the expressions derived from the `use_in_where` property. This produces the following statement:

```
DBMS DECLARE cursor FOR SELECT select_list
 FROM table_list WHERE link_expression
 AND where_expr AND existing_where_expr
```

`DM_GEN_REPLACE_ALL`
: Removes all the expressions based on the `use_in_where` property being `PV_YES` and `where_expr` replaces the previous data. This produces the following statement:

```
DBMS DECLARE cursor FOR SELECT select_list
 FROM table_list WHERE link_expression AND where_expr
```

You also need to call `dm_gen_change_execute_using` to remove the *`existing_where_expr`* from the `USING` clause of the `EXECUTE` statement.

Returns

0	Success.
-1	Error: `dm_gen_sql_info` was not called.
-2	Error: Invalid flag.

Description

`dm_gen_change_select_where` lets you edit the `WHERE` clause of a `SELECT` statement. The structure for the `SELECT` statement, which is generally built by a call to `dm_gen_sql_info` in the `TM_SEL_GEN` event, must already exist before `dm_gen_change_select_where` is called.

By default, the data for the `WHERE` clause comes from:

- Widgets whose `use_in_where` property is set to `PV_YES`.
- The `relations` property for the link which determines the columns for joins if it is a server link and for master/detail information if it is a sequential link.

`dm_gen_change_select_where` adds to or replaces the data based on the `use_in_where` property. For more information on how the SQL generator uses this property, refer to Chapter 33, “Using Automated SQL Generation,” in *Application Development Guide*.

In particular, this function can be used to add a `BETWEEN` clause or a subquery to a `SELECT` statement.

This function can be implemented as part of a transaction manager event function which processes the `TM_SEL_BUILD_PERFORM` event. If you are modifying the select processing for a server view, call `dm_gen_change_select_where` from an event function attached to the first parent table view in the server view.

To view a sample event function written in JPL, refer to the example in the next section. For more information on writing transaction event functions, refer to Chapter 32, “Writing Transaction Event Functions,” in *Application Development Guide*.

Example

```
# JPL Procedure:
    # Append IN clause to WHERE clause.
    # Function property is set to titles_in.

proc titles_in (event)
```

```
vars retval(5)

if (event == TM_SEL_BUILD_PERFORM)
{
vars occ(3), i(3), in_buffer(255) comma(1)

occ = @widget("qbe_titleid")->num_occurrences

# If the array "qbe_titleid" contains data, build
# a SQL "in" clause.
  if (occ > 0)
  {
  for i=1 while i <= occ
  {
    if (qbe_titleid[i] != "")
    {
       in_buffer = in_buffer ## comma ## \
        ':+qbe_titleid[i]'
       comma = ","
    }
  }

  in_buffer = "title_id in (" ##in_buffer ")"
  retval = dm_gen_change_select_where \
    ("", in_buffer, DM_GEN_APPEND)

  if (retval != 0)
  return TM_FAILURE
  }
}
return TM_PROCEED

# JPL Procedure:
# Append search condition using onscreen value.
# Function property is set to titles_where.

proc titles_where (event)

vars retval(5)

  if (event == TM_SEL_BUILD_PERFORM)
  {
  retval = dm_gen_change_select_where\
    ("", "film_minutes > ::::parm1", DM_GEN_APPEND)
  retval = dm_gen_change_execute_using("", "parm1", \
    "film_minutes", 1, DM_GEN_ABSOLUTE_OCCUR, \
    DM_GEN_APPEND)
    if (retval != 0)
    return TM_FAILURE
    }

return TM_PROCEED
```

See Also dm_gen_sql_info, dm_gen_change_execute_using

dm_gen_get_tv_alias

Gets the correlation name or alias for a table view

```
#include <tmusubs.h>
char *dm_gen_get_tv_alias(char *tv_name);
```

`tv_name`
Specifies the table view name.

Returns

- A correlation name for the table view
- `NULL` string: `tv_name` is null.

Description

`dm_gen_get_tv_alias` returns the correlation name, or alias, for the specified table view name.

Generally, the SQL generator uses the value in the table view's Name property as the table's correlation name in a generated `SELECT` statement. However, if the table view name contains illegal characters for a correlation name, the SQL generator removes the offending characters.

The SQL generator calls this function to generate correlation names. If you modify generated SQL statements with one of the `dm_gen_change` functions and any argument supplies a column name, you must supply the proper correlation name.

Example

```
# JPL Procedure:
   # Adds a column to the select list for the current \
   # server view and sets copy as the target.

proc rentals_hook(event)
   {
   vars retval(5) colexp(64)

if (event==TM_SEL_BUILD_PERFORM)
     {
       colexp=dm_gen_get_tv_alias\
         (sm_tm_pinquire(TM_TV_NAME) ## ".copy_num")

       retval=dm_gen_change_select_list\
         ("", colexp, "copy", DM_GEN_APPEND)
     }
     return TM_PROCEED
   }
```

dm_gen_sql_info

Generates a data structure used in SELECT statement generation

```
#include <tmusubs.h>
int dm_gen_sql_info(int type, char *cursor_name);
```

`type`
: Type of `SELECT` to generate, specified by one of these constants:

```
SELECT
VALIDATE
CHECK_PKEY
```

`cursor_name`
: Name of the cursor associated with the SQL statement.

Returns

0	Success.
<0	One of the transaction error codes.

Description

`dm_gen_sql_info` generates a data structure associated with `SELECT` statements. The type is `SELECT` when the function is called as the result of the transaction commands `SELECT` and `VIEW`. The `type` is `VALIDATE` when the function is called as a result of processing a validation link. The type is `CHECK_PKEY` when the function is called as a result of checking for duplicate key values before inserting a new row or updating the primary key columns.

Example

```
int gen_select (cursor)
    char *cursor;
    {
      int retcode;
      retcode = dm_gen_sql_info(SELECT, cursor1);
      ...
      return retcode;
    }
```

See Also

`dm_free_sql_info`

dm_get_connection_option

Gets a database connection option

```
#include <dmuproto.h>
int dm_get_connection_option(char *connection, char *option);
```

`connection`
The name of the connection.

`option`
The option whose value is to be returned.

Returns

- value of the `option`.
- `-1`: `connection` does not exist or `option` is invalid.

Description

`dm_get_connection_option` gets the value of a connection option. The valid values of `option` depend on the database engine that the connection is to. Table 5-5 lists the values that can be used.

See Also

`dm_set_connection_option`

dm_get_db_conn_handle

Gets a handle to a database connection logon structure

```
#include <dmuproto.h>
int dm_get_db_conn_handle_handle(char *connection, void *handle,
    int size);
```

`connection`
: Name of the database connection.

`handle`
: Pointer to the connection structure.

`size`
: Size of the handle.

Environment	C only

Returns

0	Success.
-1	A `NULL` handle.
-2	Named connection not found.
-3	Invalid handle.
-4	Size of the handle differs from value specified in `size`.

Description

`dm_get_db_conn_handle` obtains a handle to the logon data structure for a named database connection. This information can be used in database engine programs that need information about the Panther database connection.

See Also

`dm_get_db_cursor_handle`

dm_get_db_cursor_handle

Gets a handle to a database cursor's structure

```
#include <dmuproto.h>
int dm_get_db_cursor_handle(char *name, void *handle, int size);
```

`name`
: Name of the database cursor.

`handle`
: Pointer to the cursor structure.

`size`
: Size of the handle.

Environment C only

Returns

0	Success.
-1	A `NULL` handle.
-2	Cursor not found.
-3	Invalid handle.
-4	Size of the handle differs from value specified in `size`.

Description `dm_get_db_cursor_handle` obtains a handle to a copy of a database cursor's structure pointer. This information can be used in database engine programs that need information about the Panther database cursors.

If name is NULL or an empty string, the default cursor is used. If the named cursor is found, the support routine is called to retrieve the cursor handle.

See Also `dm_get_db_conn_handle`

dm_get_driver_option

Gets a database driver option

```
#include <dmuproto.h>
int dm_get_driver_option(char *engine, char *option);
```

`engine`
: The name of the database engine.

`option`
: The option whose value is to be returned.

Returns

- value of the `option`.
- `DM_NODATABASE` if `engine` is not the name of an installed database engine.
- `-1`: Invalid `option`.

Description

`dm_get_driver_option` gets the value of a database engine's generic connection options. The valid values of `option` depend on the database engine. Table 5-6 lists the values that can be used.

See Also

`dm_set_driver_option`

dm_getdbitext

Gets the text of the last-executed DBMS command

```
#include <dmuproto.h>
char *dm_getdbitext(void);
```

Environment C only

Returns A pointer to the last-executed database command.

Description `dm_getdbitext` lets you get the full text of the last-executed `DBMS` command. This includes all commands executed from JPL with `dbms`, or executed from C with `dm_dbms` or `dm_dbms_noexp`.

You must call this function from within an installed entry, error, or exit handler. This function stores the data in a pool of buffers that it shares with other functions, so you must either process the returned string immediately or copy it to another variable for additional processing.

This function gets the same string that is passed to the first argument of an installed entry, error, or exit handler; however, these handlers are limited to 255 characters.

Example

```
int
logfunc PARMS((stmt, engine, flag))
PARM    (char *stmt)
PARM    (char *engine)
LASTPARM(int flag)
{
      FILE *fp;
      if (strlen(stmt) >= 255)
         stmt = dm_getdbitext();
      fp = fopen("dbi.log", "a");
      fprintf(fp, "%s\n", stmt);
      fclose(fp);
      return 0;
}
```

See Also `DBMS ONERROR`, `DBMS ONEXIT`

dm_init

Initializes access to a specific database engine

```
#include <dmuproto.h>
int dm_init(char *engine, int support_function, int case,
   char *arg);
```

`engine`

A name you assign to the engine. If an application uses two or more engines, the application uses the mnemonic `engine` to indicate a particular DBMS. Most examples in the guide use a vendor name as the mnemonic, for example `sybase` or `oracle`, but any character string that is not a keyword is valid. For a list of keywords, refer to Chapter 13, "Keywords in Database Drivers." If `engine` is already installed, `dm_init` returns 0.

`support_function`

One of the function names documented in the `dbiinit.c` file. The file name is usually in the form `dm_`*`vendor`*`sup` where *`vendor`* is an abbreviated vendor name. For example:

```
dm_sybsup
dm_orasup
dm_intsup
```

`case`

Sets the case processing for the specified engine. The constants are shown in Table 5-4 in Description.

`arg`

Reserved for future use. Set this parameter to 0.

Environment C only

Returns

- 0: Success.
- A return code from the support function.

Description Before an application can access a database, Panther must perform an engine initialization. The initialization adds the engine name to the list of available engines, allocates a data structure for the engine, calls the engine's support function to initialize the data structure, and sets case handling for the engine. You can use the `vendor_list` structure in `dbiinit.c` to initialize an engine at startup or else use `dm_init` to initialize an engine at a later point in the application.

The `case` parameter specifies how Panther uses case to map column names to variables when executing a `SELECT` statement. Table 5-4 lists the available options.

Table 5-4 Database engine case constants

Constant	Description
`DM_DEFAULT_CASE`	Use the case option set in the support function for that engine. For information on this setting, refer to the documentation for "Database Drivers."
`DM_PRESERVE_CASE`	Use case exactly as returned by the engine.
`DM_FORCE_TO_UPPER_CASE`	Force all column names returned by an engine to upper case. Therefore, the application should use upper case names for Panther variables.
`DM_FORCE_TO_LOWER_CASE`	Force all column names returned by an engine to lower case. Therefore, the application should use lower case names for Panther variables.

After the engine is initialized, the application can declare a connection on it.

Example

```
#include <dmerror.h>
#include <smusrdbi.h>

int retcode;

   retcode = dm_init("jdb", dm_jdbsup, DM_DEFAULT_CASE, 0);
```

See Also `dm_reset`

dm_is_connection

Verifies that a connection is open

```
#include <dmuproto.h>
int dm_is_connection(char *connection_name);
```

`connection_name`
: Specifies a connection name that is declared in a `DBMS DECLARE CONNECTION` command.

Returns

1 True: Connection exists.

0 False: Connection does not exist, either because it was never declared or was closed.

Example

```
#include <smdefs.h>
#include <dmuproto.h>

int free_resources()
    {

  if (dm_is_connection("work_connection"))
      {
        dm_dbms("close connection work_connection");
      }
      return 0
    }
```

dm_is_cursor

Verifies that a cursor is open

```
#include <dmuproto.h>
int dm_is_cursor(char *cursor_name);
```

`cursor_name`
: Specifies a cursor name. For a named cursor, use the name specified in a `DBMS DECLARE CURSOR` command. For a default cursor, specify `cursor_name` as being `default_cursor` or as being `0`.

Returns

1 The cursor exists.

0 The cursor does not exist, either because it was never declared or has been closed.

Example

```
#include <smdefs.h>
#include <dmuproto.h>

int free_resources()
   {

     if (dm_is_cursor("work_cursor"))
     {
       dm_dbms("close cursor work_cursor");
     }
     return 0
   }
```

dm_is_engine

Verifies that a database engine is initialized

```
#include <dmuproto.h>
int dm_is_engine(char *engine);
```

`engine`
: Specifies an engine name. The engine name is the character string assigned to a database engine in the `dbiinit.c` or Windows initialization file. For more information about specifying engine names, refer to Chapter 8, “Connecting to Databases,” in *Application Development Guide*.

Returns

1 True: Engine is initialized.
0 False: Engine is not initialized.

Example

```
// Test if engine was installed

#include <smdefs.h>
   #include <dmuproto.h>

int
   eng_connection()
   {

     if (dm_is_engine("sybase"))
     {
       dm_dbms("engine sybase");
       dm_dbms("declare c1 connection for ...");
     }
     return 0
   }
```

dm_odb_preserves_cursor

Checks if the ODBC datasource preserves the cursor on a commit or rollback

```
int dm_odb_preserves_cursor(void);
```

Returns

≥1	Datasource preserves cursor on both a commit and a rollback.
0	Cursor is not preserved.

Description

`dm_odb_preserves_cursor` checks to see whether the ODBC datasource preserves the cursor on a commit or a rollback. Unless the datasource is ascertained to preserve the cursor on both operations, this routine returns that the cursor is not preserved.

dm_reset

Disables support for a named database engine

```
#include <dmuproto.h>
int dm_reset(char *engine);
```

`engine`
: The name assigned to the DBMS in `dm_init` or in the `vendor_list` structure of `dbiinit.c`.

Environment C only

Returns

0	The database engine was successfully disabled.
-1	`engine` is not a valid engine name.

Description An application can call this function to disable support for a named engine. If the function executes successfully, it performs the following steps:

1. Closes all active connections on the engine.
2. Calls the support function to perform any engine-specific reset processing.
3. If `engine` was the default engine, sets the default engine to 0.
4. Frees all data structures associated with the engine.

After an engine is reset, the application cannot connect to the engine unless it initializes the engine with `dm_init`.

See Also `dm_init`

dm_set_connection_option

Sets a database connection option

```
#include <dmuproto.h>
int dm_set_connection_option(char *connection, char *option,
                                        int value);
```

`connection`
The name of the connection.

`option`
The option to be set.

`value`
The option's new value.

Returns

- 0: Success.
- `-1`: `connection` does not exist or `option` or `value` is invalid.

Description `dm_set_connection_option` sets a database connection option to a new value. See Table 5-5 below for a list of the values of `option` that are supported for each driver.

Table 5-5 Connection options by driver

Option	Value
Informix driver	
`xa_connection*`	`1` for XA connection; `0` for normal connection.
SQL Server driver	
`cursor_pool_size`	Allow the client code to reuse closed database connections in a pool to reduce overhead when making new connections. The default value is zero, which disables this feature. New in Panther 5.10.
ODBC driver	
`bind_set_scale`	When not zero, the scale is always set when binding numeric columns. Set when connecting to DB2. New in Panther 5.10.

Table 5-5 Connection options by driver

Option	Value
`case_flag`	Allows setting the case flag for a connection. `value` can be one of `PV_LOWER`; `PV_UPPER` or `PV_MIXED`. New in Panther 5.50.
`count_decimal_digits`	When set, the scale will be set when numeric values are bound as character strings. Set when connecting to SQL Server. New in Panther 5.00.
`force_char_binding`	When this option is set, if the bind type is `SQL_VARCHAR`, it is set to `SQL_CHAR`. Set when connecting to Oracle and Informix. New in Panther 4.60.
`force_date_convert`	When set, date fields will always be bound in the internal ODBC date format even when character string binding is requested. Set when connecting to Oracle. New in Panther 4.60.
`long_date_bind`	When set, uses at least 19 characters when binding `SQL_TIMESTAMP` columns. Set when connecting to SQL Server. New in Panther 5.10.
`oracle_empty_string`	Special null string handling for Oracle. Set when connecting to Oracle. New in Panther 4.60.
`set_concurrency`	Value for the `SQL_ATTR_CONCURRENCY` ODBC statement handle attribute. Set to `SQL_CONCUR_VALUES` when connecting to SQL Server. New in Panther 5.10.
Oracle OCI driver	
`no_utf8_conversion`	Can only be set for UTF8 connections. When set, the application must convert character date from and to UTF8. New in Panther 5.40.
`utf8*`	1 if the `UTF8` option was specified in the `DECLARE CONNECTION` statement for the connection. New in Panther 5.40.
`xa_connection*`	`1` for XA connection; `0` for normal connection.
Oracle Pro*C driver	
`xa_connection*`	`1` for XA connection; `0` for normal connection.

Table 5-5 Connection options by driver

Option	**Value**
Sybase CT Library driver	
`xa_connection*`	`1` for XA connection; `0` for normal connection.
*Option is read-only..	

See Also   `dm_get_connection_option`

dm_set_driver_option

Sets a database driver option

```
#include <dmuproto.h>
int dm_set_driver_option(char *engine, char *option, int value);
```

`engine`
: The name of the engine.

`option`
: The option to be set.

`value`
: The option's new value.

Returns

- 0: Success.
- `DM_NODATABASE` if `engine` is not the name of an installed database engine.
- `-1`: Invalid `option` or `value`.

Description

`dm_set_connection_option` sets a database engine's generic connection options to a new value. See Table 5-6 for a list of the values of `option` that are supported.

Table 5-6 Driver options by driver

Option	Value
Informix driver	
`select_numeric_as_string`	Set to nonzero to select numeric values as strings to avoid rounding errors. New in Panther 5.00.
ODBC driver	
`enable_mars`	When used with a Microsoft SQL Server ODBC driver, enables the MARS (Multiple Active Result Sets) feature. New in Panther 5.20.
`extended_fetch`	When zero, the ODBC `SQLExtendedFetch` function will not be used even if it is supported by the driver. New in Panther 4.50.

Table 5-6 Driver options by driver

Option	Value
`force_date_convert`	When set, date fields will be bound with the ODBC date format rather as strings. New in Panther 4.60.
`long_escape_seq`	ODBC uses escape sequences to add certain features such as date literals to SQL. These escape formats have two formats, long and short. Long escape sequences are deprecated in ODBC 3.
`new_date_binding`	If set (the default), date fields will be bound depending on their date format. If not set, date fields will always be bound with the date/time format.
`no_more_rows_value`	If not zero (the default), `@dmengerrcode` will be set to this value when `@dmretcode` is set to `DM_NO_MORE_ROWS`. New in Panther 5.51.
Oracle OCI driver	
`blank_as_null`	The OCI Library treat the null string `''` as the NULL value. By default, this driver uses the string `' '` instead to keep this from happening. 0 does this; 2 uses `''`; any other value used `NULL`.
`neg_err_mode`	When set, positive error codes other than `END_OF_FETCH` are returned as negative error codes. This option affects the value of `@dmengerrcode` when an engine error is reported.
`select_numeric_as_double`	Set to nonzero to select values as doubles when retrieving numeric arrays from stored procedures. New in Panther 5.00.
`select_numeric_as_string`	Set to nonzero to select numeric values as strings to avoid rounding errors. New in Panther 4.10.
`v2_mode`	Use old Oracle error codes if not `0`. For backwards compatibility with older JAM code.
Oracle Pro*C driver	
`blank_as_null`	The Pro*C Library treat the null string `''` as the NULL value. By default, this driver uses the string `' '` instead to keep this from happening. 0 does this; 2 uses `''`; any other value uses `NULL`.

Table 5-6 Driver options by driver

Option	Value
`select_numeric_as_double`	Set to nonzero to select values as doubles when retrieving numeric arrays from stored procedures. New in Panther 5.00.
`select_numeric_as_string`	Set to nonzero to select numeric values as strings to avoid rounding errors. New in Panther 4.10.
Sybase CT Library driver	
`expand_bind`	Who will expand bind markers in SQL statements: `0` Sybase expands bind markers (default); `1` Panther will expand bind markers if Sybase cannot; `2` Panther will expand bind markers.
`old_display_lengths`	Compatibility for column widths with the `CATQUERY` command before Panther 4.10: `0` DB Library compatible; `1` older version compatible.
`xa_connection*`	Type of the default connection: `1` for XA connection; `0` for normal connection; `-1` if no connection.
Sybase DB Library driver	
`allow_password_change`	Normally the `DECLARE CONNECTION` will fail if the password has expired. Setting this option to `1` will allow the connection so that the Sybase `sp_password` script can be run. New in Panther 5.10.
`msg_severity_threshhold`	Normally severity 10 Sybase server messages are treated by the driver as fatal errors. Setting this option to `11` will change this behavior. See the Sybase documentation for more information.
`secure_login`	Set to non-zero to encrypt the password in `DECLARE CONNECTION` processing. New in Panther 5.40.

*Option is read-only..

See Also `dm_get_driver_option`

dm_set_max_fetches

Sets the maximum number of rows in a select set

```
#include <dmuproto.h>
int dm_set_max_fetches(int count);
```

`count`
: Maximum number of rows to be in the select set. If `-1`, return the current setting.

Returns The new maximum number of rows. If `count` is `-1`, the current value.

Description `dm_set_max_fetches` determines the maximum number of rows that will be retrieved from a `SELECT` statement or `DBMS CONTINUE` command. If `-1` is passed in as the value, the function returns the current value; otherwise, it returns the new maximum value.

Initially, the maximum number of fetches is determined by the default value of the `max_fetches` application property, which is 1,000. This property provides an alternative way to modify the maximum number of fetches.

See Also `dm_set_max_rows_per_fetch`

dm_set_max_rows_per_fetch

Sets the maximum number of rows per fetch

```
#include <dmuproto.h>
int dm_set_max_rows_per_fetch(int count);
```

`count`
: Maximum number of rows per fetch. If `-1`, return the current setting.

Returns The new maximum number of rows per fetch. If `count` is `-1`, the current value.

Description `dm_set_max_rows_per_fetch` sets the maximum number of rows per fetch. The default (and maximum) value is `1000`. This affects the number of rows retrieved on each fetch request, not the number of rows retrieved by a `SELECT` statement or `DBMS CONTINUE` command. Use this function in order to optimize your application on a specific platform.

The `max_rows_per_fetch` application property also sets this value.

See Also `dm_set_max_fetches`

dm_set_onevent

Install a C DBi event hook function

```
int dm_set_onevent(char *handler);
```

`handler`
: The name of the DBi event handler. If `handler` is the null pointer or the null string, the DBi event handler is uninstalled. Otherwise, it must be the name of a C function installed in the prototyped function list.

Returns

0	The DBi event hook function was installed or uninstalled.
-1	`handler` is not in the list of installed prototyped C functions.

Description

`handler` is called after the `DBMS ONEXIT` function is or would be called. The prototype of this function is:

```
void handler(char *command, char *args, char *sql,
             int elapsed_time);
```

This function is passed the following parameters:

`command`
: A string containing the DBi command, for example `"ALIAS"` for the `DBMS ALIAS` command.

`arg`
: The argument passed to the `DBMS` command. It is same as the value that would be returned by the `dm_getdbitext` function.

`sql`
: If the `DBMS` statement includes the `WITH CURSOR` clause, this value is the text of the SQL that was in the `DBMS DECLARE CURSOR` statement that created the cursor. Otherwise it is the null pointer.

`elapsed_time`
: This is the elapsed time for the `DBMS` command in milliseconds. It may be zero for commands like `DBMS ALIAS` that do not require any database interactions.

dm_set_tm_clear_fast

Determines the behavior of the CLEAR command in transaction manager

```
#include <tmusubs.h>
int dm_set_tm_clear_fast(int clear_setting);
```

`clear_setting`
Setting for transaction manager `CLEAR` operations:

1	True: Clear data by server view.
0	False: Clear data by table view.
-1	The current setting.

Returns The current value of the `tm_clear_fast` property.

Description `dm_set_tm_clear_fast` determines how the transaction manager clears data in table views. By default, the transaction manager clears data by table view. To have the transaction manager clear data by server view, change the value of the `tm_clear_fast` application property by calling this function (or setting the property at runtime).

Do not call this function while the transaction manger is traversing table views for a `CLEAR` command. The current setting applies to the entire application; you cannot apply the setting per table view.

If you have added widgets to the synchronization group that are not part of a table view, they will be cleared.

dm_val_relative

Sets bits for validation after SELECT statements are executed

```
#include <tmusubs.h>
void dm_val_relative(void);
```

Description

`dm_val_relative` sets validated bits, and can be called after successful lookup/validation when using validation links. Because this function uses the data structure generated by `dm_gen_sql_info` for validation, you should call `dm_val_relative` before calling `dm_free_sql_info` to free the data.

sm_adjust_area

Recalculates widget positions

```
void sm_adjust_area(void);
```

Environment

Motif, Windows

Description

`sm_adjust_area` recalculates the positions of widgets on the current screen and redraws the screen accordingly. It uses Panther's positioning algorithm to map character-mode coordinates to the current GUI environment. You should call this function when runtime changes to the screen might cause widgets to overlap—for example, move a widget, add a new one, or change widget dimensions.

sm_allget

Loads data from the active LDBs to the current screen

```
void sm_allget(int respect_flag);
```

`respect_flag`
Indicates whether to write to fields that already contain data:

0	Initialize all fields, regardless of their status.
≥1	Initialize only empty or unmodified fields.

Description

`sm_allget` copies data from the active local data blocks to fields on the current screen with matching names. Panther calls this function automatically unless LDB processing is turned off through `sm_dd_able`.

`sm_allget` overwrites or respects existing data according to the value of `respect_flag`. `sm_allget` leaves unchanged the `mdt` property of the fields that it initializes.

Example

```
#include <smdefs.h>
   #include <smkeys.h>

   /* If you open a window with sm_r_window and want named
    * fields initialized from the LDB, where LDB processing
    * is off, you need to call sm_allget. You might use
    * this to make the active LDBs read-only for a certain
    * transaction. */

 sm_dd_able(0);
    ...
    if (sm_r_window("popup", 5, 24) == 0)
    {
      sm_allget(0);
      while (sm_input(IN_DATA) != EXIT)
      {
       ...
      }
      sm_close_window();
    }
```

See Also `sm_dd_able`, `sm_lstore`

sm_*amt_format

Writes formatted data to a field

```
int sm_amt_format(int field_number, char *buffer);
int sm_e_amt_format(char *field_name, int element, char *buffer);
int sm_i_amt_format(char *field_name, int occurrence,
    char *buffer);
int sm_n_amt_format(char *field_name, char *buffer);
int sm_o_amt_format(int field_number, int occurrence,
    char *buffer);
```

`field_name, field_number`
: The field to receive the formatted data.

`element`
: The onscreen element in the field.

`occurrence`
: The occurrence in the field.

`buffer`
: A pointer to the data to write.

Returns

0	Success.
-1	The field is not found or the occurrence is out of range.
-2	The edited string does not fit in the field.

Description `sm_amt_format` writes data to a field in the following steps:

1. Panther checks whether the field's format properties are set for numeric display. If so, it formats the data in `buffer` accordingly.
2. Panther calls `sm_putfield` to write the string to the specified field. If the field's `data_formatting` property is set to `PV_NONE`, `sm_putfield` writes the unedited string. If the resulting string is too long for the field, Panther truncates it.

Example

```
#include <smdefs.h>

    /* Write a list of real numbers, stored as character strings,
     * to the screen. The first and last fields in the list are
     * tagged with special names.
    */

int fld, first, last;
extern char *values[]; /* defined elsewhere */

last = sm_n_fldno("last");
first = sm_n_fldno("first");
for (fld = first; fld <= last; ++fld)
{
    sm_amt_format(fld, values[fld - first]);
}
```

See Also `sm_dtofield`, `sm_strip_amt_ptr`

sm_append_bundle_data

Sends data to a bundle item

```
int sm_append_bundle_data(char *bundle_name, int item_no,
    char *data);
```

`bundle_name`
: The name of the bundle to get data. Supply `NULL` or an empty string to specify the unnamed bundle.

`item_no`
: The bundle offset of the item to get `data`. You add data items to a bundle through successive calls to `sm_append_bundle_item`; each data item is identified by its offset in the bundle, where the first data item has an offset value of 1. If `item_no` already contains data, Panther appends `data` as the item's latest occurrence.

`data`
: A single occurrence of data to append to `item_no`.

Returns

0	Success.
-1	Invalid bundle name or item number.
-2	Memory allocation error.

Description

`sm_append_bundle_data` sends a single occurrence of data to the specified data item in `bundle_name`. A bundle contains sequentially numbered data items, where each data item can hold one or more occurrences of send data for later access by `sm_get_bundle_data`. If the source data contains multiple occurrences, Panther ends each occurrence with a null string terminator.

This function assumes the existence of the specified bundle and item. Before calling this function, create the target bundle and its items with calls to `sm_create_bundle` and `sm_append_bundle_item`, respectively.

Example

```
/* Iterate over all fields on current screen and
     * send data to bundle
     */
    void sendScreenDataToBundle(int numFields)
    {
    int ret, i, item;
    ret = sm_create_bundle("myBundle");
if (ret == 0)
   {
    sm_append_bundle_item("myBundle");
    item = sm_bundle_item_count("myBundle");
    for (i = 1; i <= numFields; i++)
       {
       sm_append_bundle_data("myBundle",
          item, sm_i_fptr("mySend", i));
       }
   }
return 0;
}
```

See Also `sm_append_bundle_item`

sm_append_bundle_done

Optimizes memory allocated for a send bundle

```
int sm_append_bundle_done(char *bundle_name);
```

`bundle_name`
: The name of the bundle. Supply `NULL` or empty string to specify the unnamed bundle.

Returns

0	Success.
-1	Invalid bundle name.

Description `sm_append_bundle_done` optimizes the memory allocated for a send bundle. Call this function after you finish appending items and data to a bundle.

See Also `sm_append_bundle_data`

sm_append_bundle_item

Adds a data item to a bundle

```
int sm_append_bundle_item(char *bundle_name);
```

`bundle_name`
: The name of the bundle to get a new item. Supply `NULL` or empty string to specify the unnamed bundle.

Returns

0	Success.
-1	Invalid bundle name.
-2	Memory allocation error.

Description `sm_append_bundle_item` appends a new data item to the end of the specified bundle. After you create a data item, you can send one or more occurrences of data to it by calling `sm_append_bundle_data`.

This function assumes the existence of `bundle_name`, previously created with `sm_create_bundle`. A bundle contains sequentially numbered data items, where the first data item has an offset of 1.

Example See the example in `sm_append_bundle_data`.

See Also `sm_append_bundle_data`

sm_*at_cur

Displays a window at the cursor location

```
int sm_d_at_cur(char *address);
int sm_l_at_cur(int lib_desc, char *name);
int sm_r_at_cur(char *name);
```

`address`
: The address of the screen in memory.

`lib_desc`
: Specifies the library in which the window is stored, where `lib_desc` is an integer returned by `sm_l_open`. You must call `sm_l_open` before you read any screens from a library.

`name`
: The name of the window.

Environment `sm_d_at_cur` is C only

Returns

0	Success.
-1	Screen file's format is incorrect.
-2	Screen cannot be found.
-3	System ran out of memory but the previous screen was restored.
-5	System ran out of memory after the screen was cleared.
-6	Library is corrupted.

Description `sm_window`.

sm_*attach_drawing_func

Associates a drawing function with a widget

```
#include <smmwuser.h>
int sm_mw_attach_drawing_func(int widgetnumber, int (*drawfunc));
int sm_mwn_attach_drawing_func(char *widgetname, int (*drawfunc));
int sm_mwe_attach_drawing_func(char *widgetname, int element,
    int (*drawfunc));

#include <smxmuser.h>
int sm_xm_attach_drawing_func(int widgetnumber, void(*drawfunc),
    XtPointer data);
int sm_xmn_attach_drawing_func(char *widgetname, void(*drawfunc),
    XtPointer data);
int sm_xme_attach_drawing_func(char *widgetname, int element,
    void(*drawfunc), XtPointer data);
```

`widgetname`, `widgetnumber`
: Specifies the widget to get `drawfunc`.

`element`
: If the widget is an array, specifies the element in `widgetname` to get `drawfunc`.

`drawfunc`
: The drawing function to attach to the specified widget. Drawing function declarations for Windows and Motif are shown in Description.

`data`
: Points to a user-defined structure that contains the data required by the drawing function.

Environment Motif, Windows

Returns

0	Success.
-1	Invalid widget or element, or the appropriate data structures or handles do not exist and cannot be created.

Description

`sm_attach_drawing_func` attaches the drawing function pointed to by `drawfunc` to the specified widget or element on the current screen. The widget must have its `customer_drawn` property set to `PV_YES`. You can use your own drawing functions with dynamic labels, push buttons, and toggle buttons. Panther handles all processing for these widgets except for drawing them, although it does draw the shading for push button widgets.

The most convenient place to attach a drawing function is at screen entry. Once attached, the drawing function is called whenever the widget needs to be painted, drawn or refreshed, regardless of whether the paint message comes from the window manager or from Panther.

Windows Draw Function Declaration

For Windows applications, declare the drawing function as follows:

```
int drawfunc(HWND handle, UINT message, WPARAM wParam,
    LPARAM lParam);
```

The `HWND` argument is a handle to the widget. If the widget is a dynamic label, the `message` argument is a `WM_PAINT` message. If the widget is a push button or toggle button, the `message` argument is a `WM_DRAWITEM` message. For dynamic labels, the `lParam` and `wParam` arguments are not used. For push buttons or toggle buttons, the `wParam` argument specifies the identifier of the widget that sent the message, and the `lParam` argument points to a `DRAWITEMSTRUCT` structure, which provides information on how to paint the widget.

Refer to the Windows SDK documentation for details on `WM_PAINT` and `WM_DRAWITEM` messages, and the `DRAWITEMSTRUCT` data structure.

Note: Because Panther draws the shading on a push button and toggle button, it alters a field in the `DRAWITEMSTRUCT` which specifies the rectangle to draw in. The rectangle passed to the `drawfunc` in the `rcItem` field is reduced slightly to account for the shading. The `drawfunc` therefore should draw in the entire rectangle that it is passed, and not draw any shading. Furthermore, the `hDC` item in the structure is altered, allowing for faster display and less flashing.

Panther selects Panther's color palette into the device context. For a dynamic label, the color palette is selected into the device context during the `BeginPaint()` call in the `drawfunc`. For a push button or toggle button, the palette is selected into the memory device context before `drawfunc` is called.

After `drawfunc` returns, Panther draws the cursor or focus rectangle. Panther ignores the return value from `drawfunc`.

Motif Draw Function Declaration

For Motif, declare the drawing function as follows:

```
void drawfunc(Widget wdgt, XtPointer xtpUserData, XtPointer
xtpCallBackData);
```

Example

```
#include <smdefs.h>
#include <smmwuser.h>

int MyDrawingFunc(HWND, UINT, WPARAM, LPARAM);

/* sample drawing function */
   int
   MyDrawingFunc(hWnd, message, wParam, lParam)
   HWND hWnd;
   UINT message;
   WPARAM wParam;
   LPARAM lParam;
   {
          PAINTSTRUCT ps;
          HBRUSH hBrush;

       BeginPaint(hWnd, &ps);

       hBrush = CreateSolidBrush(RGB(0, 0, 255));
          FillRect(ps.hdc, &ps.rcPaint, hBrush);
          DeleteObject(hBrush);

       EndPaint(hWnd, &ps);

       return(0);
   }

   {
          ...
          /* attach drawing function to widget number 2 */
          if (sm_attach_drawing_func(2, MyDrawingFunc) == -1)
          {
                 << error handling >>
          }
          ...
   }

int MyButtonDrawingFunc(HWND, UINT, WPARAM, LPARAM);

/* sample drawing function */
   int
   MyButtonDrawingFunc(hWnd, message, wParam, lParam)
   HWND hWnd;
   UINT message;
```

```
WPARAM wParam;
LPARAM lParam;
{
        DRAWITEMSTRUCT *dis;
        HBRUSH hBrush;

    dis = (DRAWITEMSTRUCT *)lParam;

    hBrush = CreateSolidBrush(RGB(0, 0, 255));
        FillRect(dis->hDC, &dis->rcItem, hBrush);
        DeleteObject(hBrush);
        return(0);
}

{
        ...
        /* attach drawing function to widget number 3 */
      if (sm_attach_drawing_func(3, MyButtonDrawingFunc) == -1)
        {
        << error handling >>
        }
        ...
}
```

sm_backtab

Backtabs to the previous unprotected field

```
void sm_backtab(void);
```

Description

`sm_backtab` moves the cursor to the first enterable position of the field with the next-lowest field number that is tab-accessible. The following conditions can modify this behavior:

- The cursor is not in the current field's first enterable position and the field is left-justified. In this case, `sm_backtab` moves the cursor to the current field's first enterable position.
- The cursor is in a field with a previous-field property and one of the fields specified by the property is accessible to tabbing. The cursor moves to the first enterable position of that field.
- The cursor is in the first position of the first unprotected field on the screen, or before the first unprotected field on the screen. The cursor wraps backward into the last unprotected field.
- There are no unprotected fields. The cursor remains stationary.

If the destination field is shiftable, it is reset according to its justification. The first enterable position depends on the justification of the field and, in fields with embedded punctuation, on the presence of punctuation.

This function does not immediately trigger field entry, exit, or validation processing. This processing occurs according to the cursor position when control returns to `sm_input`.

Panther calls this function when the Panther logical key BACK is struck.

See Also

`sm_home`, `sm_last`, `sm_nl`, `sm_tab`

sm_bel

Issues a beep from the terminal

```
void sm_bel(void);
```

Description

`sm_bel` causes the terminal to beep, usually by transmitting the ASCII `BEL` code to it. If there is a `BELL` entry in the video file, `sm_bel` transmits that instead. This usually causes the terminal to flash.

Even if there is no `BELL` entry, use this function instead of sending a `BEL`, because certain displays use `BEL` as a graphics character.

This function is automatically called when message text begins with `%B`.

Example

```
#include <smdefs.h>

   /* Beep if cost is too high. */
   if (sm_n_dblval("cost") > 1000.00)
       sm_bel();
```

sm_bi_compare

Compares widgets in the current table view with their before-image values

```
#include <tmusubs.h>
int sm_bi_compare(void);
```

Returns

`DM_TM_ERR_GENERAL` if no transaction or table view is available.

Success—one of the following constants:

- `BI_UNCHANGED`: Occurrence was not changed.
- `BI_DELETED`: Occurrence was deleted.
- `BI_INSERTED`: Occurrence was inserted.
- `BI_KEY_NULLED`: A primary key field in the occurrence was cleared or set to `NULL`.
- `BI_KEY_CHANGED`: A primary key field in the occurrence was changed to a non-`NULL`/non-empty value.
- `BI_UPDATED`: A non-primary key field in the occurrence was changed.

Description

`sm_bi_compare` compares an occurrence value with its before-image and returns a code indicating the status of the comparison. Comparison codes are listed above.

The occurrence is the current occurrence number as determined by `sm_tm_inquire("TM_OCC")`. A positive occurrence number indicates an onscreen occurrence. A negative occurrence number indicates a deleted occurrence; an occurrence is deleted by the logical key DELL or by a call to `sm_i_doccur`.

In the standard transaction models, the requests `TM_INSERT`, `TM_UPDATE`, and `TM_DELETE` each call `sm_bi_compare`. This allows the model to choose the appropriate processing for a changed occurrence.

A special case exists when a row's primary key value is set to empty or `NULL`. The program can do this in one of the following ways:

- Write an empty string to the field.
- Call `sm_tm_command("CLEAR")`.
- Call `sm_tm_clear`.

In the standard models both the `TM_DELETE` and `TM_INSERT` requests test for `BI_KEY_CHANGED` and both perform processing for this change. Therefore, if a primary key value changes, the standard models delete the occurrence using the before-image value of the primary key and insert a new occurrence using the onscreen value of the primary key. The model may be changed so that `TM_UPDATE` handles all updates, including primary key changes.

This function operates on the current table view. It is intended to be called from a transaction model or event function.

Example

```
/* The following example taken from the standard
       transaction model for JDB shows the processing for the
       TM_UPDATE request. */

case TM_UPDATE:
       /* Do nothing, except for updates */
    occ_type = sm_bi_compare();
    if (occ_type != BI_UPDATED)
    {
       break;
    }

    if (!reuse_cursor)
    {
       save_cursor_type = 0;
    }
    reuse_cursor = 0;

    sm_tm_push_model_event(TM_UPDATE_EXEC);
    sm_tm_push_model_event(TM_UPDATE_DECLARE);
    sm_tm_push_model_event(TM_GET_SAVE_CURSOR);
    break;
```

sm_bi_copy

Copies current values of a range of occurrences to before images

```
#include <tmusubs.h>
int sm_bi_copy(void);
```

Returns

- `0`: Success.
- `DM_TM_ERR_GENERAL`: No transaction or table view is available.
- `DM_TM_ERR_MALLOC`: Memory allocation error.

Description

`sm_bi_copy` writes the current values of a range of occurrences to their respective before-image occurrences. The starting occurrence is the value of `sm_tm_inquire("TM_OCC")` and the range of occurrences is determined by the value of `sm_tm_inquire("TM_OCC_COUNT")`. If `TM_OCC_COUNT` has a value of `-1`, `sm_bi_copy` gets the number of occurrences in the table view. If `TM_OCC` has a value of `1` and `TM_OCC_COUNT` has `-1`, `sm_bi_copy` copies every occurrence in the table view. Use `sm_tm_iset` to set the values of `TM_OCC` and `TM_OCC_COUNT` before calling `sm_bi_copy`.

The `SELECT` transaction command calls `sm_bi_copy` for updatable and non-updatable table views. It sets `TM_OCC` to the first occurrence where data was fetched; it sets `TM_OCC_COUNT` to the number of rows fetched. Therefore, `sm_bi_copy` copies each selected occurrence.

The standard transaction models call `sm_bi_copy` in the `TM_POST_SAVE` request if the current mode is `TM_UPDATE_MODE` and `sm_bi_initialize` was successful. Notice that the models set `TM_OCC_COUNT` to `-1` before calling `sm_bi_copy`. This ensures that all onscreen occurrences are copied.

sm_bi_initialize

Initializes before-image data for widgets in the current table view

```
#include <tmusubs.h>
int sm_bi_initialize(void);
```

Returns

- 0: Success. Before-image successfully initialized for the table view or the table view has the `updatable` property (under Transaction) set to `PV_NO`.
- `DM_TM_ERR_TBLNAME`: Table view did not have `table` property set.
- `DM_TM_ERR_PRIMARY_KEY`: Table view did not have a `primary_key` property set.
- `DM_TM_ERR_COL_NOT_FOUND`: Widget not found for primary key column.
- `DM_TM_ERR_MALLOC`: Memory allocation error.
- `DM_TM_ERR_GENERAL`: No transaction or table view is available.

Description

`sm_bi_initialize` initializes or reinitializes before-image data for the widgets in the current table view. Before-image describes the state of transaction data before the user or program changes it.

The transaction commands `NEW` and `SELECT` call `sm_bi_initialize`. For the `NEW` command, the before-image for the table view is empty. For the `SELECT` command, a before-image is defined for each row in the select set.

To initialize the before-image structures, the function first examines the properties of the current table view and the table view's members. It builds the table view's insert list and update list and it verifies that the current table view can participate in the before-image. If a table view has the `updatable` property set to `PV_YES`, it must also have values in the `table` and `primary_key` properties (under Database).

If the Table and Primary Key properties are not set, `sm_bi_initialize` returns an error. Furthermore, `sm_bi_initialize` verifies that a widget exists for each column named by the table view's `primary_key` property. If the widget does not exist in the current table view, the transaction manager looks for a link that names the current table view as a child. The criteria is satisfied if the primary key column is named in the `relations` property of the link and that property points to an onscreen widget, a literal, or to a widget in the link's parent table view (or the parent of the server view). Otherwise, `sm_bi_initialize` returns an error.

The standard transaction models call `sm_bi_initialize` as part of the processing for the `TM_POST_SAVE` request. If an application has saved data while in new or update mode, the models call `sm_bi_initialize` after the save completes. This allows the application to use the current screen data as the starting point for the next save.

For example, assume the application executes `sm_tm_command("NEW")` to enter new customer data. The user enters the data and the application executes `sm_tm_command("SAVE")`. If the save is successful (e.g., it generates and executes a SQL `INSERT` statement), the standard model calls `sm_bi_initialize` before returning control to Panther. To enter the customer's spouse, the user can change the appropriate fields and call `sm_tm_command("SAVE")` again. This is also equivalent to calling `sm_tm_command("COPY")` after a `SAVE`.

Similarly, for the `SELECT` command, the use of `sm_bi_initialize` in the standard models allows the application to continue updating the screen data after a save. If customer data is fetched with `sm_tm_command("SELECT")` and the user changes the customer's phone number and calls `sm_tm_command("SAVE")`, the model performs save processing (e.g., generates and executes a `SQL UPDATE` statement) and, by default, calls `sm_bi_initialize`. The user can continue updating the onscreen data without re-selecting it. If the user enters a comment and calls `sm_tm_command("SAVE")` again, the transaction manager performs save processing for all changes since the last call to `sm_bi_initialize`. Therefore, it might generate and execute a `SQL UPDATE` statement for the comment; it does not repeat save processing for the earlier phone number change.

This function operates on the current table view. It is intended to be called from a transaction model or event function.

sm_bkrect

Sets the background color of a rectangle

```
int sm_bkrect(int start_line, int start_col, int num_of_lines,
    int num_of_col, int bkgr_colors);
```

`start_line`, `start_col`
: Specify the upper-left corner of the area to set, where the values of `start_line` and `start_column` can range from 0 through the length and width of the screen less 1, respectively.

`num_of_lines`
: The length of the area to set.

`num_of_col`
: The width of the area to set.

`bkgr_colors`
: The attributes to set as the area's background color.

Environment Character-mode

Returns

0	Success.
1	The starting line and column are valid but the rectangle was truncated to fit.
-1	Invalid starting line or column.

Description `sm_bkrect` changes the background color of a rectangular area of the current screen. The background color must be one of the constants defined in `smattrib.h`. You can highlight the background color by OR'ing the background color attribute with `B_HILIGHT`.

All fields or elements that start inside the area have their background attributes changed to the specified attribute. Display text inside the rectangular area has its background attribute set. Make sure that fields or elements that change are entirely inside the area; otherwise, a ragged edge results.

Example

```
/*     Draw some colored squares on the display*/
    int colors[] =
    {
        B_RED,
        B_BLUE,
        B_WHITE,
        B_CYAN
    };

int mondrian(void)
    {
      int  i;
        for (i=0;i<sizeof(colors)/sizeof(int);i++)
      {
          sm_bkrect((i/2) * 10,(i & 1) * 40, 10, 40, colors[i]);
      }
      return(0);
    }
```

sm_c_off

Turns the cursor off

```
void sm_c_off(void);
```

Description

`sm_c_off` tells Panther that the normal cursor setting is off. Use this function when all fields on the current screen are protected. The normal cursor setting is in effect except under these circumstances:

- The cursor is off when a block cursor is in use, as during menu processing.
- The cursor is off while screen manager functions are writing to the display.
- The cursor is on within certain error message display functions.

If the display cannot turn its cursor on and off—`CON` and `COF` entries are not defined in the video file—this function has no effect.

Use `sm_c_on` to turn the cursor on.

Example

```
sm_ferr_reset(0, "Verify that the cursor is turned ON");
    sm_c_off();
    sm_femsg(0, "Verify that the cursor is turned OFF");
    sm_c_on();
    sm_femsg(0, "Verify that the cursor is turned ON");
```

See Also

`sm_c_on`

sm_c_on

Turns the cursor on

```
void sm_c_on(void);
```

Description

`sm_c_on` tells Panther that the normal cursor setting is on. The normal setting is in effect except under these circumstances:

- The cursor is off when a block cursor is in use, as during menu processing.
- The cursor is off while screen manager functions are writing to the display.
- The cursor is on within certain error message display functions.

If the display cannot turn its cursor on and off—`CON` and `COF` entries are not defined in the video file—this function has no effect.

Use `sm_c_off` to turn the cursor off.

Example

```
sm_ferr_reset(0, "Verify that the cursor is turned ON");
   sm_c_off();
   sm_femsg(0, "Verify that the cursor is turned OFF");
   sm_c_on();
   sm_femsg(0, "Verify that the cursor is turned ON");
```

See Also

`sm_c_off`

sm_c_vis

Turns the cursor position display on or off

```
void sm_c_vis(int display);
```

`display`

Specifies whether to turn the cursor position display on or off:

- 0 causes subsequent status line messages to be displayed without the cursor's position display.
- Non-zero displays subsequent status line messages with the cursor's position display. This includes background status messages. Messages that would overlap the cursor position display are truncated.

Description

`sm_c_vis` toggles display of the cursor position on and off according to the value of `display`. This function has no effect if the `CURPOS` entry in the video file is not defined. In this case, the cursor position display never appears.

Panther uses an asynchronous function and a status line function to perform the cursor position display. If either one is already installed, `sm_c_vis` overrides it.

Example

```
#include <smdefs.h>
   #include <smkeys.h>

   /* Toggle the cursor position display on or off when
    * the PF10 key is struck. The first time the key is
    * struck, it will go on.
    */

   static int cpos_on = 0;

   switch (sm_input(IN_DATA))
   {
   ...
   case PF10:
      sm_c_vis (cpos_on ^= 1);
   ...
   }
```

sm_calc

Executes a math expression

```
int sm_calc(int field_number, int occurrence, char *expression);
```

`field_number`
: The field to use for relative field references, for backward compatibility only. If `expression` references fields according to current conventions, supply 0.

`occurrence`
: The occurrence in `field_number` to use for relative field references, for backward compatibility only. If `expression` references fields according to current conventions, supply 0.

`expression`
: A math expression. Refer to “Performing Calculations and Validating Numbers” on page 8-26 in *Using the Editors* for information on creating math expressions.

Returns

0	Success
-1	A math error occurred.

Description

`sm_calc` lets you execute a math expression. Use this function to perform mathematical operations that use the contents of one or more fields and then insert the result into a field.

If, in the event of a math error, you want the cursor to move a specific field, specify that field with `field_number`. If the field is an array and `occurrence` is offscreen, Panther scrolls that occurrence into view.

Example

```
/* Compute payment due date. */

    sm_calc(0, 0, "paymentduedate = @date(shipdate) + 30");
```

sm_cancel

Resets the display and exits

```
void sm_cancel(int arg);
```

`arg`
: A dummy argument that always has a value of 0. This argument lets the C function `signal` use `sm_cancel` as a signal handler.

Description

`sm_initcrt` installs this function to handle keyboard interrupts. `sm_cancel` calls `sm_resetcrt` to restore the display to the operating system's default state, and exits to the operating system.

Depending on your operating system, you can also install this function to handle conditions that normally cause a program to abort. If a program aborts with `sm_cancel` installed, its call to `sm_resetcrt` ensures that your terminal is restored to its normal state.

Example

```
/* the following program segment could be found in
    * some error routines */

   #include <smdefs.h>
   if (error)
   {
       sm_fquiet_err(0, "fatal error -- can't continue!\n");
       sm_cancel(0);
   }

   /* The following code can be used on a UNIX system to
    * install sm_cancel() as a signal handler. */

   #include <smdefs.h>
   #include <signal.h>

   signal(SIGTERM, sm_cancel);
```

sm_ckdigit

Validates data with a check digit function

```
int sm_ckdigit(int field_number, char *field_data, int occurrence,
    int modulus, int minimum_digits);
```

`field_number`
: The field to validate. If `field_number` is 0, `sm_ckdigit` uses the data in `field_data`. If an error occurs and `field_number` is 0, no message is posted.

`field_data`
: Specifies the data to validate. If `field_data` is null, the string to check is obtained from the `field_number` and `occurrence` and an error message is displayed if the string is bad.

`occurrence`
: The occurrence in `field_number` to validate.

`modulus`
: Specifies the check digit algorithm to use. By default, `sm_ckdigit` supports mod 10 and mod 11 algorithms. For more information about the check digit algorithms, refer to the source code of `sm_ckdigit` that is distributed with Panther.

`minimum_digits`
: The minimum number of digits required by the check digit algorithm.

Returns

0	The value of `field_number` or `field_data` is valid.
-1	The field contents lack the minimum number of digits or proper check digit.
-2	`field_data` is null and the field or occurrence cannot be found.

Description

`sm_ckdigit` checks whether the data in `field_data` or `occurrence` contains the required minimum number of digits and ends with the proper check digit. This function is typically called by Panther at field validation; it uses the values in the field's Check Digit and Minimum Digits properties as arguments for parameters `modulus` and `minimum_digits`, respectively.

If you specify a field occurrence and its data is invalid, Panther issues an error message before returning. If you set `field_number` to 0 and supply invalid data for `field_data`, Panther does not issue any message.

You can install your own check digit function to replace `sm_ckdigit`. For more information on installing functions, refer to “Installing Functions” on page 44-5 in the *Application Development Guide*.

sm_cl_all_mdts

Clears the mdt property for all occurrences

```
void sm_cl_all_mdts(void);
```

Description

`sm_cl_all_mdts` resets to `PV_NO` the `mdt` property of all occurrences, onscreen and off, for every field on the current screen. This property indicates whether the data in an occurrence has changed since screen entry.

Panther sets an occurrence's `mdt` property to `PV_YES` when it is modified after screen entry, either because of keyboard entry or a call to a function like `sm_putfield`. A field undergoes validation only if its `mdt` property is set to `PV_YES`.

You can clear an individual occurrence, set its `mdt` property to `PV_NO`.

Example

```
 /* Clear mdt property for all fields on the screen.
     * Then write data to the last field, and check that its
     * mdt property is the only one set. */

    vars ct
    vars not_modified = 1
    vars total_fields = @screen("@current")->numflds

call sm_cl_all_mdts()
    @field_num(total_fields) = "Hello" // modify last field

for ct = 1 while ct <= total_fields && not_modified
    {
        if @field_num(ct)->mdt == PV_YES
            not_modified = 0
    }
    ct = ct - 1 // remove last increment of counter

    if ct == total_fields && !not_modified
       msg emsg("last field is the only one modified")
    else if ct < total_fields
       msg emsg("Something is rotten here")
    else
       msg emsg("Nothing has changed")
    return
```

See Also

`sm_tst_all_mdts`

sm_cl_unprot

Clears data from unprotected widgets

```
void sm_cl_unprot(void);
```

Description

`sm_cl_unprot` erases onscreen and offscreen data from all widgets (with the exception of list boxes) that are unprotected from clearing—that is, their `clearing_protect` property is set to `PV_NO`. Date and time fields that take system values are reinitialized. Fields whose `null_field` property are set to `PV_YES` are reset to their null indicator values.

This function is normally bound to the CLR key.

To clear data from list box widgets, refer to `sm_clear_array`.

Example

```
/* The following code clears all unprotected fields
     * and puts the cursor into the first one. */

    sm_cl_unprot();
    sm_home();
```

sm_*clear_array

Clears all data in an array

```
int sm_clear_array(int field_number);
int sm_n_clear_array(char *field_name);
int sm_1clear_array(int field_number);
int sm_n_1clear_array(char *field_name);
```

`field_name`, `field_number`
    A field in the array to clear.

Returns

0	Success.
-1	The field does not exist.

Description

`sm_clear_array` clears all data from the array that contains `field_number` or `field_name` and resets the number of occurrences in the array to 0. The array is cleared even if it is protected from clearing.

`sm_1clear_array` and `sm_n_1clear_array` only clear the specified array; `sm_clear_array` and `sm_n_clear_array` also clear arrays synchronized with the array unless they are protected from clearing.

Example

```
/* Clear the entire array of "names" and arrays
 * synchronized with "names". */
   sm_n_clear_array("names");

   /* Clear the "totals" column of a screen,
    * without clearing arrays synchronized with "totals". */
   sm_n_1clear_array("totals");
```

sm_close_window

Closes the current window

```
int sm_close_window(void);
```

Returns

0	Success.
-1	No window is open.

Description

`sm_close_window` closes a screen opened as a window by `sm_r_window`, `sm_r_at_cur`, or one of their variants.

`sm_close_window` erases the currently open window and restores the screen to its state before the window opened. If LDB processing is active, `sm_lstore` writes data from the named fields to the LDB; otherwise, all window data is lost. If the closed window was spawned by another one, Panther makes the parent window the current one and restores the cursor to its last position in that window.

Panther automatically calls `sm_close_window` when you close a form with `sm_jclose`. `sm_jclose` calls `sm_jform` to pop the form stack and calls `sm_close_window` to empty the form's window stack.

Note: `sm_close_window` does not close the base screen in a window stack—that is, the active form. To close the active form, call `sm_jclose`.

Example

```
#include <smdefs.h>
    #include <smkeys.h>

    /* In a validation function, if the field contains a */
    /* special value, open up a window to prompt for a */
    /* second value and save it in another field. */

    int validate (field, data, occur, bits)
    char *data;
    int field, occur, bits;
    {
        char buf[256];

        if (bits & VALIDED)
            return 0;

        if (strcmp(data, "other") == 0)
        {
            sm_r_at_cur "getsecval");
            if (sm_input(IN_DATA) != EXIT)
                sm_getfield(buf, 1);
            else
                buf[0] = 0;
            sm_close_window();
            sm_n_putfield("secval", buf);
        }

        return 0;
    }
```

See Also `sm_r_window`, `sm_wselect`

sm_com_load_picture

Returns the object ID for a graphics file

```
#include <smmwuser.h>
int sm_com_load_picture(char *name, int width, int height);
```

`name`
: The name of the graphics file located in a Panther library or in a directory specified by `SMPATH`.

`width`, `height`
: The size of the graphic. If 0, the picture keeps it natural size; otherwise, these parameters can be used to shrink or enlarge the picture. In JPL these parameters can be omitted and therefore default to 0.

Environment Windows

Scope Client

Returns

- An object ID which represents the picture. The caller is responsible for destroying the picture (by calling `sm_obj_delete_id`) when the picture is no longer needed.

Description `sm_com_load_picture` gets an object ID for the specified picture so that the image can be passed as a parameter in `sm_obj_call` or as a value in `sm_obj_set_property`.

In those functions, the image's object ID can be referenced using `@obj`.

Example In the following example, `@obj` must be used since the `ImageListcontrol` does not supply sufficient information in its type library. In other cases, `@obj` may not be needed (but is not harmful). If you get a `type mismatch` error without using `@obj`, try `@obj` in the call.

```
proc fill_imagelist
{
vars imagelist// imagelist control
vars images// list of images in the imagelist
vars pic   // one picture

@app()->current_component_system=PV_SERVER_COM
imagelist = sm_obj_create("MSComctlLib.ImageListCtrl")
images = sm_obj_get_property(imagelist, "ListImages")

pic = sm_com_load_picture("logo.bmp")
call sm_obj_call(images, 1, '', @obj(pic))
sm_obj_delete_id(pic)

pic = sm_com_load_picture("folder.bmp")
call sm_obj_call(images, 2, '', @obj(pic))
sm_obj_delete_id(pic)

pic = sm_com_load_picture("screen.bmp")
call sm_obj_call(images, 3, '', @obj(pic))
sm_obj_delete_id(pic)

call sm_obj_delete_id(images)

    // install the ImageList into the control

call sm_obj_set_property(control->id, "ImageList", imagelist)
call sm_obj_delete_id(imagelist)
}
```

See Also `sm_obj_call`, `sm_obj_set_property`

sm_com_QueryInterface

Accesses an interface of a COM component

```
#include <smmwuser.h>
hr = sm_com_QueryInterface(obj_id, iid, ppv);

HRESULT hr;

REFIID iid;

LPVOID *ppv;
```

`obj_id`

An integer handle that identifies the COM object whose interface you want to get. The handle is returned by `sm_obj_create` for service components, `sm_prop_id` for ActiveX controls.

Environment Windows, Web; C only

Returns

- 0: The `HRESULT` is `S_OK`; the last call succeeded.
- `E_NOINTERFACE` if the interface is not available.

Description `sm_com_QueryInterface` can be used to access an interface of a COM component. This function provides low-level access to the component and, as such, can only be called from C or C++.

For more information on using the QueryInterface method, refer to the ActiveX and COM specifications.

Example

```
int id = sm_prop_id ("treeview");
    LPDISPATCH pDispatch;
    HRESULT hr;

hr = sm_com_QueryInterface
       (id, IID_IDispatch, (LPVOID *)&pDispatch);

if (SUCCEEDED(hr))
    {
    ...
    pDispatch->Release ();
    }
```

sm_com_result

Gets the error code returned by the last call to a COM component

```
int sm_com_result(void);
```

Environment: Windows, Web

Scope: Client

Returns: On Windows platforms:

- The `HRESULT` from the most recent COM function call. Refer to `winerror.h` for values; 0 is the value for `S_OK`.

On UNIX platforms:

0	Success.
-1	Failure.

Description: `sm_com_result` returns the result from the last call to an `sm_com` routine. Only the result of the last call is recorded; subsequent calls will overwrite this value.

The `HRESULT` values are only available on Windows platforms and are determined by the OLE specifications and the COM component author. For values, refer to `winerror.h` and the documentation for the COM component.

Since an error handler will only be fired on negative exception codes, use this function to retrieve positive exceptions.

Example:

```
#include <smuprapi.h>
    {
    id = sm_prop_id("spinner");
    sm_obj_set_property(id, "Value", "40");
    errcode = sm_com_result();
    }
```

sm_com_result_msg

Gets the error message returned by the last call to a COM component

```
char *sm_com_result_msg(void);
```

Environment	Windows, Web
Scope	Client
Returns	The error message as a string.
Description	`sm_com_result_msg` returns a string giving the text of the error message.
See Also	`sm_com_result`

sm_com_set_handler

Sets an event handler for the specified event on an ActiveX control

```
int sm_com_set_handler(int obj_id, char *event, char *handler);
```

`obj_id`
: An integer handle that identifies the COM component whose method you want to call. The handle is returned through `sm_prop_id` for ActiveX controls.

`event`
: The designated event fired by the ActiveX control.

`handler`
: The handler to set for the specified event. This can be a prototyped function or a JPL procedure.

Environment Windows, Web

Scope Client

Returns

0	Success.
-1	Failure: The event is not supported by the component.

Description `sm_com_set_handler` sets the handler for the specified event. Refer to the documentation for the ActiveX control to see which events are available.

The ActiveX control can pass parameters as part of the event. If parameters exist, the handler must perform the necessary processing. For an example, open the TreeView ActiveX control in the *Panther COM Samples* and look at the `node_click` procedure.

The return value from the handler is ignored.

Note: COM components (as opposed to ActiveX controls) normally do not generate events. However, this routine can be used for any COM components that do fire events.

Example

```
// This C function calls the onURL handler on the
   // URLSelected event.

#include <smuprapi.h>
   int id;
```

```
    int retcode;

{
    id = sm_prop_id("treeview");
    retcode = sm_com_set_handler(id, "URLSelected", "onURL");
    }

// This is the same JPL procedure.

vars retcode

call sm_com_set_handler(treeview->id, "URLSelected", "onURL")

proc onURL
...
return
```

sm_*copyarray

Copies the contents of one array to another

```
int sm_copyarray(int target_fnum, int source_fld);
int sm_n_copyarray(char *target_fname, char *source_name);
```

`target_fnum`, `target_fname`
An element in the array to receive the data.

`source_fld`, `source_name`
An element in the source array.

Returns

0	Success.
-1	One of the fields or LDB entries is not found.

Description

`sm_copyarray` and `sm_n_copyarray` copy the contents of the specified source array into a target array. For each destination array occurrence, the `mdt` property is set to `PV_YES` and the `valided` property to `PV_NO` to indicate that the occurrence is modified and needs validation.

Because `sm_copyarray` references fields by number, they must be on the current screen. `sm_n_copyarray` looks for the named fields first in the current screen; if the screen omits one or both of the specified arrays, the function looks for the named entry in the current LDB. If found there, `sm_n_copyarray` gets the data from or writes to that entry.

Source and target arrays must be compatible to ensure the integrity of the copied data. Otherwise, Panther handles differences between the two arrays as follows:

- If the source data is too long for its target, Panther truncates it automatically and issues no warning.
- If the data is too short, Panther pads the target occurrence with spaces.
- If the target array has fewer occurrences than the source array, Panther discards the data in the extra occurrences.
- If the target array has more occurrences than the source array, Panther clears the data from the extra target occurrences but maintains their allocation.

See Also `sm_clear_array`, `sm_getfield`, `sm_putfield`

sm_create_bundle

Creates a send bundle

```
int sm_create_bundle(char *bundle_name);
```

`bundle_name`

The name of the buffer, or *`bundle`*, in which to store the send data. Bundle names can be up to 31 characters long. You can create up to ten bundles of send data in memory. One of these bundles can be unnamed. JPL's `send` and `receive` commands identify the unnamed bundle as the default bundle. Create an unnamed bundle by supplying a null argument.

Returns

0	Success.
-2	Memory allocation failure.

Description

`sm_create_bundle` creates a new send bundle. The bundle initially is empty. After you create a bundle, you can append data items to it and send data to those items through `sm_append_bundle_item` and `sm_append_bundle_data`, respectively.

If an existing bundle is already named `bundle_name`, Panther frees the existing bundle and replaces it with the new one. If ten bundles already are in memory, Panther removes the oldest bundle.

sm_d_msg_line

Displays a message on the status line

```
void sm_d_msg_line(char *message, int display_attr);
```

`message`
: A pointer to the message to display. To clear the message previously displayed with this function, supply an empty string.

`display_attr`
: The display attribute to use for `message`, one of the constants defined in `smattrib.h`. A value of 0 clears the message previously displayed with this function.

  Foreground colors can be used alone or OR'd together with one or more highlights, a background color, and a background highlight. If you do not specify a highlight or a background color, the attribute defaults to white against a black background. Omitting a foreground color causes the attribute to default to black.

Description

`sm_d_msg_line` displays the contents of `message` on the status line with an initial display attribute of `display_attr`. If the cursor position display is turned on (refer to `sm_c_vis`), the end of the status line contains the cursor's current row and column.

Messages displayed with `sm_d_msg_line` override both background and field status text. They remain on all screens until you clear the status line with another call to `sm_d_msg_line`, where `message` gets an empty string and `display_attr` gets 0. Once cleared, the previously overridden message redisplays. The function `sm_d_msg_line` is itself overridden by `sm_ferr_reset` and related functions, or by the ready/wait message enabled by `sm_setstatus`.

Several percent escapes let you control the content and presentation of status messages. The character that follows the percent sign must be in uppercase. Note that if a message containing percent escapes is displayed before `sm_initcrt` is called, the percent escapes appear in the message.

If a string of the form `%A`*nnnn* appears anywhere in the message, the hexadecimal number *nnnn* is interpreted as a display attribute to be applied to the remainder of the message. Use numeric values to specify the logical display attributes you need to construct embedded attributes. These values are specified in Table 5-7:

Table 5-7 Panther color/attribute mnemonics

Foreground Attributes*		**Background Attributes**	
Attribute Mnemonic	**Hex Code**	**Attribute Mnemonic**	**Hex Code**
`REVERSE`	0010	`B_HILIGHT`	8000
`UNDERLN`	0020		
`BLINK`	0040		
`HILIGHT`	0080		
`DIM`	1000		
Foreground Colors		**Background Colors**	
`BLACK`	0000	`B_BLACK`	0000
`BLUE`	0001	`B_BLUE`	0100
`GREEN`	0002	`B_GREEN`	0200
`CYAN`	0003	`B_CYAN`	0300
`RED`	0004	`B_RED`	0400
`MAGENTA`	0005	`B_MAGENTA`	0500
`YELLOW`	0006	`B_YELLOW`	0600
`WHITE`	0007	`B_WHITE`	0700
`NORMAL_ATTR`	0007	`B_CONTAINER`	4000

** Attributes are additive. One or more foreground attributes can be added to a background attribute, foreground color and background color.*

If you want a digit to appear immediately after the attribute change, pad the attribute to 4 digits with leading zeros. If the following character is not a legal hex digit, then leading zeros are unnecessary.

If a string of the form `%K`*`keyname`* appears anywhere in the message, *`keyname`* is interpreted as a logical key constant, and the whole expression is replaced with the key label string defined for that key in the key translation file. If there is no label, the `%K` is stripped out and the constant remains. Key constants are defined in `smkeys.h`.

If the message begins with a `%B`, Panther beeps the terminal (using `sm_bel`) before issuing the message.

Example

```
/* The following prompt uses labels for the EXIT and
     * return keys, and underlines crucial words. */

   sm_d_msg_line("Press %KEXIT to %A0027abort%A7, "
      "or %KNL to %A0027continue%A7.");

   /* To clear the status line, use: */

   sm_d_msg_line("", 0);
```

See Also `sm_ferr_reset`, `sm_msg`

sm_*dblval

Returns the value of a field as a double precision floating point

```
double sm_dblval(int field_number);
double sm_e_dblval(char *field_name, int element);
double sm_i_dblval(char *field_name, int occurrence);
double sm_n_dblval(char *field_name);
double sm_o_dblval(int field_number, int occurrence);
```

`field_name`, `field_number`
The field with the value to get.

`element`
The element in `field_name` with the value to get.

`occurrence`
The occurrence with the value to get.

Environment C only

Returns

- The real value of the field.

Description `sm_dblval` returns the contents of the specified field as a double precision floating point. It calls `sm_strip_amt_ptr` to remove extra amount editing characters before it converts the data.

Example

```
#include <smdefs.h>

    /* Retrieve the value of a starting parameter. */

    double param1;

    param1 = sm_n_dblval("param1");
```

See Also `sm_dtofield`, `sm_strip_amt_ptr`

sm_dd_able

Turns LDB write-through on or off for all LDBs

```
int sm_dd_able(int flag);
```

`flag`

Specifies whether to turn LDB processing on or off:

0	Turn processing off; no data is exchanged between screens and LDBs.
1	Turn processing on for all LDBs loaded into memory.

The previous state of LDB write-through:

0	LDB write-through was off for all LDBs.
1	LDB write-through was on for one or more LDBs.

Description

`sm_dd_able` enables or disables data exchange between screens and all loaded LDBs according to the value of `flag`.

Individual LDBs can have their write-through capability selectively turned on or off via `sm_ldb_state_set`, but attempting activate write-through for an LDB will not work if `sm_dd_able` has already been called to turn processing off for all loaded LDBs.

For more information about LDB processing, refer to “Using Local Data Blocks” on page 25-7 in *Application Development Guide*.

See Also

`sm_ldb_state_set`

sm_dde_client_connect_cold

Creates a cold DDE link to a server

```
#include <smmwuser.h>
int sm_dde_client_connect_cold(char *server, char *topic,
   char *item, char *field);
```

`server`
: The server application's name—for example, `WINWORD`

`topic`
: The server topic, typically the file name of the spreadsheet or document—for example, `SALES.DOC`

`item`
: The server item—for example, `DDE_LINK1`

`field`
: The name of the widget to receive server data.

Environment Windows

Returns

1 Success.
0 Failure.

Description `sm_dde_client_connect_cold` creates a cold DDE link between a widget and a server application. Given a cold link, the server does not notify the client Panther application of changes to linked data. The application must explicitly request data updates by calling `sm_dde_client_request`.

Before creating a link, Panther must be enabled as a client. Panther checks whether a connection to the server application already exists—for example, another open screen has a link to this server. If no connection exists, Panther attempts to establish one. After Panther verifies or establishes a connection, it creates a cold link between the widget and the specified topic and item.

This function can succeed only if the server application is already running; otherwise Panther posts an error message. If the link cannot be created, Panther posts an error message.

See Also `sm_dde_client_request`

sm_dde_client_connect_hot

Creates a hot DDE link to a server

```
#include <smmwuser.h>
int sm_dde_client_connect_hot(char *server, char *topic,
   char *item, char *field);
```

`server`
: The server application's name—for example, `WINWORD`

`topic`
: The server topic, typically the file name of the spreadsheet or document—for example, `SALES.DOC`

`item`
: The server item—for example, `DDE_LINK1`

`field`
: The name of the widget to receive server data.

Environment Windows

Returns

1	Success.
0	Failure.

Description

`sm_dde_client_connect_hot` creates a hot DDE link between a widget and a server application. Given a hot link, the server automatically updates the widget whenever the linked data changes.

Before creating a link, Panther must be enabled as a client. Panther checks whether a connection to the server application already exists—for example, another open screen has a link to this server. If no connection exists, Panther attempts to establish one. After Panther verifies or establishes a connection, it creates a hot link between the widget and the specified topic and item.

This function can succeed only if the server application is already running; otherwise Panther posts an error message. If the link cannot be created, Panther posts an error message.

sm_dde_client_connect_warm

Creates a warm DDE link to a server

```
#include <smmwuser.h>
int sm_dde_client_connect_warm(char *server, char *topic,
   char *item, char *field);
```

`server`
: The server application's name—for example, `WINWORD`

`topic`
: The server topic, typically the file name of the spreadsheet or document—for example, `SALES.DOC`

`item`
: The server item—for example, `DDE_LINK1`

`field`
: The name of the widget to receive server data.

Environment Windows

Returns

1	Success.
0	Failure.

Description `sm_dde_client_connect_warm` creates a warm DDE link between a widget and a server application. Given a warm link, the server notifies the client Panther application of changes to linked data. However, the application must explicitly request data updates by calling `sm_dde_client_request`.

When the server notifies Panther that linked data has changed, Panther checks whether a callback function is installed and uses it to notify the application; otherwise, it uses its own callback function. Use `sm_dde_install_notify` to install a callback function.

Before creating a link, Panther must be enabled as a client. Panther checks whether a connection to the server application already exists—for example, another open screen has a link to this server. If no connection exists, Panther attempts to establish one. After Panther verifies or establishes a connection, it creates a warm link between the widget and the specified topic and item.

This function can succeed only if the server application is already running; otherwise Panther posts an error message. If the link cannot be created, Panther posts an error message.

See Also `sm_dde_client_request`, `sm_dde_install_notify`

sm_dde_client_disconnect

Destroys a DDE link to a server

```
#include <smmwuser.h>
int sm_dde_client_disconnect(char *server, char *topic,
   char *item, char *field);
```

`server`
: The server application's name—for example, `WINWORD`

`topic`
: The server topic, typically the file name of the spreadsheet or document—for example, `SALES.DOC`

`item`
: The server item—for example, `DDE_LINK1`

`field`
: The name of a widget on the active screen.

Environment Windows

Returns

- 1 Success.
- 0 Failure.

Description `sm_dde_client_disconnect` destroys a client DDE link on the active screen. If the link specified is the last link to a server application, Panther also closes the connection to that server.

Note: When a screen closes, Panther automatically destroys its DDE links.

sm_dde_client_off

Disables DDE client activity

```
#include <smmwuser.h>
void sm_dde_client_off(void);
```

Environment Windows

Description `sm_dde_client_off` prevents Panther from acting as a DDE client.

See Also `sm_dde_client_on`

sm_dde_client_on

Enables DDE client activity

```
#include <smmwuser.h>
void sm_dde_client_on(void);
```

Environment Windows

Description `sm_dde_client_on` lets Panther act as a DDE client.

See Also `sm_dde_client_off`

sm_dde_client_paste_link_cold

Creates a cold DDE paste link between a widget and a DDE server

```
#include <smmwuser.h>
int sm_dde_client_paste_link_cold(char *field);
```

`field`

The name of the widget to receive server data.

Environment Windows

Returns

1	Success.
≤0	Failure.

Description `sm_dde_client_paste_link_cold` requests a cold DDE paste link between a widget and a server application. Panther gets the clipboard data and its source—server, topic, and item. Subsequent requests to update data use this source information to get new data from the server. Given a cold paste link, the server does not notify the client Panther application of changes to linked data. The application must explicitly request data updates by calling `sm_dde_client_request`.

Before creating a paste link, two conditions must be true:

- The clipboard must contain data copied from the server.
- Panther must be enabled as a client.

Panther checks whether a connection to the server application already exists—for example, another open screen has a link to this server. If no connection exists, Panther attempts to establish one. After Panther verifies or establishes a connection, it creates a cold link between the widget and the data source.

This function can succeed only if the server application is already running; otherwise Panther posts an error message. If the link cannot be created, Panther posts an error message.

sm_dde_client_paste_link_hot

Creates a hot DDE paste link between a widget and a DDE server

```
#include <smmwuser.h>
int sm_dde_client_paste_link_hot(char *field);
```

`field`
: The name of the widget to receive server data.

Environment Windows

Returns

1	Success.
≤0	Failure.

Description `sm_dde_client_paste_link_hot` requests a hot DDE paste link between a widget and a server application. Panther gets the clipboard data and its source—server, topic, and item. Subsequent requests to update data use this source information to get new data from the server. Given a hot paste link, the server automatically updates the widget whenever the linked data changes.

Before creating a paste link, two conditions must be true:

- The clipboard must contain data copied from the server.
- Panther must be enabled as a client.

Panther checks whether a connection to the server application already exists—for example, another open screen has a link to this server. If no connection exists, Panther attempts to establish one. After Panther verifies or establishes a connection, it creates a hot link between the widget and the data source.

This function can succeed only if the server application is already running; otherwise Panther posts an error message. If the link cannot be created, Panther posts an error message.

sm_dde_client_paste_link_warm

Creates a warm DDE paste link between a widget and a DDE server

```
#include <smmwuser.h>
int sm_dde_client_paste_link_warm(char *field);
```

`field`
: The name of the widget to receive server data.

Environment Windows

Returns

1	Success.
≤0	Failure.

Description

`sm_dde_client_paste_link_warm` requests a warm DDE paste link between a widget and a server application. Panther gets the clipboard data and its source—server, topic, and item. Subsequent requests to update data use this source information to get new data from the server. Given a warm paste link, the server notifies the client Panther application of changes to linked data. However, the application must explicitly request data updates by calling `sm_dde_client_request`.

When the server notifies Panther that linked data has changed, Panther checks whether a callback function is installed and uses it to notify the application; otherwise, it uses its own callback function. Use `sm_dde_install_notify` to install a callback function.

Before creating a paste link, two conditions must be true:

- The clipboard must contain data copied from the server.
- Panther must be enabled as a client.

Panther checks whether a connection to the server application already exists—for example, another open screen has a link to this server. If no connection exists, Panther attempts to establish one. After Panther verifies or establishes a connection, it creates a warm link between the widget and the data source.

This function can succeed only if the server application is already running; otherwise Panther posts an error message. If the link cannot be created, Panther posts an error message.

sm_dde_client_request

Requests data from a DDE server

```
#include <smmwuser.h>
int sm_dde_client_request(char *server, char *topic, char *item,
   char *field);
```

`server`
: The server application's name—for example, `WINWORD`

`topic`
: The server topic, typically the file name of the spreadsheet or document—for example, `SALES.DOC`

`item`
: The server item—for example, `DDE_LINK1`

`field`
: The name of a widget on the active screen.

Environment Windows

Returns

- 1 Success.
- 0 Failure.

Description

`sm_dde_client_request` requests data from a DDE server. Call this function to update cold and warm link data on Panther screens.

This function can succeed only if the server application is already running; otherwise Panther posts an error message.

See Also

`sm_dde_client_connect_cold`, `sm_dde_client_connect_warm`, `sm_dde_client_paste_link_cold`, `sm_dde_client_paste_link_warm`

sm_dde_execute

Sends a command to a DDE server

```
#include <smmwuser.h>
int sm_dde_execute(char *server, char *topic, char *command);
```

`server`
: The server application's name—for example, `WINWORD`

`topic`
: The server topic, typically the file name of the spreadsheet or document—for example, `SALES.DOC`

`command`
: A command in the server application's syntax.

Environment Windows

Returns

- 1 Success.
- 0 Failure.

Description `sm_dde_execute` sends a command from a Panther client to a server application. The server decides how to execute this command.

This function can succeed only if the server application is already running; otherwise Panther posts an error message.

See Also `sm_dde_poke`

sm_dde_install_notify

Installs a callback function that executes on changes in warm link data

```
#include <smmwuser.h>
void sm_dde_install_notify(void (*callback)(char *, char *));
```

`callback`
: The name of the callback function to install.

Environment Windows

Description `sm_dde_install_notify` installs a function that Panther calls when it gets notification from a server that warm link data has changed. If no callback function is installed, Panther uses its own callback function to notify the application. After the application is notified, it must explicitly request the data by calling `sm_dde_client_request`.

Panther supplies two arguments to a callback function: the name of the screen, and the name of the field that contains the link data.

Declare a callback function as follows:

```
void callback(char* screenname, char *fieldname);
```

Example

```
/* Function to notify user of new data via a message and
        a checkbox.*/
   #include <smdefs.h>

void notify(s_name, f_name)
   char *s_name;
   char *f_name;
   {
      int g_occur;        /* group occurrence number */
      char *g_name;       /* group name              */
      char buff[128];
      sprintf(buff,"New data available for %s on %s",
              f_name, s_name);
      sm_d_msg_line(buff, 10);

  /* Locate next field, get group name, and use it to set a
      checklist item indicating that new data is available.
   */
```

```
    g_name=sm_ftog(sm_e_fldno(f_name,0) + 1, &g_occur);
   sm_select(g_name, g_occur);
}
```

See Also `sm_dde_client_request`

sm_dde_poke

Pokes data into a DDE server

```
#include <smmwuser.h>
int sm_dde_poke(char *server, char *topic, char *item,
   char *data);
```

`server`
: The server application's name—for example, `WINWORD`

`topic`
: The server topic, typically the file name of the spreadsheet or document—for example, `SALES.DOC`

`item`
: The server item—for example, `DDE_LINK1`

`data`
: The data to send to the server.

Environment Windows

Returns

1	Success.
0	Failure.

Description `sm_dde_poke` sends unsolicited data from a Panther client to a server application. The server decides whether to accept or reject this data. A connection to the server must already exist; however, a link to the specified topic and item is not required.

See Also `sm_dde_execute`

sm_dde_server_off

Disables DDE server activity

```
#include <smmwuser.h>
void sm_dde_server_off(void);
```

Environment Windows

Description `sm_dde_server_off` prevents Panther from acting as a DDE server.

See Also `sm_dde_server_on`

sm_dde_server_on

Enables DDE server activity

```
#include <smmwuser.h>
void sm_dde_server_on(void);
```

Environment	Windows
Description	`sm_dde_server_on` enables Panther to act as a DDE server.
See Also	`sm_dde_server_off`

sm_delay_cursor

Changes the state of the mouse pointer

```
int sm_delay_cursor (int state);
```

`state`

Specifies the cursor's new state with one of these arguments:

`SM_AUTO_BUSY_CURSOR`
: Toggles the mouse pointer between the default cursor and the delay cursor depending on whether the application is awaiting input or not. The default cursor appears whenever Panther is awaiting input.

`SM_BUSY_CURSOR`
: Changes the mouse pointer into the delay cursor.

`SM_DEFAULT_CURSOR`
: Restores the default cursor.

`SM_SAME_CURSOR`
: Leaves the mouse pointer unchanged. Use this argument to get the pointer's current state.

`SM_TEMP_BUSY_CURSOR`
: Temporarily changes the mouse pointer to the delay cursor. Panther restores the mouse pointer to the default cursor after Panther refreshes the screen.

Returns

- The mouse pointer's previous state, one of the arguments specified for the parameter `state`, excluding `SM_SAME_CURSOR`.

Description

`sm_delay_cursor` sets the mouse pointer to be either the default cursor or the delay cursor, or gets the mouse pointer's current state, according to the value of `state`. It can also specify to change the cursor's state automatically, depending on whether the application is awaiting input or not.

You can set the default cursor for a screen through the **`pointer`** property. In Windows and Motif, the default cursor is an arrow. The delay cursor in Windows is an hourglass; in Motif, the delay cursor is usually a wristwatch icon. You can change Motif's default cursor through the `pointerShape` resource.

Because character-mode Panther does not change the mouse pointer shape, `sm_delay_cursor` resets the background status line message to the value of `SM_WAIT` or `SM_READY`. Note that you can turn background status messages on and off through `sm_setstatus`.

sm_deselect

Deselects an occurrence in a selection group

```
int sm_deselect(char *selection_group, int grp_occurrence);
```

`selection_group`
The name of the selection group with the item to deselect.

`grp_occurrence`
The occurrence in `selection_group` to deselect.

Returns

-1	Arguments do not reference an occurrence.
0	Occurrence not previously selected.
1	Occurrence previously selected.

Description

`sm_deselect` lets you deselect an occurrence within a selection group. You can use `sm_select` to select a group occurrence.

See Also

`sm_select`

sm_dicname

Sets the repository name

```
int sm_dicname(char *filespec);
```

`filespec`
: The repository's name and, optionally, path. If no path is specified, Panther searches for the file according to the paths specified in `SMPATH`.

Environment C only

Returns

0	Success.
-1	Insufficient memory.
-2	Unable to find `filespec`.
-3	`filespec` is not a repository.

Description `sm_dicname` sets the name of the repository to open in the screen editor. You can also specify a repository by setting the `SMDICNAME` variable in your setup file to the desired repository's name. During an editing session, you can close and open repositories through the screen editor's File menu. Only one repository can be open at a time.

Example

```
#include <smdefs.h>

    /* Set the name of the application's repository
     * to /usr/app/common.dic .*/

    sm_dicname("/usr/app/common.dic");
```

sm_disp_off

Gets the cursor's offset in the current field

```
int sm_disp_off(void);
```

Returns

≥0	The difference between cursor's position and the start of the field.
-1	The cursor is not in a field.

Description

`sm_disp_off` returns the difference between the field's first position and the current cursor location. `sm_disp_off` ignores offscreen data. To get the total cursor offset in a shiftable field, use `sm_sh_off`.

See Also

`sm_sh_off`

sm_*dlength

Gets the length of a field's contents

```
int sm_dlength(int field_number);
int sm_e_dlength(char *field_name, int element);
int sm_i_dlength(char *field_name, int occurrence);
int sm_n_dlength(char *field_name);
int sm_o_dlength(int field_number, int occurrence);
```

`field_name`
`field_number`
The field with the data to evaluate.

`element`
The element in `field_name` with the data to evaluate.

`occurrence`
The occurrence in the field with the data to evaluate.

Returns

≥0	Length of field contents.
-1	The field is not found.

Description

`sm_dlength` returns the length of the data in the specified field or occurrence of a field. The length includes any data that is shifted offscreen and therefore out of view. The length excludes leading blanks in right-justified fields, and trailing blanks in left-justified fields.

Example

```
#include <smdefs.h>

    /* Save the contents of the "rank" field in a buffer
     * of the proper size. */

    char *save_rank;

    if ((save_rank = malloc(sm_n_dlength("rank") + 1)) == NULL)
    {
        report_error("malloc error.");
    }
    else
    {
        sm_n_getfield(save_rank, "rank");
    }
```

sm_do_uinstalls

Installs an application's event functions

```
void sm_do_uinstalls(void);
```

Environment

C only

Description

Event functions are installed with the library function `sm_install`. The call to this function is typically, but not necessarily, made by `sm_do_uinstalls`, whose source is in `funclist.c`.

`sm_do_uinstalls` is usually called by the main function. The provided source code calls the library function `sm_install` to install dummy function lists. You should replace these dummy calls with your own installation calls.

In general, you should install event functions after the call to `sm_initcrt`, which initializes the display. One exception applies: you should always install an initialization function before the call to `sm_initcrt`.

For more information about installing event functions, refer to "Installing Functions" on page 44-5 in *Application Development Guide*.

See Also

`sm_initcrt`, `sm_install`

sm_*doccur

Deletes occurrences from a field

```
int sm_i_doccur(char *field_name, int occurrence, int count);
int sm_o_doccur(int field_number, int occurrence, int count);
```

`field_name`
`field_number`
: The field with the occurrences to delete. In Panther 5.50 and later, `field_name` can also be a grid frame or a syncronized scrolling group.

`occurrence`
: The first occurrence to delete in the array specified by `field_number` or `field_name`.

`count`
: The number of occurrences to delete, starting with `occurrence`. If you supply a negative value, Panther inserts new occurrences above `occurrence`, with the same restrictions that apply to `sm_ioccur`.

Returns

≥0	The number of occurrences deleted.
-1	The field or occurrence number is out of range.
-3	Insufficient memory available.

Description

`sm_i_doccur` and `sm_o_doccur` delete data from `count` occurrences, starting with `occurrence`. If the array is scrolling, Panther then deallocates `count` occurrences. Panther moves up data in the occurrences after the last-deleted occurrence to prevent gaps in the array.

If `count` is equal to or greater than the number of allocated occurrences, Panther deletes all data from the array.

If other arrays are synchronized with this one, `sm_doccur` performs the same operation on them, provided their `clearing_protect` property is set to `PV_NO`. `sm_doccur` ignores the target array's `clearing_protect` setting.

You can use `sm_doccur` to insert new occurrences in a field by supplying a negative value for `count`. You can achieve the same effect with `sm_ioccur`.

This function is normally bound to the logical key DELL.

See Also `sm_ioccur`

sm_*drawingarea

Gets a handle to the current screen that can be passed to the window manager

```
#include <smmwuser.h>
HWND sm_mw_drawingarea(void);

#include <smxmuser.h>
Widget sm_xm_drawingarea(void);
```

Environment Motif, Windows; C only

Returns

- Success: On Windows, an HWND handle to the window; on Motif, a Widget ID.
- Failure: `NULL` if there is no current screen.

Description `sm_mw_drawingarea` and `sm_xm_drawingarea` get a handle to the current screen—in the case of Windows, a HWND handle; under Motif, a Widget ID. Use these functions with `sm_translatecoords` to place objects such as bitmapped graphics or custom widgets on a Panther screen. Refer to `sm_translatecoords` for a Windows example that uses this function.

Note: The Widget ID that `sm_xm_drawingarea` returns is not a recognizable X widget type. Consequently, you cannot directly call XmAddCallback with it. To use this Widget ID, you must call XmAddEventHandler.

See Also `sm_translatecoords`, `sm_widget`

sm_*dtofield

Writes a real number to a field

```
int sm_dtofield(int field_number, double value, char *format);
int sm_e_dtofield(char *field_name, int element, double value,
    char *format);
int sm_i_dtofield(char *field_name, int occurrence, double value,
    char *format);
int sm_n_dtofield(char *field_name, double value, char *format);
int sm_o_dtofield(int field_number, int occurrence, double value,
    char *format);
```

`field_name`
`field_number`
: The field to receive `value`.

`element`
: The element in `field_name` to receive `value`.

`occurrence`
: The occurrence in the field to receive `value`.

`value`
: The real number data to write.

`format`
: Specifies the format to apply to `value`. To supply a value of 0, cast the argument as follows: `(char *)0`.

Environment C only

Returns

0	Success.
-1	The field is not found.
-2	The field format properties are set for numeric display, but the formatted output is too wide for it.

Description `sm_dtofield` converts the real number `value` to user-readable format as specified by format. It then moves this value into the specified field with a call to `sm_amt_format`. If the `format` string is empty and the field's `data_formatting` property is set to

PV_NUMERIC, Panther uses the field's numeric formatting subproperties to determine precision. If data_formatting is set to PV_NONE, Panther uses the precision set by the behavior variable DECIMAL_PLACES.

You can round the number of decimal places to *n* places with the format string "%.*n*f". To truncate, use the format string "%t.*n*f".

Example

```
/* Place the value of pi on the screen, using the
     * formatting attached to the field. */

    sm_n_dtofield("pi", 3.14159, (char *)0);

    /* Do it again, using only three decimal places.
     sm_n_dtofield("pi", 3.14159, "%5.3f");
```

See Also sm_amt_format, sm_dblval

sm_femsg

Displays an error message and awaits user acknowledgement

```
void sm_femsg(int msg_num, char *message);
```

`msg_num`
: A Panther message number. If you supply a string value for `message`, Panther ignores this parameter.

`message`
: The error message to display. To use the `msg_num`-specified message, set this parameter to `NULL`.

Description

`sm_femsg` displays the specified message either on the status line or in a popup window and awaits user acknowledgement. This function also calls the error event function if one is installed.

Window versus Status Line Display

By default, GUI versions of Panther always display messages in a popup window with an OK button. Character-mode Panther always displays messages in a window only if the configuration variable `MESSAGE_WINDOW` is set to `ALWAYS`. If you set this variable to `WHEN_REQUIRED` (the default), Panther displays messages on the status line except when these conditions occur:

- The message overflows the status line. Note that Panther prevents the message from overlapping the cursor row/column display, if it is turned on.
- The message wraps to multiple lines.
- You specify window display with the `%W` format option.

Note: You can force display of a message to the status line on all GUI and character-mode platforms, regardless of `MESSAGE_WINDOW`'s setting, if the message contains the `%Mu` option, or the behavior variable `ER_KEYUSE` is set to `ER_USE`.

Message Acknowledgment

Users can dismiss the error message by pressing the acknowledgement key. In a window-displayed message, OK and space bar also serve to dismiss the error message. The acknowledgement key—by default, space bar—can be set through the behavior variable `ER_ACK_KEY`. If the user acknowledges the message through the keyboard, Panther discards the key. You can modify this behavior for individual messages through the `%Mu` option, described later.

Message Appearance and Behavior

Several behavior variables determine default message presentation and behavior. For more information about these variables, refer to Chapter 2, “Application Variables,” in *Configuration Guide*. You can change these defaults at runtime through `sm_option`.

You can change message behavior and appearance for individual messages by embedding percent escape options in the message text. Use these options after the call to `sm_initcrt`; otherwise, the percent characters appear as literals.

`%A`*`attr-value`*

Change the display of the subsequent string to the *`attr-value`*-specified attribute, where *`attr-value`* is a four-digit hexadecimal value. If the string to get the attribute change starts with a hexadecimal digit (`0...F`), pad *`attr-value`* with leading zeros to four digits. refer to Table 45-2 on page 45-9 in the *Application Development Guide* for valid attribute values.

This option is valid only for messages that display on the status line. Panther ignores this option if the message displays in a window.

`%B`

Beep the terminal with `sm_bel` before the message displays. This option must be at the beginning of the message.

`%K`*`key-logical`*

Display key label for logical key, where *`key-logical`* is a logical key constant. When Panther displays the message, it replaces *`key-logical`* with the key label string defined for that key in the key translation file. If there is no label, the `%K` is stripped out and the constant remains. Key constants are defined in `smkeys.h`

Note: If `%K` is used in a status line message, the user can push the corresponding logical key onto the input queue by mouse-clicking on the key label text.

`%Md`

Force the user to press the acknowledgment key (`ER_ACK_KEY`) in order to dismiss the error message. Panther discards the key that is pressed. If the user presses any other key, Panther displays an error message or beeps, depending on how behavior variable `ER_SP_WIND` is set. The `%Md` option corresponds to the default message behavior when behavior variable `ER_KEYUSE` is set to `ER_NO_USE`.

This option must precede the message text.

`%Mt[`*`time-out`*`]`

Force temporary display of message to the status line. Panther automatically dismisses the message after the specified timeout elapses and restores the previous status line display. Timeout specification is optional; the default timeout is one second. You can specify another timeout in units of 1/10 second with this syntax:

`#(`*`n`*`)`

where *n* is a numeric constant that specifies the timeout's length. If *n* is more than one digit, the value must be enclosed with parentheses. For example, this statement displays a message for 2 seconds:

```
err = sm_femsg(0, "%Mt(20)Changes saved to database.");
```

The user can dismiss the message before the timeout by pressing any key or mouse clicking. Panther then processes the keyboard or mouse input.

If the message is too long to fit on the status line, Panther displays the message in a window. In this case, users can dismiss the message only by choosing OK or pressing the acknowledgement key. Panther then discards any keyboard input.

This option must precede the message text.

`%Mu`

Force message display to the status line and permit any keypress to serve as both error acknowledgment and data entry. Panther processes the key that is pressed. This option must precede the message text. This option corresponds to default message behavior when behavior variable `ER_KEYUSE` is set to `ER_USE`.

If the message is too long to fit on the status line, Panther displays the message in a window. In this case, users can dismiss the message only by choosing OK or by pressing the acknowledgement key or space bar. Panther then discards any keyboard input used to dismiss the message.

`%N`

Insert a line break and force display of the message in a window.

`%W`

Force display of the message in a window. This option must be at the beginning of the message.

See Also

`sm_ferr_reset`, `sm_fqui_msg`, `sm_fquiet_err`

sm_ferr_reset

Displays an error message and awaits user acknowledgement

```
void sm_ferr_reset(int msg_num, char *message);
```

`msg_num`
: A Panther message number. If you supply a string value for `message`, Panther ignores this parameter.

`message`
: The error message to display. To use the `msg_num`-specified message, set this parameter to `NULL`.

Description

`sm_ferr_reset` displays the specified message either on the status line or in a popup window and awaits user acknowledgement. This function also calls the error event function if one is installed.

Window versus Status Line Display

By default, GUI versions of Panther always display messages in a popup window with an OK button. Character-mode Panther always displays messages in a window only if the configuration variable `MESSAGE_WINDOW` is set to `ALWAYS`. If you set this variable to `WHEN_REQUIRED` (the default), character-mode Panther displays messages on the status line except when these conditions occur:

- The message overflows the status line. Note that Panther prevents the message from overlapping the cursor row/column display, if it is turned on.
- The message wraps to multiple lines.
- You specify window display with the `%W` format option.

Note: You can force display of a message to the status line on all GUI and character-mode platforms, regardless of `MESSAGE_WINDOW`'s setting, if the message contains the `%Mu` option, or the behavior variable `ER_KEYUSE` is set to `ER_USE`.

`sm_ferr_reset` and `sm_femsg` function identically when messages are displayed in a window. If the message is displayed on the status line, `sm_ferr_reset` forces the cursor on at the current field and forces off global flag `sm_do_not_display`.

Message Acknowledgment

Users can dismiss the error message by pressing the acknowledgement key. In a window-displayed message, OK and space bar also serve to dismiss the error message. The acknowledgement key—by default, space bar—can be set through the behavior variable `ER_ACK_KEY`. If the user acknowledges the message through the keyboard, Panther discards the key. You can modify this behavior for individual messages through the `%Mu` option (described under `sm_femsg`).

Several behavior variables determine default message presentation and behavior. For more information about these variables, refer to Chapter 2, “Application Variables,” in *Configuration Guide*. You can change these defaults at runtime through `sm_option`.

You can also change message behavior and appearance for individual messages through percent escapes embedded in the message text (described under “Message Appearance and Behavior”).

See Also

`sm_femsg`, `sm_fqui_msg`, `sm_fquiet_err`

sm_ffree

Free memory allocated by sm_fmalloc

```
sm_ffree(VOIDPTR ptr);
```

`ptr`
: Pointer to memory allocated by `sm_fmalloc` or functions like `sm_strdup` that call `sm_fmalloc`.

Description .Unless `ptr` is the null pointer, the allocated memory that `ptr` points to is freed.

See Also `sm_fmalloc`, `sm_strdup`

sm_fi_path

Returns the full path name of a file

```
char *sm_fi_path(char *file_name);
```

`file_name`
A pointer to the name of the file whose path is sought.

Returns

- A pointer to a static buffer that contains the path.
- 0: The file cannot be found on any path.

Description

`sm_fi_path` finds the full path name of a file. The file can be a screen or any other type of file. `sm_fi_path` returns a pointer to a static buffer that contains the file's full path name.

Panther searches for `file_name` in the current directory, then along the path given to `sm_initcrt`, and finally along the path defined by `SMPATH`.

If the file is found, the full path name is returned to the caller. Because the static buffer used to hold the full path name is shared by several functions, it should be used or copied immediately.

Example

```
char *file, *path;
   ...
   if ((path = sm_fi_path(file)) == NULL)
       sm_femsg(0, "Unable to find file");
   else
       sm_d_msg_line(path, INHERITED);
   endif
```

sm_file_copy

Copies a file

```
int sm_file_copy(char *source, char *destination, char *mode);
```

`source`, `destination`
: The paths of the file in its original and new locations. `source` and `destination` must be different file paths.

  `destination` must include a file name; implicit copying to the same name as the source file yields an error. In three-tier applications, the path to `source` and `destination` can include a file access server ID in this format:

  `[`*`server-id`*`!]`*`path`*

  If you omit *`server-id`*, Panther looks for the file locally.

`mode`
: Specifies whether to perform a text or binary copy; supply one of these arguments:

`"b"`	Binary transfer
`"t"`	Text transfer

Returns

0 Success

-1 Failure: the specified source file does not exist; an invalid argument was supplied for `mode`; or another I/O error occurred.

- `TP_INVALID_CONNECTION`: Unable to connect to the specified server.

Description

`sm_file_copy` copies the specified file. If the destination file does not exist, `sm_file_copy` creates it; if it already exists, the function overwrites it.

In a three-tier environment, you can copy files to and from a remote file access server by prefixing the file path with the server ID. For example, this JPL copies file `rpt.out` from server `oak` to the local client:

```
vars err = ""
if sm_file_exists("oak!/disk/reports/rpt.out")
{
   err = sm_file_copy \
       ("oak!/disk/reports/rpt.out", "c:\reports\rpt.out", "b")
   if err == 0 && cleanup() == 1
       call sm_file_remove("oak!/disk/reports/rpt.out")
}
```

See Also `sm_file_exists`, `sm_file_move`, `sm_file_remove`

sm_file_exists

Checks whether a file exists

```
int sm_file_exists (char *path);
```

`path`

Specifies the file to check. In three-tier applications, the path can include a file access server ID in this format:

`[server-id !]path`

If you omit *`server-id`*, Panther looks for the file locally.

Returns

1	File exists.
0	File does not exist.
-1	Failure: An I/O error occurred.
●	`TP_INVALID_CONNECTION`: Unable to connect to the specified server.

Description

`sm_file_exists` lets you ascertain whether a file exists. In a three-tier environment, you can check a file on a remote file access server by prefixing the file path with the server ID.

For example, this JPL verifies the existence of file `rpt.out` on server `oak` before moving it to the local machine:

```
if sm_file_exists("oak!/disk/reports/rpt.out")
{
   call sm_file_move \
   ("oak!/disk/reports/rpt.out", "c:\reports\rpt.out", "b")
}
```

Note: On file systems that allow file-level permissions, `sm_file_exists` only verifies the existence of files for which the user has read permission.

See Also

`sm_file_copy`, `sm_file_move`, `sm_file_remove`

sm_file_move

Copies a file and deletes its source

```
int sm_file_move(char *source, char *destination, char *mode);
```

`source, destination`

The paths of the file in its original and new locations. `source` and `destination` must be different file paths.

`destination` must include a file name; implicit copying to the same name as the source file yields an error. In three-tier applications, the path to source and destination can include a file access server ID in this format:

`[`*`server-id`*`!]`*`path`*

If you omit *`server-id`*, Panther looks for the file locally.

`mode`

Specifies whether to perform a text or binary copy; supply one of these arguments:

`"b"`	Binary transfer
`"t"`	Text transfer

Returns

0	Success
-1	Failure: the specified source file does not exist; or another I/O error occurred.

- `TP_INVALID_CONNECTION`: Unable to connect to the specified server.

Description

`sm_file_move` copies a file to the specified destination. If the destination file does not exist, `sm_file_move` creates it; if it already exists, the function overwrites it. When the copy operation is complete, the function deletes the source file. Calling this function is equivalent to successive calls to `sm_file_copy` and `sm_file_remove`.

In a three-tier environment, you can move files to and from remote file access servers by prefixing the file path with the server ID. For example, this JPL moves file `rpt.out` from server `oak` to the local machine:

```
if sm_file_exists("oak!/disk/reports/rpt.out")
{
    call sm_file_move \
        ("oak!/disk/reports/rpt.out", "c:\reports\rpt.out", "b")
}
```

See Also `sm_file_copy`, `sm_file_exists`, `sm_file_remove`

sm_file_remove

Deletes a file

```
int sm_file_remove(char *path);
```

`path`

Specifies the file to delete. In three-tier applications, the path can include a file access server ID in this format:

`server-id``!`*`path`*

If you omit *`server-id`*, Panther looks for the file locally.

Returns

0	Success
-1	Failure: the specified file does not exist; or another I/O error occurred.
●	`TP_INVALID_CONNECTION`: Unable to connect to the specified server.

Description

`sm_file_remove` deletes a file. In a three-tier environment, you can remove a file from a remote file access server by prefixing the file path with the server ID. For example, this JPL deletes file `rpt.out` from server oak after moving it to the local workstation client:

```
vars err = ""
if sm_file_exists("oak!/disk/reports/rpt.out")
{
    err = sm_file_copy \
        ("oak!/disk/reports/rpt.out", "c:\reports\rpt.out", "b")
    if err == 0 && cleanup() == 1
        call sm_file_remove("oak!/disk/reports/rpt.out")
}
```

See Also

`sm_file_copy`, `sm_file_exists`, `sm_file_move`

sm_filebox

Opens a file selection dialog box

```
void int sm_filebox(char *buffer, int length, char *path,
    char *file_mask, char *title, int open_save);
```

`buffer`
: On return, contains the selected file's name. Make sure that `buffer` is at least the size specified by `length`.

`length`
: The length of `buffer`.

`path`
: The initial path for the directory tree. If you supply an empty string, the dialog box initially shows the directory in which the Panther application was launched.

`file_mask`
: A filter to narrow down the display of files in `path`. Use at least one wildcard character. For example, to narrow down the display to all files that have the extension doc, supply `"*.doc"` as the argument.

  To show all files, supply an empty string.

`title`
: The text of the dialog box's title. Supply an empty string to suppress title display.

`open_save`
: Valid only for Windows, determines the title of the file type option menu; ignored by other platforms. The title is platform-specific; for example, in Windows, `FB_OPEN` sets the title to List Files of Type.

Environment C only

Returns

1	Success: the user chose OK and Panther copied the filename to `buffer`.
0	The user chose Cancel. No text is copied to `buffer`.
-1	Failure: A malloc error occurred or `buffer` was too small.

Description

sm_filebox invokes a file selection box that lets users choose a file to open or save a file. On GUI platforms, Panther uses the GUI's standard file selection dialog. The dialog box initially displays the contents of the path-specified directory, and lists files that match the wildcard specification in file_mask. Users can browse through the directory tree. When the user chooses OK, Panther copies to buffer the name of the file to open or save.

If you are running an application on Windows, Panther uses the value of open_save to change the title of the file type option menu. You specify the option menu's contents through sm_filetypes.

Example

```
#include <smdefs.h>

#define LEN 256
   char buf [LEN];

sm_filebox(buf, LEN, "c::\\videobiz", "*.tbl", "", FB_OPEN);
```

See Also

sm_filetypes, sm_jfilebox

sm_filetypes

Adds an option to the file type option menu

```
int sm_filetypes(char *option_text, char *filters);
```

`option_text`
: The text of the option to display on the file type option menu.

`filters`
: A semicolon-separated list of file masks that specify the files selected through `description`.

Environment Windows

Returns

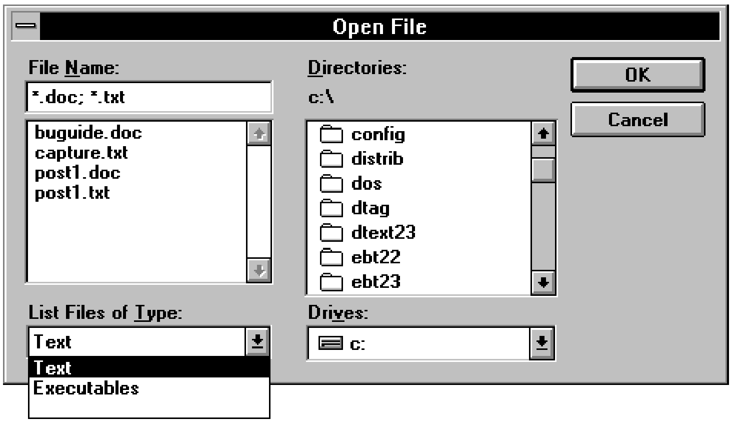
0	The description is successfully added to the list.
-1	A memory allocation error occurred.

Description `sm_filetypes` defines a file type and adds it to the option menu that Panther displays in a file selection dialog box. This menu gives users an easy way to specify which files to show in the current directory.

You build the option menu through repeated calls to `sm_filetypes`. For example, the following statements define two files types, Text and Executables:

```
sm_filetypes("Text", "*.doc; *.txt");
sm_filetypes("Executables", "*.com; *.exe; *.bat");
```

The dialog box subsequently invoked by `sm_filebox` or `sm_jfilebox` contains an option menu with these file types. Options are displayed in order of their definition:



To change the menu, first reinitialize the current one by calling `sm_filetypes` with `NULL` arguments or empty strings, as in this JPL statement:

```
call sm_filetypes("", "")
```

See Also   `sm_filebox`

sm_fio_a2f

Writes the contents of an array to a file

```
int sm_fio_a2f(char *file_name, char *array_name);
```

`file_name`
: The name of the target file.

`array_name`
: The name of a Panther widget that serves as the source array.

Returns

0	Success.
-4	`SMFIO_IO_ERROR`: Error during write operation.
-7	`SMFIO_OPEN_ERROR`: Unable to open file—for example, because the file does not exist or is protected.
-8	`SMFIO_FIELD_ERROR`: Nonexistent field.
-13	`SMFIO_GETFIELD`: Unable to read the field's contents.

Description

`sm_fio_a2f` writes the contents of the specified array to a file. The contents of each occurrence are written as a single line to the file.

Example

```
proc array2file()
   vars fileName, retErr

/* get the file name sent from previous dialog */
   receive DATA fileName

   /* put array's contents into file */
   retErr = sm_fio_a2f(fileName, "comments")
   if retErr != 0
   {
     msg emsg "Error - error number :retErr"
   }
   return
```

See Also

`sm_fio_f2a`

sm_fio_close

Closes an open file stream

```
int sm_fio_close(int file_stream);
```

`file_stream`
A handle to the file to close, obtained by `sm_fio_open`.

Returns

0	Success.
-1	`SMFIO_INVALID_HANDLE`: Invalid file handle.
-2	`SMFIO_HANDLE_CLOSE`: Handle points to closed file.
-4	`SMFIO_IO_ERROR`: Standard I/O error. Check the value in system variable `errno` to determine the nature of the error.

Description

`sm_fio_close` closes the specified file and releases its handle for reuse. You should call this function after all read and write operations that require an open file stream—for example, after calling `sm_fio_gets`.

This function is similar to the C function `fclose`, except that `sm_fio_close` takes an integer argument so that it can be called from JPL.

See Also

`sm_fio_open`

sm_fio_editor

Invokes an external text editor for an array

```
int sm_fio_editor(char *array_name);
```

`array_name`
The name of the array whose contents you wish to edit.

Returns

0	Success.
-4	`SMFIO_IO_ERROR`: Standard I/O error. Check the value in system variable `errno` to determine the nature of the error.
-8	`SMFIO_FIELD_ERROR`: Nonexistent field.
-9	`SMFIO_FILE_TRUNCATE`: Array not large enough to accept all file data; partial read was successful.
-10	`SMFIO_LINE_BREAK`: One or more lines in the file were too long and wrapped to the next occurrence.
-11	`SMFIO_NO_EDITOR`: Panther behavior variable `SMEDITOR` is undefined; no editor is available to handle the operation.
-12	`SMFIO_PUTFIELD`: Unable to write to the field.
-13	`SMFIO_GETFIELD`: Unable to read the field's contents.

Description

`sm_fio_editor` invokes the editor specified in the behavior variable `SMEDITOR` and writes the contents of `array_name` to a temporary file. Each occurrence is written as a single line to that file.

When you exit the editor, Panther writes the edited text back to the array. Panther attempts to write each line in the file to a single occurrence. If any line is too long for its target occurrence, Panther breaks the line and writes the overflow text to the next occurrence. If the array contains too few occurrences to read the entire file, `sm_fio_editor` discards the excess text.

sm_fio_error

Gets the error returned by the last call to a file I/O function

```
int sm_fio_error(void);
```

Returns

0	Success.
-1	`SMFIO_INVALID_HANDLE`: Invalid file handle.
-2	`SMFIO_HANDLE_CLOSE`: Handle points to closed file.
-3	`SMFIO_EOF`: Already at end of file.
-4	`SMFIO_IO_ERROR`: Standard I/O error. Check the value in system variable `errno` to determine the nature of the error.
-5	`SMFIO_INVALID_MODE`: Invalid mode specified for open operation.
-6	`SMFIO_NO_HANDLES`: All available file handles currently in use.
-7	`SMFIO_OPEN_ERROR`: Unable to open the file—for example, because it does not exist or is protected.
-8	`SMFIO_FIELD_ERROR`: Nonexistent field.
-9	`SMFIO_FILE_TRUNCATE`: Array not large enough to accept all file data; partial read was successful.
-10	`SMFIO_LINE_BREAK`: One or more lines in the file were too long and wrapped to the next occurrence.
-11	`SMFIO_NO_EDITOR`: Panther behavior variable `SMEDITOR` is undefined; no editor is available to handle the operation.
-12	`SMFIO_PUTFIELD`: Unable to write to the field.
-13	`SMFIO_GETFIELD`: Unable to read the field's contents.

Description

`sm_fio_error` gets the last value returned by a file I/O function. Use this function after calling `sm_fio_gets` and `sm_fio_handle`, which respectively return an empty string and `NULL` when an error occurs. In both cases, you must call `sm_fio_error` to determine the actual cause of the error.

Note: Because the same error code variable is shared by all JPL file I/O routines, you should call `sm_fio_error` before making any other I/O operations with Panther library functions.

Example

```
/* Write the contents of an ASCII file to a single- *
    * line text array. The file stream handle was      *
    * obtained earlier by a call to sm_fio_open()      *
    */
```

```
proc getStr()
   {
     vars str, occurNo, err, fileStream, maxOccurs
     call sm_fio_error_set(0)

  /* get array size */
     maxOccurs = @widget("comments")->max_occurrences

  /* get file stream handle sent from previous dialog */
     receive BUNDLE f_handle DATA fileStream

  /* loop through array occurrences */
     for occurNo = 1 && err = 0 \
         while (err == 0 && occurNo <= maxOccurs)
     {
        /* get the next string in file stream */
        str = sm_fio_gets(fileStream, 32)

     /* check for error condition like EOF */
        if (str == "")
        {
           err = sm_fio_error()
        }
        /* read string into occurrence */
        comments[occurNo] = str
     }

  /* close the file stream when done */
     call sm_fio_close(fileStream)
     return
   }
```

sm_fio_error_set

Sets the file I/O error

```
int sm_fio_error_set(int new_error);
```

`new_error`
: The error code to set, one of the file I/O error codes shown in the Returns section below.

Returns The value returned by the last call to a file I/O function, one of the following:

0	Success.
-1	`SMFIO_INVALID_HANDLE`: Invalid file handle.
-2	`SMFIO_HANDLE_CLOSE`: Handle points to closed file.
-3	`SMFIO_EOF`: Already at end of file.
-4	`SMFIO_IO_ERROR`: Standard I/O error. Check the value in system variable `errno` to determine the nature of the error.
-5	`SMFIO_INVALID_MODE`: Invalid mode specified for open operation.
-6	`SMFIO_NO_HANDLES`: All available file handles currently in use.
-7	`SMFIO_OPEN_ERROR`: Unable to open the file—for example, because it does not exist or is protected.
-8	`SMFIO_FIELD_ERROR`: Nonexistent field.
-9	`SMFIO_FILE_TRUNCATE`: Array not large enough to accept all file data; partial read was successful.
-10	`SMFIO_LINE_BREAK`: One or more lines in the file were too long and wrapped to the next occurrence.
-11	`SMFIO_NO_EDITOR`: Panther behavior variable `SMEDITOR` is undefined; no editor is available to handle the operation.
-12	`SMFIO_PUTFIELD`: Unable to write to the field.
-13	`SMFIO_GETFIELD`: Unable to read the field's contents.

Description `sm_fio_error_set` sets the error code for Panther's file I/O processing functions. Use this function to clear the last-reported error.

For an example of this function, refer to `sm_fio_error`.

sm_fio_f2a

Writes a file's contents to an array

```
int sm_fio_f2a(char *file_name, char *array_name);
```

`file_name`
: The name of the file to read.

`array_name`
: The name of a Panther widget.

Returns

0	Success.
-4	`SMFIO_IO_ERROR`: Standard I/O error. Check the value in system variable `errno` to determine the nature of the error.
-7	`SMFIO_OPEN_ERROR`: Unable to open the file—for example, because it does not exist or is protected.
-8	`SMFIO_FIELD_ERROR`: Nonexistent field.
-9	`SMFIO_FILE_TRUNCATE`: Array not large enough to accept all file data; partial read was successful.
-10	`SMFIO_LINE_BREAK`: One or more lines in the file were too long and wrapped to the next occurrence.
-12	`SMFIO_PUTFIELD`: Unable to write to the field.

Description

`sm_fio_f2a` writes the contents of a file to an array. All previous text in the array is overwritten. If the array belongs to a synchronized scrolling group, the data of other members in the group is unaffected.

Panther attempts to write each line in the file to a single occurrence. If any line is too long for its target occurrence, Panther breaks the line and writes the overflow text to the next occurrence. If the array contains too few occurrences to read the entire file, `sm_fio_f2a` discards the excess text.

Example

```
proc file2array()
   vars fileName, retErr

/* get file name sent from previous dialog */
   receive DATA fileName

/* put file's contents into array*/
   retErr = sm_fio_f2a(fileName, "comments")
```

```
if retErr != 0
    {
      call io_errproc(retErr)
    }
    return
```

See Also `sm_fio_a2f`

sm_fio_getc

Reads the next byte from the specified file stream

```
int sm_fio_getc(int file_stream);
```

`file_stream`
A handle to the required file stream, obtained by `sm_fio_open`.

Returns

≥0	Next character in the file stream as an integer.
-3	`SMFIO_EOF`: Already at end of file.
-4	`SMFIO_IO_ERROR`: Standard I/O error. Check the value in system variable `errno` to determine the nature of the error.

Description

`sm_fio_getc` reads a character from the specified file stream and returns the result as an integer. This function is similar to the C function `fgetc` and is intended to read the contents of binary files.

Note: This function only returns the ASCII integer value of the read character.

sm_fio_gets

Reads a line from a file

```
char *sm_fio_gets(int file_stream, int maxlen);
```

`file_stream`
A handle to the required file stream, obtained by `sm_fio_open`.

`maxlen`
The number of bytes to read.

Returns

- A pointer to the string read from `file_stream`.
- An empty string if an error occurred.

Description

`sm_fio_gets` reads `maxlen` bytes from the current line in `file_stream` or to the end of the line and returns that string. If the current line is shorter than `maxlen`, `sm_fio_gets` only reads up to the end of the line. If the current line is longer than `maxlen`, the function returns only `maxlen` characters and sets the error code to `SMFIO_LINE_BREAK`. The next read operation on this file stream by `sm_fio_gets` continues where the last read ended. This function strips newline characters before reading it into the return value.

If the read operation fails, the function returns an empty string and sets the appropriate error code. You can get this error code by calling `sm_fio_error`. Because an empty string can also be a valid return value—for example, the file stream contains a blank line—you should interleave calls to `sm_fio_gets` with calls to `sm_fio_error` to determine whether an error condition exists and to ascertain its nature. `sm_fio_gets` can set one of these error codes:

`SMFIO_INVALID_HANDLE`	Invalid file handle.
`SMFIO_HANDLE_CLOSE`	Handle points to closed file.
`SMFIO_EOF`	Already at end of file.
`SMFIO_IO_ERROR`	Standard I/O error. Check the value in system variable `errno` to determine the nature of the error.
`SMFIO_LINE_BREAK`	The line is longer than `maxlen` characters.

Note: Because the same error code variable is shared by all JPL file I/O routines, you should call `sm_fio_error` before calling any other I/O library functions.

Example

```
/* Write the contents of an ASCII file to a single- *
     * line text array. The file stream handle was      *
     * obtained earlier by a call to sm_fio_open()      *
     */

proc getStr()
    {
      vars str, occurNo, err, fileStream, maxOccurs
      call sm_fio_error_set(0)

  /* get array size */
      maxOccurs = @widget("comments")->max_occurrences

  /* get file stream handle sent from previous dialog */
      receive BUNDLE f_handle DATA fileStream

  /* loop through array occurrences */
      for occurNo = 1 && err = 0 \
          while (err == 0 && occurNo <= maxOccurs)
      {
         /* get the next string in file stream */
         str = sm_fio_gets(fileStream, 32)

     /* check for error condition like EOF */
         if (str == "")
         {
            err = sm_fio_error()
         }
         /* read string into occurrence */
         comments[occurNo] = str
      }

  /* close the file stream when done */
      call sm_fio_close(fileStream)
      return
    }
```

sm_fio_handle

Gets a handle to an open file

```
FILE *sm_fio_handle(int file_stream);
```

`file_stream`

A handle to the required file stream, obtained by `sm_fio_open`.

Environment C only

Returns

- `FILE *` pointer to the specified file.
- `NULL`: Failure—for example, the file is closed. Call `sm_fio_error` to ascertain the nature of the failure.

Description `sm_fio_handle` gets a `FILE` * pointer to a JPL file stream opened by `sm_fio_open`. You can pass this handle to routines written in C. This function lets you write your own extensions to Panther file I/O functions.

Note: This function cannot be called from JPL.

sm_fio_open

Opens the specified file and returns a handle to the JPL caller

```
int sm_fio_open(char *path, char *mode);
```

`path`
: Path name of file to open.

`mode`
: Describes the file type—binary or text—and type of access required, one of the following constants described in Table 5-8 in Description.

Returns

- ≥0 A handle to the opened file.
- -5 `SMFIO_INVALID_MODE`: Invalid mode specified for open operation.
- -6 `SMFIO_NO_HANDLES`: All available file handles currently in use.
- -7 `SMFIO_OPEN_ERROR`: Unable to open the file—for example, because it does not exist or is protected.

Description

`sm_fio_open` opens a file in the specified mode and returns an integer handle to a file stream that is accessible to other Panther library file I/O functions. Supply this handle to these functions for all subsequent I/O operations on the file stream.

Note: The file stream that is opened by `sm_fio_open` is not accessible to standard C library functions.

You can open a file in one of the modes shown in Table 5-8:

Table 5-8 File access modes

Mode identifier	Access description
`r`	Open read-only text file.
`rb`	Open read-only binary file.
`w`	Create write-only text file.
`wb`	Create write-only binary file.
`a`	Open text file for append.

Table 5-8 File access modes *(Continued)*

Mode identifier	Access description
`ab`	Open binary file for append.
`r+b`	Open binary file for update.
`w+b`	Create binary file for update.
`a+b`	Open binary file for append or update.

If a server executes this command, it must have the necessary permissions for the requested operation to the specified path. For example, when a Web client issues `sm_fio_open` to create or open a file, the Web application server's user ID must have write permission to that file's directory.

Example

```
// this validation routine is attached to a
    // push button on a dialog screen that gets the
    // user-entered name of a file and opens it

vars fileStream = SMFIO_INVALID_HANDLE
   vars operation
   receive BUNDLE mode DATA operation

if (operation == "w")
   {
      fileStream = getFileHandle(file, "w")
   }
   if (operation == "r")
   {
     fileStream = getFileHandle(file, "r")
   }

   // All-purpose routine for supplying file handles
   proc getFileHandle(fileName, mode)
   vars fileHandle
   fileHandle = sm_fio_open(fileName, mode)
   if fileHandle < 0
   {
     msg emsg "I/O error :fileHandle - reenter file name
     sm_n_gofield("fileName")
   }

if fileHandle >= 0
   {
     send BUNDLE f_handle DATA fileHandle
   }
   return
```

sm_fio_putc

Writes a single byte to an open file

```
int sm_fio_putc(int byte, int file_stream);
```

`byte`
: An ASCII integer value to write. Attempts to write any other kind of value—for example, a string—yield an error.

`file_stream`
: A handle to the file to write to, obtained by `sm_fio_open`.

Returns

0	Success
-1	`SMFIO_INVALID_HANDLE`: Invalid file handle.
-2	`SMFIO_HANDLE_CLOSE`: Handle points to closed file.
-4	`SMFIO_IO_ERROR`: Standard I/O error. Check the value in system variable `errno` to determine the nature of the error.

Description

`sm_fio_putc` writes the specified integer character—a single byte—to a file opened by `sm_fio_open`. The value should be the integer value of an ASCII character. Call this function only for file streams opened by `sm_fio_open`. JPL. Routines that are written in C should call `fputc`. Do not call Panther and C functions on the same I/O stream.

Be sure to call `sm_fio_close` on `file_stream` after you finish writing the data; the actual write operation is not complete until the handle to this file stream is released.

sm_fio_puts

Writes a line of text to an open file

```
int sm_fio_puts(char *string, int file_stream);
```

`string`

Character string to be output.

`file_stream`

A handle to the file to write to, obtained by `sm_fio_open`.

Returns

0	Success.
-1	`SMFIO_INVALID_HANDLE`: Invalid file handle.
-2	`SMFIO_HANDLE_CLOSE`: Handle points to closed file.
-4	`SMFIO_IO_ERROR`: Standard I/O error. Check the value in system variable `errno` to determine the nature of the error.

Description

`sm_fio_puts` writes the contents of `string` to the specified open file and appends a newline `\n` character. Be sure to call `sm_fio_close` on `file_stream` after you finish writing the data; the actual write operation is not complete until the handle to this file stream is released.

Example

```
proc putStr()
    {
      vars str, occurNo, err, fileStream, maxOccurs
      call sm_fio_error_set(0)

  /* get array size */
      maxOccurs = @widget("comments")->max_occurrences

  /*get file stream handle sent from previous dialog */
      receive BUNDLE f_handle DATA fileStream

  /* loop through array occurrences */
      for occurNo = 1 && err = 0 \
          while (err == 0 && occurNo <= maxOccurs)
      {
         /* get string in current occurrence */
         str = comments[occurNo]

     /* put string into next line of file stream */
```

```
       err = sm_fio_puts(str, fileStream)
    }

/* close file stream when done */
    call sm_fio_close(fileStream)
    return
  }
```

sm_fio_rewind

Resets the file stream to the beginning of a file

```
int sm_fio_rewind(int file_stream);
```

`file_stream`
A handle to the file to rewind, obtained by `sm_fio_open`.

Returns

0	Success.
-1	`SMFIO_INVALID_HANDLE`: Invalid file handle.
-2	`SMFIO_HANDLE_CLOSE`: Handle points to closed file.
-4	`SMFIO_IO_ERROR`: Standard I/O error. Check the value in system variable `errno` to determine the nature of the error.

Description

`sm_fio_rewind` resets the specified file stream to the file's beginning—for example, in order to re-read a file's contents.

sm_flush

Flushes delayed writes to the display

```
void sm_flush(void);
```

Description

`sm_flush` performs delayed writes and flushes all buffered output to the display. It is called automatically by `sm_input` when the keyboard is opened and there are no keystrokes available—that is, typed ahead.

Frequent calls to this function can significantly slow execution. Because it is called whenever the keyboard opens, the display is always up to date before data entry occurs.

You must use this function if you want timed output or other non-interactive display.

Example

```
#include <smdefs.h>

   /* Update a system time field once per second,
    * until a key is pressed. */

   while (!sm_keyhit(10))
   {
      sm_n_putfield("time_now", "");
      sm_flush();
   }

   /* ...process the key */
```

See Also

`sm_m_flush`, `sm_rescreen`

sm_fmalloc

Allocate memory

```
VOIDPTR sm_fmalloc(unsigned size);
```

`size`
    Number of bytes of memory to allocate.

Returns .
- The value returned by `malloc`.
- the null pointer if `size` is zero or if the call to `malloc` failed.

Description .If `size` is not zero, the C library function `malloc` is called and the value it returns is returned. Memory allocated by calling `sm_fmalloc` should be freed by calling `sm_ffree`.

See Also `sm_ffree`, `sm_strdup`

sm_*form

Opens a screen as a form

```
int sm_d_form(char *screen_address);
int sm_l_form(int lib_desc, char *screen_name);
int sm_r_form(char *screen_name);
```

`screen_address`
A pointer to the screen's address in memory.

`lib_desc`
Specifies the library in which `screen_name` is stored, where `lib_desc` is an integer returned by `sm_l_open`. You must call `sm_l_open` before you read any screens from a library.

`screen_name`
The name of the screen.

Returns

0	Success.
-1	Screen file's format is incorrect; previous form still displayed and available.
-2	The screen cannot be found or the maximum allowable number of files is already open; previous form still displayed and available.
-4	Unable to read the specified screen after the previous screen closed.
-5	Insufficient memory available to display the screen.

Description

`sm_form` and its variants open a screen as a form. Because these functions do not update the form stack, use them only with your own executive. To open a form while under Panther control, use a control string or `sm_jform`. To display a screen as a window, use `sm_r_window`, or one of its variants.

`sm_form` displays the named screen as a form. In so doing, it discards the previously displayed form and its window stack and frees their memory. The screen displays with its upper left-hand corner at the display's upper left position (0,0).

If the function returns an error code of -1 or -2, the previously displayed form remains on display and available for use. Other negative return codes indicate that the display is undefined. The caller should display another form before using screen manager functions.

If the form is stored in a library, you can use sm_l_form to display it. If the form is memory-resident, you can use sm_d_form. sm_r_form looks for the form in all possible areas, including the disk.

Search Path

When you use sm_r_form, Panther looks for the named screen in the following locations in this order:

1. The application's memory-resident list; if found, sm_d_form is called to display the screen.
2. All open libraries; if found, sm_l_form is called to display the screen.

If the search fails and the supplied file name has no extension, Panther appends the SMFEXTENSION-specified extension to the file name and repeats the search. If all searches fail, sm_r_form displays an error message and returns.

Memory-resident Screens

You can save processing time by using sm_d_form to display memory-resident screens. Memory-resident screens are useful in applications with a limited number of screens, and in environments with a slow disk. A memory-resident screen never changes at runtime, so it can be made sharable on systems that support sharing read-only data. sm_r_form can also display memory-resident screens if they are properly installed with sm_formlist. To create memory-resident screens, use bin2c to convert editable screens from disk files to program data structures that you can compile into your application.

Screens Stored in Libraries

You can also save processing time with sm_l_form to display screens from a library. A library is a single file that stores screens, JPL modules, and menus. You can assemble a library from individual screen files with formlib. Libraries let you distribute a large number of screens with an application, and can improve efficiency by reducing the number of search paths.

Example

```
#include <smdefs.h>
   #include <setjmp.h>

   /* If an abort condition exists, read in a special
    * form to handle that condition, discarding all
    * open windows. */

   extern jmp_buf re_init;

   if (sm_isabort(ABT_OFF) > 0)
   {
      sm_r_form("badstuff");
```

```
        if (sm_message_box("Do you want to continue?", 0,
            SM_MB_YESNO, 0) == SM_IDYES)
           longjmp(re_init);
        else sm_cancel();
     }
```

See Also sm_r_window, sm_r_at_cur, sm_formlist

sm_formlist

Updates the list of memory-resident files

```
int sm_formlist(struct form_list *ptr_to_form_list);
```

`ptr_to_form_list`
: A pointer to the form list to update.

Environment
: C only

Returns
: 0 Success.
: -1 Insufficient memory is available for the new list.

Description

`sm_formlist` adds JPL modules and screens to the memory-resident form list. Each member of the list is a structure that contains the name of the JPL module or screen as a character string and its address in memory. You usually call this function from `main`. You can also call it elsewhere in an application program to augment to the memory resident list.

The library functions `sm_r_form`, `sm_r_window`, and `sm_r_at_cur` search for the specified screen in the memory-resident list before they try to read it from disk. The `call` command and library function `sm_jplcall` search the memory-resident list when they look for a JPL procedure to execute.

Because no count is given with the list, be careful to end the list with a null entry.

To make a JPL module or screen memory resident:

1. Use the `bin2c` utility to create a static C structure initialized with the binary content of the object.

2. Compile and link the structure with the application executable.

Alternatively, read the object into memory after opening it with the C function `fopen`.

Example

```
#include <smdefs.h>

   /* Add 2 screens to memory-resident form list. */

   struct form_list new_list[] =
   {
      {"new_form1",   new_form1},
      {"new_form2",   new_form2},
      {0,   0}
   };

   sm_formlist(new_list);
```

See Also `sm_rmformlist`

sm_*fptr

Gets the contents of a field

```
char *sm_fptr(int field_number);
char *sm_e_fptr(char *field_name, int element);
char *sm_i_fptr(char *field_name, int occurrence);
char *sm_n_fptr(char *field_name);
char *sm_o_fptr(int field_number, int occurrence);
```

`field_name`
`field_number`
: The field with the data to get.

`element`
: The element that contains the data to get.

`occurrence`
: The occurrence that contains the data to get.

Environment
C only

Returns
- A pointer to the field's contents.
- 0: The field cannot be found.

Description
`sm_fptr` returns the contents of the specified field. Panther strips leading or trailing blanks.

`sm_fptr` shares with several other functions a pool of buffers where it stores returned data. Consequently, you should immediately process or copy the value returned by this function.

Example
```
#include <smdefs.h>

   /* This function reports the contents of a field. */

   void report(fieldname)
   char *fieldname;
   {
      char buf[256], *stuf;
      if ((stuf = sm_n_fptr(fieldname)) == NULL)
         return;
       sprintf(buf, "Field '%s' contains '%s'",
                    fieldname, stuf);
                sm_femsg(0, buf);
            }
```

See Also `sm_getfield`, `sm_putfield`

sm_fqui_msg

Displays an error message preceded by a constant tag

```
void sm_fqui_msg(int msg_num, char *message);
```

`msg_num`
: A Panther message number. If you supply a string value for `message`, Panther ignores this parameter.

`message`
: The message to display on the status line. To use the `msg_num`-specified message, set this parameter to `NULL`.

Description

`sm_fqui_msg` is identical to sm_femsg except that it prepends a tag—for example, `ERROR:`—to the specified message. `sm_fqui_msg` gets the tag from the `SM_ERROR` entry in the message file. In GUIs, the `SM_ERROR` text is also preceded by the stop icon.

For more information on options available for this function, refer to `sm_femsg`.

See Also

`sm_femsg`, `sm_ferr_reset`, `sm_fquiet_err`

sm_fquiet_err

Displays an error message preceded by a constant tag

```
void sm_fquiet_err(int msg_num, char *message);
```

`msg_num`
: A Panther message number. If you supply a string value for `message`, Panther ignores this parameter.

`message`
: The message to display on the status line. To use the `msg_num`-specified message, set this parameter to `NULL`.

Description

`sm_fquiet_err` is identical to `sm_ferr_reset` except that it prepends a tag—for example, `ERROR:`—to the specified message. `sm_fquiet_err` gets the tag from the `SM_ERROR` entry in the message file. In GUIs, the `SM_ERROR` text is also preceded by the stop icon.

For more information on options available for this function, refer to `sm_ferr_reset`.

See Also

`sm_femsg`, `sm_ferr_reset`, `sm_fqui_msg`

sm_free_bundle

Destroys a send bundle

```
int sm_free_bundle(char *bundle_name);
```

`bundle_name`
: The name of the bundle to destroy. Supply `NULL` or empty string to specify the unnamed bundle.

Returns

0	Success.
-1	Invalid bundle name.

Description `sm_free_bundle` destroys the specified send bundle and frees the memory allocated for it.

See Also `sm_create_bundle`

sm_*ftog

Converts field references to selection group references

```
char *sm_ftog(int field_number, int *grp_occurrence);
char *sm_e_ftog(char *field_name, int element,
    int *grp_occurrence);
char *sm_i_ftog(char *field_name, int occurrence,
    int *grp_occurrence);
char *sm_n_ftog(char *field_name, int *grp_occurrence);
char *sm_o_ftog(int field_number, int occurrence,
    int *grp_occurrence);
```

`field_name`, `field_number`
: The field whose group name is sought.

`element`
: The element in `field_name` whose group name and group occurrence number is sought.

`occurrence`
: The occurrence in the specified field whose group name and group occurrence number is sought.

`grp_occurrence`
: On return, contains the group occurrence number that is currently in the specified field.

Environment C only

Returns

- A pointer to the group name if found and, through `grp_occurrence`'s output value, the group occurrence number.
- 0 otherwise and `grp_occurrence` is unchanged.

Description `sm_ftog` converts field references to group references. It returns the name of the selection group that contains the referenced field, and puts the field's group occurrence number into the address pointed to by `grp_occurrence`.

Use `sm_i_gtof` to convert selection group references back into field references.

Caution: This function returns a pointer to internal data that remains valid only for the duration of the current screen. Do not change the pointer. Doing so can yield unpredictable and possibly disruptive results.

See Also `sm_i_gtof`

sm_*fval

Forces field validation

```
int sm_fval(int field_number);
int sm_e_fval(char *array_name, int element);
int sm_i_fval(char *field_name, int occurrence);
int sm_n_fval(char *field_name);
int sm_o_fval(int field_number, int occurrence);
```

`field_name`
`field_number`
    The field to validate.

`element`
    The element in `field_name` to validate.

`occurrence`
    The occurrence in the specified field to validate.

Returns

0	The field data is valid, or the field's `no_validation` property is set to `PV_YES`.
-1	Unable to find the validation function specified for this field.
-2	The field or occurrence specification is invalid.

Description

`sm_fval` performs validation on the specified field and returns the result. This function is called automatically when the cursor exits a field whose `no_validation` property is set to `PV_NO`. A field whose `no_validation` property is set to `PV_YES` never undergoes validation processing. When called for a tab card, `sm_fval` first validates all the widgets on that card and then calls the card's validation function. To perform validation on all screen fields, call `sm_s_val`.

During field validation, Panther tests a field's data against a number of formatting and input property settings, in the order shown in Table 5-9. Some are skipped if the field is empty or its `valided` property is set to `PV_YES`–that is, there is no data to verify or the data already passed verification.

Table 5-9 Property settings and field validation

Property setting	Skip if valid	Skip if empty
`required = PV_YES`	y	n

Table 5-9 Property settings and field validation *(Continued)*

Property setting	Skip if valid	Skip if empty
`must_fill` = `PV_YES`	y	y
`regular_exp` = *`expr`*	y	y
`minimum_value` = *`value`*	y	y
`maximum_value` = *`value`*	y	y
Check Digit = *`value`**	y	y
`data_formatting` = `PV_DATE_TIME`	y	y
`table_lookup` = *`expr`*	y	y
`data_formatting` = `PV_NUMERIC`	y	y**
Validation Function*	n	n
Auto Field Function*	n	n
JPL Validation*	n	n
`calculation`	n	n

*Properties that are not accessible at runtime.
**If the field has a numeric format, the `empty_format` property also is tested.

You can force validation for an empty field by setting its `required` property to `PV_YES`. If a field has embedded characters, Panther performs validation if it contains at least one character that is neither blank nor punctuation; otherwise, it treats the field as empty. Math expressions, JPL functions and field validation functions are never skipped, because they are liable to modify other fields.

Field validation is performed automatically within `sm_input` when the cursor exits a field. `sm_s_val` validates all fields on a screen during screen exit. Applications should call this function only to force validation of other fields. Field validation is also performed when `sm_fval` or `sm_validate` is called to validate a tab card or tab deck of which the field is a member.

Example

```
// Verify that the previous field is valid before using
   // the data in the current one
```

```
proc validate(fieldnum, data, occurrence, bits)

{
      if (sm_fval(fieldnum - 1))
      {
         // Put cursor in the previous field to show error
         call sm_gofield(fieldnum - 1);
         return 1;
      }
      // otherwise process this field's data
      ...
   }
```

See Also `sm_n_gval`, `sm_s_val`, `sm_validate`

sm_*get_bi_data

Returns the specified before-image data

```
#include <tmusubs.h>
char *sm_i_get_bi_data(char *field_name, int occurrence);
char *sm_o_get_bi_data(int field_number, int occurrence);
```

`field_name`
`field_number`
: The field whose before-image data is requested.

`occurrence`
: The field's occurrence number. A negative number indicates deleted before-image data.

Environment C only

Returns

- A pointer to the before-image data.
- 0: Error.

Description `sm_get_bi_data` retrieves the before-image data for the specified field and occurrence.

sm_get_bundle_data

Reads an occurrence of bundle item data

```
char *sm_get_bundle_data(char *bundle_name, int item_no,
   int occur);
```

`bundle_name`
: The name of the bundle to read. Supply `NULL` or empty string to specify the unnamed bundle.

`item_no`
: The bundle offset of the item whose data you want to read. Each data item is identified by its offset within the bundle, where the first data item has an offset value of 1.

`occur`
: The occurrence to read from `item_no`. If the data item contains only one occurrence, supply a value of 1.

Returns

- Success: A pointer to the buffer that gets the bundle data.
- Failure: `NULL` pointer.

Description

`sm_get_bundle_data` reads an occurrence from the data item `item_no` and returns a pointer to the data's location. Each occurrence in a bundle item is a null-terminated string. If `occur` is greater than 1, `sm_get_bundle_data` traverses the bundle item until it finds the specified occurrence.

Because Panther reuses the memory location in which the bundle data is copied, you should read this data immediately after calling `sm_get_bundle_data`.

See Also

`sm_append_bundle_data`, `sm_is_bundle`

sm_get_bundle_item_count

Counts the number of data items in a bundle

```
int sm_get_bundle_item_count(char *bundle_name);
```

`bundle_name`
: The name of the bundle. Supply `NULL` or empty string to specify the unnamed bundle.

Returns

≥0	The number of items in the bundle.
-1	Invalid bundle name.

Description

`sm_get_bundle_item_count` counts the number of data items in the specified bundle. You can call this function before reading send data into a screen to ensure that a data item exists for each receiving field, or to set a counter for successive calls to `sm_get_bundle_data` or `sm_append_bundle_data` within a loop.

See Also

`sm_append_bundle_item`

sm_get_bundle_occur_count

Counts the number of occurrences in a data item

```
int sm_get_bundle_occur_count(char * bundle_name, int item_no);
```

`bundle_name`
: The name of the bundle. Supply `NULL` or empty string to specify the unnamed bundle.

`item_no`
: The bundle offset of the item whose occurrences you want to count. Each data item is identified by its offset within the bundle, where the first data item has an offset value of 1.

Returns

≥0	The number of items in the bundle.
-1	Invalid bundle name or item number.

Description

`sm_get_bundle_occur_count` counts the number of occurrences in the specified data item. Use this function to get the number of occurrences in a data item. This lets you supply the correct argument to `sm_get_bundle_data` to read the entire contents of the item into a buffer. You can also use the function's return value to set a counter for successive reads from this buffer into a target field.

Example

```
/* read data occurrences from a bundle data item
      into a field
    */
   char *occur_data, array_data;
   int num_occurs, emp_name_occur;
   emp_name_occurs = 1;

/*count the number of occurrences in the data item */
   num_occurs = sm_get_bundle_occur_count("myBundle", 1);

/*get item data and put into field*/
   for (occur = 1;occur <= num_occurs;occur++, emp_name_occur++)
       sm_i_putfield
         ("emp_names",
          emp_name_occur,
          sm_get_bundle_data("myBundle",1,occur);
```

See Also

`sm_get_bundle_data`

sm_get_next_bundle_name

Gets the name of the bundle created before the one specified

```
char *sm_get_next_bundle_name(char *bundle_name);
```

`bundle_name`
A pointer to the name of the bundle that precedes the bundle to get. Supply `NULL` or empty string to get the most recently created bundle.

Returns

- The name of the next bundle.
- An empty string if `bundle_name` does not exist or there are no more bundles.

Description

`sm_get_next_bundle_name` returns the name of the bundle whose creation preceded the one specified. Call this function iteratively inside a loop to traverse the list of all existing bundles, from youngest to oldest.

sm_*get_tv_bi_data

Gets before-image data

```
#include <tmusubs.h>
char *sm_i_get_tv_bi_data(char *field_name, int occurrence,
   char *tv);
char *sm_o_get_tv_bi_data(int field_number, int occurrence,
   char *tv);
```

`field_name`
`field_number`
: The field whose before-image data is requested.

`occurrence`
: The field's occurrence number. A negative number refers to a deleted occurrence.

`tv`
: The name of a table view in a transaction that is on the same screen as the specified field. If null or empty, the current transaction manager command determines the current table view to search for the before image.

Environment C only

Returns

- 0: Error.
- A pointer to the before-image data.
- An empty string if the specified field does not exist or is not on the same screen as an explicitly specified table view.

Description `sm_get_tv_bi_data` obtains the before-image values for the specified field and occurrence, from the point of view of the specified table view.

These routines are not to be used when previous release behavior has been set by calling `sm_tm_old_bi_context`. However, if the table view name is supplied, it will be used, despite the `sm_tm_old_bi_context` setting.

To obtain the table view containing the specified field, use *`fieldName`*`?tv` as the `tv` parameter.

See Also `sm_get_bi_data`, `sm_tm_old_bi_context`

sm_getenv

Get the value of an environment variable

```
char *sm_getenv(char *varname);
```

`varname`
    The name of the environment variable whose value is desired.

Returns The value of the environment variable named by `varname` or a null string.

Description .The environment variable table or Panther's copy of this table is searched for the variable named `varname`. If the variable is found, its value is returned, else the null string ("") is returned.

Example

```
vars oracle_home = sm_getenv("ORACLE_HOME")
```

sm_*getfield

Copies the contents of a field

```
int sm_getfield(char *buffer, int field_number);
int sm_e_getfield(char *buffer, char *field_name, int element);
int sm_i_getfield(char *buffer, char *field_name, int occurrence);
int sm_n_getfield(char *buffer, char *field_name);
int sm_o_getfield(char *buffer, int field_number, int occurrence);
```

`buffer`
: On return, contains the data copied from the specified field.

`field_name`, `field_number`
: The field to copy, where `field_name` can be the name of a field or group.

`element`
: The element to copy.

`occurrence`
: The occurrence to copy.

Environment C only

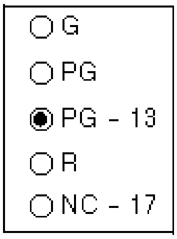
Returns

≥0	The total length of the field's contents.
-1	The field cannot be found.

Description `sm_getfield` copies data from the specified field or occurrence to `buffer`. Panther omits leading blanks from right justified fields and trailing blanks from other fields If you specify the field by name and the field is not on the screen, Panther looks for the corresponding LDB entry. If you call the function during screen entry processing, Panther first checks the LDB for an entry if `ENTEXT_OPTION` is set to `LDB_FIRST`.

Make sure that `buffer` is large enough to receive the field's contents—at least one greater than the field's maximum length.

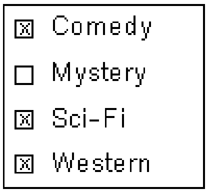
If you call `sm_n_getfield` on a radio button group that allows one selection, `buffer` gets the group occurrence number of the selected item. For example, the radio button group `rating` has the third occurrence, `PG-13`, selected:



Given this selection, the following call to `sm_n_getfield` puts the string `"3"` into `buffer`:

```
retvar = sm_n_getfield(buffer, "rating");
```

To get selections in a group that allows multiple selections—for example a group of check box widgets—issue successive calls to either `sm_i_getfield` or `sm_o_getfield`. For example, the `genre` check box group has occurrences 1, 3, and 4 selected:



Panther sees a group's value as an array whose elements contain the offsets of the selected items. Thus, Panther stores the selections in `genre` as follows:

```
genre[1] = "1"
genre[2] = "3"
genre[3] = "4"
genre[4] = " "
```

The following code gets the selections in `genre` through successive calls to `sm_i_getfield`:

```
int len, grp_occ, num_occs;
char *select_val;

/* get number of selections in group "genre" */
num_occs = sm_prop_get_int(sm_prop_id("genre"),
                           PR_NUM_OCCURRENCES);
```

```
/* get offset of selections */
for (grp_occ = 1; grp_occ <= num_occs; grp_oc++)
{
   len = sm_i_getfield(select_val, "genre", grp_occ);
   ...
}
```

Each call to `sm_i_getfield` gets the next `val` selection in the group and puts its offset into `select_val`. For instance, when `grp_occ` is 2, `sm_i_getfield` gets the second-selected item in `genre` and puts its offset value, 3 (`Sci-Fi`), into `select_val`.

Example

```
#include <smdefs.h>

   /* Save the contents of the "rank" field in a buffer
    * of the proper size. */

   int size;
   char *save_rank;

   size = sm_n_dlength("rank");
   if ((save_rank = malloc(size + 1)) == NULL)
      report_error("malloc error.");
   else
      sm_n_getfield(save_rank, "rank");
```

See Also `sm_dblval`, `sm_fptr`, `sm_intval`, `sm_lngval`, `sm_putfield`

sm_getkey

Gets the logical value of the key hit

```
#include <smkeys.h>
int sm_getkey(void);
```

Returns

- The standard ASCII value of a displayable key.
- A value greater than 255 (FF hex) for a key sequence in the key translation file.

Description

`sm_getkey` gets and interprets keyboard input and returns its logical value. Panther returns normal characters unchanged; it interprets logical keys according to the current key translation file. `sm_getkey` is called by `sm_input` and is not usually called explicitly by the application program.

Logical keys include XMIT, EXIT, HELP, arrows, data modification keys like INS, user function keys PF1 - PF24, shifted function keys SPF1 - SPF24, and others. Defined values for all are in `smkeys.h`. Some logical keys like LP and REFR are processed locally in `sm_getkey` and are not returned to the caller.

Use `sm_getkey` to retrieve logical key values previously pushed back on the input stream by `sm_ungetkey`. Because all Panther input routines call `sm_getkey`, you can use `sm_ungetkey` to generate any input sequence automatically.

When `sm_getkey` reads a key from the keyboard, it flushes the display first so the user sees a fully updated display before typing on. This is not the case for keys pushed back by `sm_ungetkey`; because input comes from the program, it is responsible for updating the display itself.

`sm_getkey` can call a number of user-installed functions. For information on installing functions, refer to "Installing Functions" on page 44-5 in *Application Development Guide*.

Like other Panther input functions, `sm_getkey` checks for externally established abort conditions on each iteration. If such a condition exists, `sm_getkey` returns the ABORT key and returns to its caller immediately. Refer to `sm_isabort`.

Note that Panther control strings are not executed within this function, but at a higher level in Panther's runtime system—that is, by functions that call `sm_getkey`.

The following outline shows how Panther processes `sm_getkey`. This presentation omits key translation for the sake of clarity; for a description of that algorithm, refer to Chapter 6, “Key Translation File,” in *Configuration Guide*.

Step 1:

- If an abort condition exists, return the ABORT key.
- If there is a key pushed back by `sm_ungetkey`, return the key.
- If playback is active and a key is available, take it directly to Step 2; otherwise read and translate input from the keyboard. When the keyboard is read and remains inactive, Panther calls the asynchronous functions, if any are installed.

Step 2:

- Pass the key to the keychange function. If that function specifies to discard the key, repeat step 1; otherwise, if an abort condition exists, return the ABORT key.
- If recording is active, pass the key to the recording function.

Step 3:

- If the routing table says to process the key locally, do so.
- If the routing table says to return the key, return it; otherwise, return to step 1.
- If the key is a soft key, return its logical value.

Example

```
#include <smdefs.h>
#include <smkeys.h>

   int query (text)
   char *text;
   {
      int key;

      sm_d_msg_line(text, REVERSE);
      for (;;)
      {
         switch (key = sm_getkey())
         {
         case XMIT:
         case 'y':
         case 'Y':
            sm_d_msg_line("", WHITE);
```

```
                  return 1;
               default:
                  sm_femsg(0, "%Mu So it's 'no'");
                  sm_d_msg_line("", WHITE);
                  return 0;
               }
            }
         }
```

See Also `sm_keyfilter`, `sm_ungetkey`

sm_*gofield

Moves the cursor into a field

```
int sm_gofield(int field_number);
int sm_e_gofield(char *field_name, int element);
int sm_i_gofield(char *field_name, int occurrence);
int sm_n_gofield(char *field_name);
int sm_o_gofield(int field_number, int occurrence);
```

`field_name, field_number`
: The destination field.

`element`
: The destination element.

`occurrence`
: The destination occurrence. If `occurrence` is offscreen, Panther scrolls it into view.

Returns

0	Success.
-1	The field is not found.

Description

`sm_gofield` puts the cursor in the first enterable position of the specified field or occurrence, according to its justification. If the field is shiftable, it is reset. If the field has embedded characters, the cursor goes to the nearest position unoccupied by a punctuation character. Use `sm_off_gofield` to put the cursor elsewhere in the field.

When called to position the cursor in a scrolling array, `sm_o_gofield` and `sm_i_gofield` return an error if the occurrence number passed exceeds by more than 1 the number of allocated occurrences in the specified array.

This function does not immediately trigger field entry, exit, or validation processing. This processing occurs according to the cursor position when control returns to `sm_input`.

If a field validation function that calls `sm_gofield` is invoked by TAB, Panther executes `sm_gofield` and moves the cursor to the specified field, then executes the TAB. To prevent this extra tab, the validation function must return non-zero. When non-zero is returned by a validation function, the field's `valided` property is set to 0 (false). In this case, reset the property to 1 (true).

Example

```
#include <smdefs.h>

   /* If the combination of this field and the previous
    * one is invalid, go back to the previous for data
    * entry. */

   int validate(field, data, occur, bits)
   int field, occur, bits;
   char *data;
   {
      if (bits & VALIDED)
         return 0;

      if (!lookup(data, sm_fptr(field - 1)))
      {
         sm_novalbit(field - 1);
         sm_gofield(field - 1);
         sm_fquiet_err(0, "Lookup failed - "
                          "please re-enter both items.");
         return 1;
      }
      return 0;
   }
```

See Also `sm_off_gofield`

sm_*gtof

Converts a selection group name and occurrence into a field number and occurrence

```
int sm_i_gtof(char *selection_group, int grp_occurrence,
    int *occurrence);
```

`selection_group`
: The name of the group whose field number is sought.

`grp_occurrence`
: The occurrence in `selection_group`.

`occurrence`
: On return, contains the occurrence number of the field.

Environment C only

Returns

≥1	The field number.
0	Cannot find the field.

Description `sm_i_gtof` converts a selection group name and occurrence into a field number and occurrence. This function lets you use other Panther library functions to manipulate selection group fields by converting group references into field references. For example, to access text from a specific field within a selection group, use `sm_i_gtof` to get the field and occurrence number, then call `sm_o_getfield` to retrieve the text.

See Also `sm_ftog`

sm_n_gval

Forces execution of a group's validation function

```
int sm_n_gval(char *group_name);
```

`group_name`
The name of the group to validate.

Returns

0	Success.
-1	The group fails any validation.
-2	The group name is invalid.

Description

`sm_n_gval` forces execution of a group's validation function. Note that since groups cannot be referenced by number, this function has only the _n_ variant.

See Also

`sm_fval`, `sm_s_val`, `sm_validate`

sm_hlp_by_name

Displays a help window

```
int sm_hlp_by_name(char *help_screen);
```

`help_screen`
The name of the help screen to display.

Returns

0	Success.
1	Success: data was copied from the help screen to the underlying field.
-1	Screen was not found or another error occurred.

Description

`sm_hlp_by_name` displays and processes the specified screen as a Panther help screen. If the help screen has a data entry field, the function copies its data back to the underlying field, as if the help screen were specified in the widget's `help_screen` property and the user pressed HELP.

Refer to Chapter 12, "Providing Help Facilities," in *Using the Editors* for information about Panther help screen creation and behavior.

sm_home

Homes the cursor

```
int sm_home(void);
```

Returns

≥1	The number of the field where the cursor is put.
0	All fields on the screen are tab-protected and the home position is not in a protected field.

Description

`sm_home` moves the cursor to the first enterable position of the first tab-accessible field on the current screen. Panther automatically calls this function when it processes the logical key HOME.

The first enterable position in a field depends on the justification of the field and, in fields with embedded characters, on the presence of punctuation. If all the screen's fields are tab-protected, `sm_home` moves the cursor to the first line and column (0,0) of the screen. If a tab-protected field occupies this position, Panther places the cursor in that field; in this case, the cursor might be invisible.

`sm_home` does not immediately trigger field entry, exit, or validation processing. Processing is based on the cursor position when control returns to `sm_input`.

See Also

`sm_backtab`, `sm_gofield`, `sm_last`, `sm_nl`, `sm_tab`

sm_inimsg

Creates a displayable error message on failure of an initialization function

```
char *sm_inimsg(int filetype, int error_code);
```

`filetype`
: Specifies the source of the error through one of the following constants, defined in `smumisc.h`:

  `B_E_KEYS`
  : Error was generated by `sm_keyinit` or `sm_n_keyinit`.

  `B_E_MSGS`
  : Error was generated by `sm_msg_read`.

  `B_E_VID`
  : Error was generated by `sm_vinit` or `sm_n_vinit`.

`error_code`
: The error code returned by the initialization function.

Environment C only

Returns

- Success: A pointer to the error message.
- Failure: Empty string.

Description `sm_inimsg` lets you display an error message to the user after an initialization function fails—for example, attempts to initialize a message file fail. You supply `sm_inimsg` with the error code returned from the failed function and a description of the function itself through parameters `error_code` and `filetype`, respectively. `sm_inimsg` uses this information to return a string that you can display to the user—for example, by passing it to `sm_fqui_msg`.

See Also `sm_keyinit`, `sm_msg_read`, `sm_vinit`

sm_*initcrt

Initializes the display and Panther data structures

```
int sm_initcrt(char *path);
void sm_jinitcrt(char *path);
void sm_jxinitcrt(char *path);
```

`path`

Specifies where to look for a library file after Panther searches the current directory. If you supply an empty string, Panther looks only in the current directory or in the paths specified by `SMPATH`. Panther searches for library files in these areas:

1. The current directory.
2. The directory specified by `path`.
3. The paths specified in the environment variable `SMPATH`.

Environment C only

Returns

- 0: Success.
- On an error, `sm_initcrt` prints a descriptive message and terminates.

Description `sm_initcrt` is called automatically by Panther. Use this function only if you write your own executive.

A custom executive should call `sm_initcrt` when screen handling starts—that is, before any screens display and the keyboard opens for screen input. `sm_initcrt` can be preceded only by those functions that set options, such as `sm_option`, and those that install functions or configuration files such as `sm_install` or `sm_vinit`.

`sm_initcrt` performs these tasks:

1. Sets a path that Panther uses to search for libraries.
2. Calls an optional user-defined initialization function. This function initializes the character string `sm_term`. If `sm_term` contains the terminal type, `sm_initcrt` proceeds to step 4.
3. Tries to ascertain the terminal type with this search algorithm:

- Looks for the variable SMTERM in the environment.
- Looks for SMTERM in SMVARS.
- Looks for the system's TERM in the environment.

If neither SMTERM or TERM are found, sm_initcrt prompts the user to supply the terminal type. If none is provided, the application terminates.

4. Finds and reads the setup file specified by SMVARS or the default configuration file smvars. If the SMSETUP variable is set, it also finds and reads this setup file.
5. Finds and reads the binary message file specified by SMMSGS. If SMMSGS cannot be found, Panther aborts initialization.
6. Finds and reads the binary video and keyboard files defined by SMVIDEO and SMKEY, respectively. These variables are defined in the SMVARS or SMSETUP setup files, or in the environment. If Panther cannot determine which files to use, it prompts for a terminal type and repeats this step.
7. Allocates memory for various data structures shared among Panther library functions.
8. Initializes the operating system's terminal channel. It is set to no echo and non-buffered input, if appropriate.
9. Initializes the operating system display.

Example

```
/* To initialize Panther without supplying a path
 * for screens:
 */

   sm_initcrt("");
```

See Also

sm_resetcrt

sm_input

Opens the keyboard for data entry and menu selection

```
int sm_input(int initial_mode);
```

`initial_mode`
    Supply `IN_AUTO`.

Returns

- The key that terminated the call to `sm_input`.
- The first character of the selected menu item.

Description

`sm_input` opens the keyboard for data entry or menu selection. This function is called automatically by Panther; use it only if you write your own executive.

`sm_input` calls `sm_getkey` to get and process keyboard entry. While in data entry mode, ASCII data can be entered into fields according to their restrictions or properties. `sm_input` returns when one of these events occurs:

- A return entry field is filled or tabbed from.
- It gets a logical key with the return bit set in the routing table.

If `sm_getkey` returns one of these logical keys—XMIT, EXIT, HELP, or a cursor position key—a routing table determines how to process it. Routing options are set by `sm_keyoption`.

See Also

`sm_getkey`, `sm_isabort`, `sm_keyoption`

sm_inquire

Gets the value of a global integer variable

```
#include <smglobs.h>
int sm_inquire(int property);
```

`property`

Specifies the global integer to get with one of the constants described in Table 5-10.

Returns

≥0	The current value of the global variable. If the variable can have a value of true or false, `sm_inquire` returns `1` for true and `0` for false.
-1	Failure.

Description

`sm_inquire` gets the integer value of the global variable specified by `property`. To modify the value of a global integer variable, use `sm_iset`.

Table 5-10 lists the constants that you can supply as arguments for `property`:

Table 5-10 Global integer variables

Constant	Meaning
`I_BSNESS`	Screen manager controls display? (true/false).
`I_INHELP`	Help level of current screen, or 0 if not in help.
`I_INERROR`	Is a message box up on the screen? (true/false)
`I_INSMODE`	In insert mode? (true/false).
`I_INXFORM`	In the screen editor? (true/false) Field validation routines are generally still called when in editor; they can check this flag to disable certain features.
`I_MXCOLMS`	Number of columns available for use by Panther on the hardware display.
`I_MXLINES`	Number of lines available for use by Panther on the hardware display.
`I_NODISP`	In non-display mode? (true/false). Initially set to false, setting this variable to true causes no further changes to the actual display, although Panther's internal screen image is kept up-to-date.

Table 5-10 Global integer variables *(Continued)*

Constant	Meaning
`I_NOMSG`	Error message display disabled? (true/false).
`I_NOWSEL`	LDB merge off for `sm_wselect`? (true/false) Normally false. True can be useful for a quick `sm_wselect`/`sm_wdeselect` pair.
`SC_AFLDMDT`	Bit mask that contains contextual information about the field's validation state and the circumstances under which a prototyped field function was called. Corresponds to the fourth standard argument passed to a non-prototyped field function.
`SC_AFLDNO`	Number of the field calling a prototyped field function. Corresponds to the first of the four standard arguments passed to a non-prototyped field function.
`SC_AFLDOCC`	Occurrence number of the field calling a prototyped field function. Corresponds to the third standard argument passed to a non-prototyped field function. The second standard argument, can be obtained from `sm_getfield` or `sm_o_getfield`.
`SC_AGRPMDT`	Bit mask that contains information about the group's validation state and the circumstances under which a prototyped group function was called. Corresponds to the second of two standard arguments passed to a non-prototyped group function. The first standard argument, a pointer to the group name, can be obtained by the `fldnum` property of a member widget and `sm_ftog` at group entry and exit. Access to the group name at group validation is not supported.
`SC_BDATTR`	Border attribute of screen.
`SC_BDCHAR`	Border character of screen.
`SC_CCOLM`	Current column number in screen (zero-based).
`SC_CLINE`	Current line number in screen (zero-based).

Example

```
if (sm_inquire(I_BSNESS))
        sm_ferr_reset(0, "Problem #2!");
    else
        fprintf(stderr,"Problem #2!\n");
```

See Also `sm_iset`, `sm_pinquire`, `sm_pset`

sm_install

Installs application event functions

```
fnc_data_r *sm_install(int func_type, fnc_data_t funcs[],
    int *num_fncs);
```

`func_type`
: Specifies the event function type. For event function types, refer to Table 44-1 on page 44-6 in *Application Development Guide*.

`funcs`
: The address of the `fnc_data` structure or array of structures to install. Functions to install with `sm_install` are stored in a `fnc_data` structure before installation. For more information about `fnc_data` structures, refer to "Preparing Functions for Installation" on page 44-4 in *Application Development Guide*.

: To deinstall functions, set `funcs` to 0. This removes all unprotected event functions of all `func_type` types.

`num_fncs`
: Supply one of these arguments:

- If an automatic function, null pointer.
- If a list of demand functions, the address of an integer whose value is the number of functions to install.

On return, this parameter points to the number of entries in the function list.

Environment C only

Returns

- When installing an automatic event with a single function, returns the address of a buffer that contains a copy of the previously installed function's data structure. If no function was previously installed, returns zero.
- When installing a function list, returns a pointer to the list.

Description `sm_install` is typically used when you build a Panther application or authoring executable. It compiles C functions and links them to Panther's function events. These C functions can be Panther library functions or functions that you write. `sm_install` can also install and deinstall functions at runtime.

The file `funclist.c`, provided in source form with Panther, can be used as a template for installing automatic and demand event functions. This file contains sample `fnc_data` structure definitions and corresponding calls to `sm_install`. Most of these calls are used to install dummy functions to the local function lists. Replace these with your own installations.

Note that in `funclist.c`, calls to `sm_install` are made by `sm_do_uinstalls`. `sm_do_uinstalls` is called after `sm_initcrt`, which calls the initialization event functions. Consequently, you should not install an initialization event function with `funclist.c`.

For specific examples of event function installation, refer to Chapter 44, “Installed Event Functions,” in *Application Development Guide*.

Example

```
/* Include the functions in fnc_data structures */

/* Install two prototyped functions that return ints,
   dereference JPL-supplied variables, and take two int
   arguments. */
   fnc_data_t proto_list[] = {
      SM_INTFNC("mark_fields(i,i)", mark_flds),
      SM_INTFNC("report(s,s)", report)
   };

/* Install a screen function that returns an int, does
   not perform dereferencing on JPL-supplied variables,
   and takes two string arguments. */
   fnc_data_t autosc_struct = SM_OLDFNC(0, auto_sfunc);

/* Install the functions */
   int ct = sizeof(proto_list) / sizeof(fnc_data_t);
   sm_install(PROTO_FUNC, proto_list, &ct);
   sm_install(DFLT_SCREEN_FUNC, &autosc_struct, (int *)0);
```

sm_*intval

Gets the integer value of a field

```
int sm_intval(int field_number);
int sm_e_intval(char *field_name, int element);
int sm_i_intval(char *field_name, int occurrence);
int sm_n_intval(char *field_name);
int sm_o_intval(int field_number, int occurrence);
```

`field_name`, `field_number`
The field whose value is sought.

`element`
The element in `field_name` whose value is sought.

`occurrence`
The occurrence in the field whose value is sought.

Environment C only

Returns

- The integer value of the specified field.

Description `sm_intval` returns the integer value of the data contained in the specified field, including its sign. All other punctuation characters are ignored. If `sm_intval` cannot find the field, it returns with 0. Because a field can contain a value of 0, you should use another method to check whether the field exists.

Example

```
/* Retrieve the integer value of the
 * "sequence" field. */

   int sequence;

   sequence = sm_n_intval("sequence");
```

See Also `sm_itofield`

sm_*ioccur

Inserts blank occurrences into an array

```
int sm_i_ioccur(char *field_name, int occurrence, int count);
int sm_o_ioccur(int field_number, int occurrence, int count);
```

`field_name`, `field_number`
: The array to receive new occurrences. In Panther 5.50 and later, `field_name` can also be a grid frame or a syncronized scrolling group.

`occurrence`
: Specifies where to insert the first occurrence in the array specified by `field_number` or `field_name`, where 0 inserts the new occurrences at the beginning of the array.

`count`
: The number of new occurrences to insert. If `count` is negative, occurrences are deleted instead, subject to the same limitations described for `sm_doccur`.

Returns

≥0	The number of occurrences actually inserted.
-1	The field or occurrence number is out of range.
-3	Insufficient memory.

Description

`sm_ioccur` inserts `count` blank occurrences before `occurrence`. If the array is scrollable, `sm_ioccur` can allocate up to `count` new occurrences. Before it inserts these, Panther checks whether the array's maximum number of occurrence is equal or greater than count plus existing data-filled occurrences:

- If true—*`max-occurs`* ≥ `count` + *`old-occurs`* —Panther inserts `count` blank occurrences before `occurrence` and pushes it and all subsequent occurrences (*`old-occurs`*) down.
- If false—*`max-occurs`* < `count` + *`old-occurs`*—Panther modifies the value of `count` to equal *`max-occurs`* - *`old-occurs`*; it then inserts as many blank occurrences as it can before `occurrence` without pushing any existing data off they array's end.

Note that `sm_ioccur` never increases the maximum number of occurrences an array can contain; you can do this by resetting the arrays' `max_occurrences` property.

Panther inserts the same number of occurrences for synchronized arrays that are unprotected from clearing. If a synchronized array is protected from clearing, Panther leaves it unchanged. Thus, you can synchronize a protected array that contains an unchanging sequence of numbers with an adjoining unprotected array whose data grows and shrinks.

`sm_o_ioccur` is normally invoked by the logical key INSL.

Example

```
#include <smdefs.h>
   /* Insert five blank lines at the beginning of
      an array named "amounts". */

   sm_i_ioccur("amounts", 0, 5);
```

See Also `sm_doccur`

sm_is_bundle

Checks whether a bundle exists

```
int sm_is_bundle(char *bundle_name);
```

`bundle_name`
: The name of the bundle to verify. Supply `NULL` or empty string to specify the unnamed bundle.

Returns

- 1 True: the bundle exists.
- 0 False: the bundle does not exist.

Description `sm_is_bundle` verifies the existence of the specified bundle.

See Also `sm_append_bundle_data`, `sm_get_bundle_data`

sm_*is_no

Tests a field for no

```
int sm_is_no(int field_number);
int sm_e_is_no(char *field_name, int element);
int sm_i_is_no(char *field_name, int occurrence);
int sm_n_is_no(char *field_name);
int sm_o_is_no(int field_number, int occurrence);
```

`field_name`, `field_number`
: The field to test.

`element`
: The element in `field_name` to test.

occurrence
: The occurrence in the field to test.

Returns

1	True: The field's first character matches the first character of the `SM_NO` entry in the message file.
0	False, or failure.

Description `sm_is_no` compares the first character of the data in the specified field or occurrence to the first letter of the `SM_NO` entry in the message file, ignoring case. A return of 0 (failure) does not indicate whether the failure occurred because the field contains the value of `SM_YES` or for another reason. To test for `SM_YES`, use `sm_is_yes`.

You can use this function with one-letter fields that specify the yes/no character edit. For these fields, users can enter only the values `SM_YES` or `SM_NO`, or space (= `SM_NO`). Unlike other functions, `sm_is_no` does not ignore leading blanks.

See Also `sm_is_yes`

sm_*is_yes

Tests a field for yes

```
int sm_is_yes(int field_number);
int sm_e_is_yes(char *field_name, int element);
int sm_i_is_yes(char *field_name, int occurrence);
int sm_n_is_yes(char *field_name);
int sm_o_is_yes(int field_number, int occurrence);
```

`field_name`, `field_number`
: The field to test.

`element`
: The element in `field_name` to test.

`occurrence`
: The occurrence in the field to test.

Returns

1 True: The field's first character matches the first character of the `SM_YES` entry in the message file.
0 False, or failure.

Description

`sm_is_yes` compares the first character of the data in the specified field or occurrence to the first letter of the `SM_YES` entry in the message file, ignoring case. A return of 0 (failure) does not indicate whether the failure occurred because the field contains the value of `SM_NO` or for another reason. To test for `SM_NO`, use `sm_is_no`.

You can use this function with one-letter fields that specify the yes/no character edit. For these fields, users can enter only the values `SM_YES` or `SM_NO`, or space (= `SM_NO`). Unlike some other functions, `sm_is_yes` does not ignore leading blanks.

See Also

`sm_is_no`

sm_isabort

Tests and sets the abort control flag

```
#include <smumisc.h>
int sm_isabort(int flag);
```

`flag`

The flag to set for abort control, one of the following defined in `smumisc.h`:

`ABT_ON`
Set abort flag.

`ABT_OFF`
Clear abort flag.

`ABT_DISABLE`
Turn abort reporting off.

`ABT_NOCHANGE`
Do not alter the flag.

Returns The previous value of the abort flag.

Description `sm_isabort` sets the abort flag to the value of `flag` and returns the old value. Abort reporting provides a quick way out of processing in the Panther library, which otherwise might involve nested calls to `sm_input`. The triggering event is the detection of an abort condition by `sm_getkey`, either an ABORT keystroke, or a call to this function with `ABT_ON`—for example, from an asynchronous function.

Example

```
#include <smdefs.h>

   /* Establish an abort condition */

   sm_isabort(ABT_ON);

   /* Verify that an abort condition exists, without
    * altering it. */

   if (sm_isabort(ABT_NOCHANGE) == ABT_ON)
      ...
```

sm_iset

Changes the value of a global integer variable

```
#include <smglobs.h>
int sm_iset (int property, int newval);
```

`property`

Specifies the global variable to change with one of these constants:

Constant	Value	Meaning
`I_INSMODE`	`0`	Enter overtype mode.
	`1`	Enter insert mode.
`I_NOWSEL`	`0`	LDB merge is on for `sm_wselect`.
	`1`	LDB merge is off for `sm_wselect`, normally set to 0. A value of 1 is useful for a quick `sm_wselect`/ `sm_wdeselect` pair, for example, to update a realtime clock.
`I_NODISP`	`0`	Enable updating of display.
	`1`	Disable updating of display, except for error messages.
`I_NOMSG`	`0`	Display error messages.
	`1`	Don't display error messages.

`newval`

The new value to assign to `property` as shown in the previous table.

Returns

≥0	Success: The previous value of `property`.
-1	Failure.

Description

Panther has a number of global parameters and settings. Use `sm_iset` to modify the current value of global integers. To get the value of a global integer, use `sm_inquire`.

If you want a process to run in the background, you can set both `I_NODISP` and `I_NOMSG` to `1`.

Example

```
void insert_mode(int on_off);
   {
      sm_iset(I_INSMODE, on_off);
   }
```

See Also

`sm_inquire`, `sm_pinquire`, `sm_pset`

sm_issv

Checks whether a screen is in the saved list

```
int sm_issv(char *screen_name);
```

`screen_name`
The name of the screen to search in the saved list.

Returns

1 True: The screen is in the saved list.
0 False.

Description

`sm_issv` searches the list of screens saved in memory for the specified screen. Call this function on screen entry to avoid redundant database queries for previously saved screens:

1. On screen exit, call `sm_svscreen` to add the screen to the save list.
2. On screen entry, call `sm_issv` to check the save list, to ascertain whether the screen has already been displayed.

Example

```
/* Perform database query only once */
   /* on the screen "results". */

   if (!sm_issv("results"))
   {
      /* do query . . .*/
      sm_svscreen(screen_list, 1);
   }
```

See Also

`sm_svscreen`

sm_*itofield

Writes an integer value to a field

```
int sm_itofield(int field_number, int value);
int sm_e_itofield(char *field_name, int element, int value);
int sm_i_itofield(char *field_name, int occurrence, int value);
int sm_n_itofield(char *field_name, int value);
int sm_o_itofield(int field_number, int occurrence, int value);
```

`field_name`, `field_number`
: The field to write.

`element`
: The element in `field_name` to write.

`occurrence`
: The occurrence in the field to write.

`value`
: The integer value to write to the field or occurrence.

Environment C only

Returns

0	Success.
-1	Failure: The field is not found.

Description `sm_itofield` converts `value` to a string and places it in the specified field. If the string is longer than the field, Panther truncates it without warning on the left or right, according to the field's justification.

Example

```
/* Find the length of the data in field number 12 */

sm_n_itofield("count", sm_dlength(12));
```

See Also `sm_intval`

sm_jclose

Closes the current window or form

```
int sm_jclose(void);
```

Returns

0	Success.
-1	No window is open—for example, the currently displayed screen is a form—or no screen is displayed.

Description

`sm_jclose` closes the active screen and restores the display to its state before the screen opened. When called for a form, `sm_jclose` pops the form stack and calls `sm_jform` to display the screen on the top of the form stack. When called for a window, `sm_jclose` calls `sm_close_window`. Panther redisplays the previous window on the window stack and puts the cursor at its last-displayed position.

Example

```
#include <smdefs.h>

   /* This is an example of a control function attached to
    * the XMIT key. It validates login and password
    * information. If the login and password are
    * incorrect, the program proceeds to close three of
    * the four "security" windows used for getting a
    * user's login and password information, and the
    * user may again attempt to enter the information.
    * If the password passes, the welcome screen is
    * displayed, and the user may proceed.
    */

   int complete_login(jptr);
   char *jptr;
   {
       char pass[10];
       sm_n_getfield(pass, "password");
       /*call routine to validate password*/
       if(!check_password(pass))
       {
           /*close current password window*/
           sm_jclose();
           /*close 3rd underlying login window*/
           sm_jclose();
           /*close 2nd underlying login window*/
           sm_jclose();
```

```
                 /*in bottom window*/
                 sm_femsg(0, "Please reenter login and password");
             }
             else
             {
                 sm_d_msg_line("Welcome to Security Systems, Inc.");
                 /*open welcome screen*/
                 sm_jform("Welcome");
             }
             return (0);
         }
```

See Also `sm_close_window`, `sm_jform`, `sm_jwindow`

sm_jfilebox

Opens a file selection dialog box

```
int sm_jfilebox(char *selection, char *path, char *file_mask,
    char *title, int open_save);
```

`selection`
: A local or global JPL variable, widget, or property to get the selected file's name.

`path`
: The initial path for the directory tree. If you supply an empty string, the dialog box initially shows the directory in which the Panther application was launched.

`file_mask`
: A filter to narrow down the display of files in `path`. Use at least one wildcard character. For example, to narrow down the display to all files that have the extension doc, supply `"*.doc"` as the argument.

To show all files, supply an empty string.

`title`
: The text of the dialog box's title. Supply an empty string to suppress title display.

`open_save`
: Valid only for Windows, determines the title of the file type option menu; ignored by other platforms. The title is platform-specific; for example, in Windows, `FB_OPEN` sets the title to List Files of Type.

Returns

1	Success: the user chose OK and Panther copied the filename to `selection`.
0	The user chose Cancel. No text is copied to `selection`.
-1	Failure: A `malloc` error occurred or `selection` was too small.

Description

`sm_jfilebox` invokes a file selection box that lets users choose a file to open or save a file. On GUI platforms, Panther uses the GUI's standard file selection dialog. The dialog box initially displays the contents of the `path`-specified directory, and lists files that match the wildcard specification in `file_mask`. Users can browse through the directory tree. When the user chooses OK, Panther copies to `selection` the name of the file to open or save.

If you are running an application on Windows, Panther uses the value of `open_save` to change the title of the file type option menu. You specify the option menu's contents through `sm_filetypes`.

Example

```
proc open_save()
   vars filename

if @widget("@current")->name == "save_button"
   {
     call sm_jfilebox \
      ("filename", "c::\\videobiz", "", "Save File", FB_SAVE)
     call save_proc(filename)
   }
   else if @widget("@current")->name == "new_button"
   {
     call sm_jfilebox \
      ("filename", "c::\\videobiz", "*.doc" "New File", FB_OPEN)
     call open_proc(filename)
   }
```

See Also `sm_filebox`, `sm_filetypes`

sm_jform

Displays a screen as a form

```
int sm_jform(char *screen_name);
```

`screen_name`
: The screen to open as a form. This character string uses the same format as a Panther control string that displays a form. This argument can optionally specify the form's position on the physical display, the size of the viewport, and which portion of the form to position in the viewport's top-left corner. For information on control string options, refer to Chapter 18, "Programming Control Strings," in *Application Development Guide*.

Returns

0	Success.
-1	The screen file's format is incorrect.
-2	The screen cannot be found.
-4	After the display cleared, the screen failed to display because of a read error.
-5	After the display cleared, the system ran out of memory.

Description

`sm_jform` displays the specified screen as a form under Panther control. If you are using your own executive, call `sm_r_form` or one of its variants to display a form. To display a window under Panther control, use `sm_jwindow`.

When `sm_jform` opens a screen as a form, Panther discards the previously displayed form and windows and frees their memory. Panther places the new form on top of the Panther form stack. You can use `sm_jclose` to close the form, or let Panther handle it—for example, when the user presses the EXIT key.

Because `sm_jform` calls `sm_r_form`, refer to `sm_r_form` for information on other details, such as how Panther finds the screen to display.

The following statement displays `myScreen`'s first row and column at row 0, column 0 of the physical display:

```
status = sm_jform("myScreen");
```

The next statement displays the screen at row 20, column 10 of the display:

```
status = sm_jform("(20,10)myScreen");
```

This statement display the screen at row 20, column 10 of the physical display in viewport that is 15 rows by 8 columns:

```
status = sm_jform("(20,10,15,8)myScreen");
```

A screen can be larger than its viewport. If the viewport does not fit on the physical display where indicated, Panther tries to place it entirely on the display at a different location. If you specify a viewport that is larger than the physical display, the viewport is the size of the physical display. To change the viewport size after the screen is displayed, set the applicable viewport properties.

Example

```
#include <smdefs.h>
    /* This could be a control function attached to the
     * XMIT key. Here we have completed entering data
     * on the second of several security screens. If
     * the user entered "bypass" into the login, he
     * bypasses the other security screens, and the
     * "welcome" screen is displayed. If the user
     * login is incorrect, the current window is
     * closed, and the user is back at the initial
     * screen (below). Otherwise, the next security
     * window is displayed. */

    int getlogin(jptr)
    char *jptr;
    {
        char password[10];
        sm_n_getfield(password, "password");
        /* check if "bypass" has been entered into login */
        if (strcmp(password, "bypass"))
            sm_jform("welcome");
            /* check if login is valid */
        else if (check_password(password))
        {
            /*close current (2nd) login window */
            sm_jclose();
            sm_femsg(NULL, "Please reenter login");
        }
        else
            sm_jwindow("login3");
        return (0);
    }
```

See Also `sm_r_form`, `sm_jwindow`

sm_*jplcall

Executes a JPL procedure

```
double sm_djplcall(char *jplcall_text);
int sm_jplcall(char *jplcall_text);
char *sm_sjplcall(char *jplcall_text);
```

`jplcall_text`
: Specifies the JPL procedure to execute, where `jplcall_text` is a string of up to 255 characters that contains the name of a JPL module or procedure and its arguments. The module or procedure must be made public with an earlier call to the JPL `public` command or to `sm_jplpublic`.

Returns

For `sm_djplcall` and `sm_jplcall`:

- The value returned by the JPL procedure.

-1 An error prevented execution of the procedure.

For `sm_sjplcall`:

- Success: A dynamically allocated string containing the value returned by the JPL procedure. When no longer needed, free this string by calling `sm_ffree`.
- Failure: Null pointer.

Description

`sm_jplcall` and its variants `sm_djplcall` and `sm_sjplcall` lets you call a JPL procedure or module from a C function. `sm_jplcall` executes a JPL procedure exactly as if the specified JPL statement were executed from within a JPL procedure. The three variants of this function differ only in their return value types.

For example, these statements in C and JPL are equivalent:

```
stat = sm_jplcall("verifysal(name, 50000)");
```

```
call verifysal(name, 50000)
```

For more information on calling JPL, refer to the `call` command.

sm_jplpublic

Executes JPL's public command

```
int sm_jplpublic(char *module_list);
```

`module_list`
: Specifies the JPL modules to load as public modules, where `module_list` is a string of up to 255 characters that contains one or more module names delimited by spaces.

Returns

0	Success.
-1	Failure.

Description

`sm_jplpublic` is the C interface to the JPL public command. Use this command to load the procedures of one or more modules into memory. Calling `sm_jplpublic` is equivalent to using the JPL `public` command. For more information, refer to the public command.

Use `sm_jplunload` to remove a module from memory.

Example

```
/* Make the error handler procedures within the file
   err_handlers available to the application */
   sm_jplpublic("err_handlers")
```

See Also

`sm_jplunload`

sm_jplunload

Executes JPL's unload command

```
int sm_jplunload(char *module_list);
```

`module_list`
: Specifies the JPL modules to unload, where `module_list` is a string that contains one or more module names delimited by spaces.

Returns

0	Success.
-1	Failure.

Description

`sm_jplunload` is the C interface to the JPL `unload` command. Use this command to remove one or more modules from memory. Modules are read into memory with `sm_jplpublic` or, in a JPL module, with the `public` command.

Calling `sm_jplunload` is equivalent to using the JPL `unload` command. For more information, refer to the `unload` command.

Example

```
void
    unload_modules()
    {
        if (sm_jplunload("select.jpl insert.jpl delete.jpl"))
            sm_ferr_reset(0,
               "Unable to unload modules from memory");
    }
```

See Also

`sm_jplpublic`

sm_jtop

Starts Panther

```
int sm_jtop(char *screen_name);
```

`screen_name`

The name of the first screen that your application displays.

Environment C only

Returns 0

Description `sm_jtop` must be called by all applications that use Panther. This function starts Panther and displays `screen_name` as a form. After the call to `sm_jtop`, Panther retains control until the user exits the application.

Panther calls various functions that handle all of the tasks required to control application flow—for example, opening the keyboard for input, opening and closing forms and windows, and processing all control strings.

If you do not use `sm_jtop`, you must write your own procedures to control application flow.

sm_jwindow

Displays a window at a given position

```
int sm_jwindow(char *screen_name);
```

`screen_name`

The screen to open as a window. `screen_name` uses the same format as a Panther control string that invokes a screen as a stacked or sibling window. Use a single ampersand (&) to specify a stacked window and a double ampersand (&&) to specify a sibling window. If no ampersand is included, the screen opens as a stacked window. The string can also specify viewport parameters.

For information on control string options, refer to Chapter 18, "Programming Control Strings," in *Application Development Guide*.

Returns

0	Success.
-1	The screen file's format is incorrect.
-2	The form cannot be found.
-3	The system ran out of memory but the previous screen was restored.

Description

`sm_jwindow` displays a screen as a window by calling `sm_r_window`. You can also call `sm_r_window` or one of its variants directly. Refer to `sm_r_window` for information on how Panther finds the screen to display.

To display a screen as a form, use `sm_jform`. To close the window programmatically, call `sm_jclose` or `sm_close_window`.

Example

```
#include <smdefs.h>

    /* This could be a control function attached to the
     * XMIT key. Here we have completed entering data
     * on the second of several security screens. If
     * the user entered "bypass" into the login, he
     * bypasses the other security screens, and the
     * "welcome" screen is displayed. If the user
     * login is incorrect, the current window is
     * closed, and the user is back at the initial
     * screen (below). Otherwise, the next security
     * window is displayed. */
```

```
int getlogin(jptr)
char *jptr;
{
    char password[10];
    sm_n_getfield(password, "password");
    /* check if "bypass" has been entered into
     * login */
    if (strcmp(password, "bypass"))
        sm_jform("welcome");
        /* check if login is valid */
    else if (check_password(password))
    {
        /*close current (2nd) login window */
        sm_jclose();
        sm_femsg(0, "Please reenter login");
    }
    else
        sm_jwindow("login3");
    return (0);
}
```

See Also `sm_jclose`, `sm_jform`, `sm_window`

sm_key_integer

Gets the integer value of a logical key mnemonic

```
#include <smkeys.h>
int sm_key_integer(char *key);
```

`key`

A logical key constant defined in `smkeys.h`. For a complete list of Panther logical keys, refer to Table 6-1 on page 6-7 in *Configuration Guide*.

Returns

≥1	The integer value of the logical key mnemonic.
-1	The mnemonic is not found.

Description

`sm_key_integer` returns the integer value of a Panther logical key constant. Panther gets this value from `smkeys.h`. This function is useful when a function needs a key's integer value but cannot access the include files.

Example

```
/* Get the integer value of the New Line/Enter key */
   int key;
   key = sm_key_integer("NL");
```

See Also

`sm_keylabel`

sm_keyfilter

Controls keystroke record/playback filtering

```
int sm_keyfilter (int flag);
```

`flag`

One of the following values:

≥1	Turn keystroke record/playback on.
0	Turn keystroke record/playback off.
<0	Return the status of keystroke record/playback.

Returns The previous value of the filter flag:

0	Recording was off.
≥1	Recording was on.

Description `sm_keyfilter` turns on or off the keystroke record/playback mechanism of `sm_getkey` according to the value of `flag`.

Example

```
/* Disable key recording and playback. */

sm_keyfilter(0);
```

See Also `sm_getkey`

sm_keyhit

Tests whether a key is typed ahead

```
int sm_keyhit(int interval);
```

`interval`
: Specifies in tenths of seconds how long to wait before it checks whether the user pressed a key. The exact length of the wait depends on the granularity of the system clock and on the hardware and operating system.

Returns

- 1 A key was typed ahead, or pressed during the `interval`-specified period.
- 0 False: no key is available.

Description

`sm_keyhit` checks whether a key has already been pressed. If a key has been pressed, it returns 1 immediately. Otherwise, it waits the specified interval. If a key is pressed during the interval the function returns 1; otherwise, it returns 0. The key, if any is struck, is not read in and is available to the usual keyboard input functions.

If the operating system does not support reads with timeout, this function ignores the interval and only returns 1 if a key has been typed ahead.

Panther calls timeout and timer functions and updates the date/time display of fields with System Update set to Yes during calls to this function.

Example

```
#include <smdefs.h>
#include <smkeys.h>

/* The following code adds one asterisk per second to
 * a danger bar, until somebody presses EXIT. */

static char *danger_bar = "*************************";
int k;

sm_d_msg_line
      ("You have 25 seconds to find the EXIT key.", WHITE);
/* Clear the danger bar area
sm_do_region(5, 10, 25, WHITE, ""); */

for (k = 1; k <= 25; ++k)
{
   sm_flush();
}
```

```
if (sm_keyhit(10))
{
   if (sm_getkey() == EXIT)
      break;
}
sm_do_region(5, 10, k, WHITE, danger_bar);

if (k <= 25)
   sm_d_msg_line("%BCongratulations! you win!");
else
   sm_ferr_reset(0, "Sorry, you lose.");
```

See Also `sm_getkey`

sm_keyinit

Initializes a key translation table

```
#include <keyfile.h>
int sm_keyinit(char *key_address);
int sm_n_keyinit(char *key_file);
```

`key_address`
: The address of a key translation table created with `key2bin` and `bin2c`; required to install a memory-resident key translation file.

`key_file`
: The name of the key translation file to use to initialize the table.

Returns

0 Success. Otherwise, Panther aborts program execution and returns to the operating system.

Description

`sm_keyinit` is called by `sm_initcrt` during initialization. You can also call it from an application program, either before or after `sm_initcrt`, to install a memory-resident key translation file.

If `sm_keyinit` fails, you can generate error messages through `sm_inimsg`. This function creates formatted output that you can display through other library functions like `sm_fqui_msg`.

sm_keylabel

Gets the printable name of a logical key

```
#include <smkeys.h>
int sm_keylabel(char *key);
```

`key`
: The logical key whose key label is sought.

Returns

- The key's name.
- Null pointer if the key has no name.

Description

`sm_keylabel` returns the label defined for `key` in the key translation file—for example, End for the XMIT key. If no label exists, the function returns the name of the logical key. Refer to Table 6-1 on page 6-7 in *Configuration Guide* for a list of Panther logical keys.

If the value of `key` is undefined in `smkeys.h`, the function returns an empty string.

Example

```
#include <smkeys.h>

    /* Put the name of the TRANSMIT key into a field
     * for help purposes. */

    char buf[80];

    sprintf(buf, "Press %s to commit the transaction.",
       sm_keylabel(XMIT));
    sm_n_putfield("help", buf);
```

sm_keyoption

Sets cursor control key options

```
#include <smkeys.h>
int sm_keyoption(int key, int mode, int newval);
```

`key`
: The key whose processing you wish to change.

`mode`
: Specifies the type of action to take on `key` with one of these values:

- `KEY_ROUTING` lets you disable a key or explicitly control the action taken when a key is pressed.
- `KEY_GROUP` lets you control the cursor action when it is within a group.
- `KEY_XLATE` lets you assign key the action performed by `newval`.

`newval`
: The new action to assign to `key`.

Returns

- The old value.
- −1: A parameter is out of range.

Description

Use `sm_keyoption` to change at runtime how `sm_input` processes `key`, where `key` is a cursor control key. Default key option values are built into Panther. This function only works with cursor control keys; these include all Panther logical keys except those of type PF, SPF, and APP. Refer to Table 6-1 on page 6-7 in *Configuration Guide* for a list of Panther logical keys.

There are three different possible values for `mode`: `KEY_ROUTING`, `KEY_GROUP` and `KEY_XLATE`. The `newval` arguments that are valid for each mode are described below. All of these modes accept a logical key constant for `key`.

`KEY_ROUTING`
: Allows access to the EXECUTE and RETURN bits of the routing table. Use this mode to disable a key or to explicitly control the action to take when a key is pressed. The following constants can be assigned to `newval`:

- `KEY_IGNORE`. Disables key. Panther does nothing when key is struck.

- `EXECUTE`. The action normally associated with key is executed; can be OR'd with `RETURN`.
- `RETURN`. No action is performed, but the function returns to the caller in your code. Use to gain direct control of key's action; can be OR'd with `EXECUTE`.

`KEY_GROUP`

Allows access to the group action bits. Use this mode to control the action of the cursor when it is within a group. The following values can be assigned to `newval`:

- `VF_GROUP` — Obey group semantics. Hitting key causes the cursor to move to the next field within the group in the indicated direction. If this constant is OR'd with `VF_CHANGE` the cursor exits the group in the indicated direction.
- `VF_CHANGE` — This value has no effect, unless it is OR'd with `VF_GROUP`. In this case the cursor exits the group in the indicated direction.
- `0` — Assigning zero to `newval` causes `key` to treat a field within a group as if it were not part of a group.
- `VF_OFFSCREEN` — Offscreen data scrolls onscreen from the direction indicated.
- `VF_NOPROT.key` — Moves cursor into a field protected from tabbing.

`KEY_XLATE`

Allows access to the cursor table. Use this mode to assign `key` the action performed by `newval`. `key` can be any cursor control key excluding `INS`, `MNBR`, `REFR`, `SFTS`, and `LP`. `newval` can be any key—logical, function, application, ASCII, and so on.

Example

```
/* newline_is_xmit: Map the new line key (Return or Enter on
    * most keyboards) to XMIT -or- reset it back to NL.
    * Invoke from a control string as:
    * ^newline_is_xmit X    To make NL act as XMIT
    * ^newline_is_xmit N    To make NL act as NL                */


int newline_is_xmit(char *cs_data);
{
   while (*cs_data && *cs_data != ' ')
      cs_data++;
    while (*cs_data == ' ')
```

```
            cs_data++;
        if (*cs_data == 'X')
        {
            sm_keyoption(NL, KEY_XLATE, XMIT);
        }
        else
        {
            sm_keyoption(NL, KEY_XLATE, NL);
        }
        return(0);
    }
```

sm_l_close

Closes a library and frees all memory associated with it

```
int sm_l_close(int lib_desc);
```

`lib_desc`
: The library to close, where `lib_desc` is an integer library descriptor returned by `sm_l_open`.

Returns

0	Success.
-1	Operating system reported an error closing the library.
-2	The library is already closed.

Example

```
/* Bring up a window from a library. */

    int ld;

    if ((ld = sm_l_open ("myforms")) < 0)
    sm_cancel();
    ...
    sm_l_at_cur(ld, "popup");
    ...
    sm_l_close(ld);
```

See Also `sm_l_at_cur`, `sm_l_form`, `sm_l_open`, `sm_l_window`

sm_l_open

Opens a library

```
int sm_l_open(char *lib_name);
```

`lib_name`
: The name of the library to open. Panther searches for `lib_name` in the current directory, then along the path given to `sm_initcrt`, and finally along the path defined by `SMPATH`.

Returns

≥1	The library file's identifier.
-1	The library cannot be opened or read.
-3	The named file is not a library.
-4	Insufficient memory is available.

Description

Use `sm_l_open` to open a library before you use a JPL module, a menu, or a screen that is in that library. `sm_l_open` opens a library in these steps:

- Allocates space in which to store information about the library.
- Leaves the library file open, and returns a descriptor that identifies the library. You can use this descriptor to explicitly search a single library—for example, to find a screen in a specific library with `sm_l_window`.

If you define the `SMFLIBS` variable in your setup file as a list of library names, Panther automatically calls `sm_l_open` for those libraries.

Panther has no limit on the number of libraries you can have open at the same time. Note that some systems have severe limits on memory or simultaneously open files.

Example

```
/* Prompt for the name of a library until a
    * valid one is found. Assume the memory-resident
    * screen contains one field for entering the library
    * name, with suitable instructions. */

int ld;
extern char libquery[];

if (sm_d_form(libquery) < 0)
    sm_cancel();
sm_d_msg_line("Please enter the name of your library.");
```

```
do {
    sm_input(IN_DATA);
} while ((ld = sm_l_open(sm_fptr (1))) < 0);
```

See Also `sm_form`, `sm_jplcall`, `sm_jplpublic`, `sm_l_close`, `sm_window`

sm_l_open_syslib

Opens a system library

```
int sm_l_open_syslib(char *lib_name);
```

`lib_name`
: The name of the library to open. Panther searches for `lib_name` in the current directory, then along the path given to `sm_initcrt`, and finally along the path defined by `SMPATH`.

Returns

≥1	The library file's identifier.
-1	The library cannot be opened or read.
-3	The named file is not a library.
-4	Insufficient memory is available.

Description

Use `sm_l_open_syslib` to open a library as a system library. The library name will not be displayed in the Library Table of Contents window. Otherwise, this function performs the same steps as `sm_l_open` which opens a library in these steps:

- Allocates space in which to store information about the library.
- Leaves the library file open, and returns a descriptor that identifies the library. You can use this descriptor to explicitly search a single library—for example, to find a screen in a specific library with `sm_l_window`.

Panther has no limit on the number of libraries you can have open at the same time. Note that some systems have severe limits on memory or simultaneously open files.

See Also

`sm_l_open`

sm_last

Positions the cursor in the last field

```
void sm_last(void);
```

Description `sm_last` places the cursor at the first enterable position of the last tab-accessible field of the current screen. The first enterable position depends on the justification of the field and, in fields with embedded punctuation, on the presence of punctuation.

Unlike `sm_home`, `sm_last` does not reposition the cursor if all fields are tab-protected.

This function does not immediately trigger field entry, exit, or validation processing. Such processing depends on the cursor position when control returns to `sm_input`.

This function is called when the Panther logical key EMOH is struck.

See Also `sm_backtab`, `sm_home`, `sm_nl`, `sm_tab`

sm_launch

Invokes a process without waiting for it to return

```
int sm_launch(char *command);
```

`command`
The command to be launched

Returns

- Success: On Windows, 0; on POSIX, the process ID of the new process.
- Failure: On Windows, -1.

Description `sm_launch` starts the command running, and does not wait for it to terminate.

See Also `sm_shell`

sm_*ldb_fld_*get

Copies data from LDBs to specific fields on the current screen

```
int sm_h_ldb_fld_get(int respect_flag, int ldb, int field_number);
int sm_n_ldb_fld_get(int respect_flag, char *ldbname,
   int field_number);
int sm_h_ldb_n_fld_get(int respect_flag, int ldb,
   char *field_name);
int sm_n_ldb_n_fld_get (intrespect_flag, char *ldbname,
   char *field_name);
```

`respect_flag`
: Indicates whether to write to fields that already contain data:

0	Initialize all fields, regardless of their status.
≥1	Initialize only empty or unmodified fields.

`ldb`
: Handle of LDB from which to get data.

`ldbname`
: Name of LDB from which to get data. Use NULL or "" (the empty string) to search through all LDBs for the one that matches the field.

`field_number, field_name`
: Field to write to on the current screen.

Returns

0	Success.
-1	Invalid field specifier.
-2	LDB entry not found.
-4	Invalid LDB specifier.

Description `sm_ldb_fld_get` copies data from specific local LDB blocks loaded into memory to specific fields on the current screen. This function has the following variants:

- `sm_h_ldb_fld_get` specifies the LDB by its handle. The LDB data is copied to the field identified by number in the `field_number` argument.

- `sm_n_ldb_fld_get` specifies the LDB by its name. The LDB data is copied to the field identified by number in the `field_number` argument. You can specify ldb as NULL or "" (the empty string) to search through all LDBs for the one that matches `field_number`.
- `sm_h_ldb_n_fld_get` specifies the LDB by its handle, and the field to be written to by its name in the `field_name` argument.
- `sm_n_ldb_n_fld_get` specifies the LDB by its name, and the field to be written to by its name in the `field_name` argument. You can specify `ldb` as `NULL` or "" (the empty string) to search through all active LDBs for the one that matches `field_name`.

See Also `sm_ldb_fld_store`, `sm_ldb_getfield`, `sm_ldb_name`, `sm_ldb_handle`

sm_*ldb_fld_*store

Copies data from specific fields to LDBs

```
int sm_h_ldb_fld_store(int ldb, int field_number);
int sm_n_ldb_fld_store(char *ldbname, int field_number);
int sm_h_ldb_n_fld_store(int ldb, char *field_name);
int sm_n_ldb_n_fld_store(char *ldbname, char *field_name);
```

`ldb`
: Handle of the LDB to write to.

`ldbname`
: Name of the LDB to write to.

`field_number, field_name`
: Field from which to get data.

Returns

0	Success.
-1	Invalid field specifier.
-2	LDB entry not found.
-3	Insufficient memory failure.
-4	Invalid LDB specifier.

Description

`sm_ldb_fld_store` copies data from the specified fields on the current screen to the (possibly) specified LDBs. This function has the following variants:

- `sm_h_ldb_fld_store` specifies the LDB by its handle. The field data is copied from the field identified by its number in the `field_number` argument.
- `sm_n_ldb_fld_get` specifies the LDB by its name. The field data is copied from the field identified by its number in the `field_number` argument. You can specify `ldb` as `NULL` or "" (the empty string) to search through all active LDBs for the one that matches `field_number`.
- `sm_h_ldb_n_fld_get` specifies the LDB by its handle, and the field to be copied from by its name in the `field_name` argument.
- `sm_n_ldb_n_fld_get` specifies the LDB by its name, and the field to be copied from by its name in the `field_name` argument. You can specify `ldb` as

`NULL` or "" (the empty string) to search through all active LDBs for the one that matches `field_name`.

See Also `sm_ldb_fld_get`, `sm_ldb_name`, `sm_ldb_handle`

sm_ldb_get_active

Gets the handle of the most recently loaded active LDB

```
int sm_ldb_get_active(void);
```

Returns

≥0	Success: The integer handle of the most recently activated LDB.
-1	Failure: No LDBs are active.

Description

`sm_ldb_get_active` searches the stack of loaded LDBs and returns the integer handle of the topmost LDB that is marked as active. If multiple LDBs are active, the most recently loaded one always has precedence during LDB write-through. Use this function together with `sm_ldb_get_next_active` to iterate over all active LDBs in order of most to least recently loaded. For example:

```
int h;
   for (
      h = sm_ldb_get_active();
      h != -1;
      h = sm_ldb_get_next_active() )
   {
   /* Do stuff with h */
   }
```

Note: The order in which LDBs are activated can be different from the order in which they were loaded.

See Also

`sm_ldb_get_next_active`

sm_ldb_get_inactive

Gets the handle of the most recently loaded inactive LDB

```
int sm_lbd_get_inactive(void);
```

Returns

≥0	Success: The integer handle of the most recently inactivated LDB.
-1	Failure: No LDBs are inactive.

Description

`sm_ldb_get_inactive` searches the stack of loaded LDBs and returns the integer handle of the topmost LDB that is also inactive. Use this function together with `sm_ldb_get_next_inactive` to iterate over all inactive LDBs in order of most to least recently loaded. For example:

```
int h;
   for (
        h = sm_ldb_get_inactive();
        h != -1;
        h = sm_ldb_get_next_inactive(h) )
   {
      /* Do stuff with h */
   }
```

See Also

`sm_ldb_get_next_inactive`

sm_ldb_get_next_active

Gets the active LDB loaded before the one specified

```
int sm_ldb_get_next_active(int prev_handle);
```

`prev_handle`
The handle of an active LDB.

Returns

≥0	Success: The handle of an activated LDB.
-1	No LDB was active before `prev_handle`.
-2	`prev_handle` is invalid.

Description

`sm_ldb_get_next_active` takes the handle of an active LDB and returns with the handle of the LDB that was most recently loaded before it and is also active. Use this function together with `sm_ldb_get_active` to iterate over all active LDBs in order of most to least recently loaded. For example:

```
int h;
   for (
        h = sm_ldb_get_active();
        h != -1;
        h = sm_ldb_get_next_active(h) )
   {
      /* Do stuff with h */
   }
```

Note: The order in which LDBs are activated can be different from the order in which they were loaded.

See Also

`sm_ldb_get_active`

sm_ldb_get_next_inactive

Gets the inactive LDB loaded before the one specified

```
int sm_ldb_get_next_inactive(int prev_handle);
```

`prev_handle`
The handle of an inactive LDB.

Returns

≥0	Success: The handle of an inactivated LDB.
-1	No LDB was inactivated before `prev_handle`.
-2	`prev_handle` is invalid.

Description

`sm_ldb_get_next_inactive` takes the handle of an inactive LDB and returns with the handle of the LDB most recently loaded before it that is also inactive. Use this function together with `sm_ldb_get_inactive` to iterate over all inactive LDBs in order of most to least recently loaded. For example:

```
int h;
   for (
        h = sm_ldb_get_inactive();
        h != -1;
        h = sm_ldb_get_next_inactive(h) )
   {
      /* Do stuff with h */
   }
```

See Also

`sm_ldb_get_inactive`

sm_*ldb_*getfield

Gets the contents of an LDB entry

```
int sm_ldb_getfield (char *buffer, int field_number,
    char *ldbname);
int sm_i_ldb_getfield (char *buffer, char *field_name,
    int occurrence, char *ldbname);
int sm_n_ldb_getfield (char *buffer, char *field_name,
    char *ldbname);
int sm_o_ldb_getfield (char *buffer, int field_number,
    int occurrence, char *ldbname);
int sm_ldb_h_getfield (char *buffer, int field_number,
    int ldbhandle);
int sm_i_ldb_h_getfield (char *buffer, char *field_name,
    int occurrence, int ldbhandle);
int sm_n_ldb_h_getfield (char *buffer, char *field_name,
    int ldbhandle);
int sm_o_ldb_h_getfield (char *buffer, int field_number,
    int occurrence, int ldbhandle);
```

`buffer`
: The buffer to get the LDB data.

`field_name`, `field_number`
: The LDB field with the data to obtain.

`occurrence`
: The occurrence that contains the data to obtain.

`ldbname`
: The name of the LDB that contains the field.

`ldbhandle`
: The handle of the LDB that contains the field.

Environment C only

Returns

≥0	The length of the data in the LDB entry.
-1	Unable to find the specified field.
-2	Unable to find the specified LDB.
-3	The occurrence number is out of range.

Description `sm_ldb_getfield` gets the contents of an entry or array occurrence in the specified LDB. This function is not callable from JPL code. This function and its variants let you specify an LDB by name or by handle. The LDB must be among one of the LDBs loaded into memory. If multiple instances of the same LDB are loaded, you can get the value from the desired instance by specifying its handle; if you specify the LDB by name, Panther gets the value from the last-loaded instance.

sm_ldb_handle

Gets the handle of an LDB

```
int sm_ldb_handle(char *ldbname);
```

`ldbname`

The name of the LDB to get.

Returns

≥0 Success: The handle of `ldbname`.

-1 Failure: Cannot find `ldbname` among the loaded LDBs.

Description

`sm_ldb_handle` takes the name of an LDB and returns with its integer handle of the specified LDB. The LDB can be active or inactive; however, it must be loaded into memory.

sm_ldb_init

Initializes or reinitializes local data blocks

```
void sm_ldb_init(void);
```

Environment C only

Description `sm_ldb_init` unloads all LDBs from memory, whether active or not. It then loads and activates the same LDBs as at application startup. At application startup, Panther calls this function and attempts to load and activate LDBs as follows:

1. Looks for the configuration variable `SMLDBLIBNAME` and opens all screens in the specified libraries as LDBs. The default value for this variable is `ldb.lib`.
2. Looks for the configuration variable `SMLDBNAME` and opens the specified screens as LDBs. For example:

   ```
   SMLDBNAME = screen1.scr screen2.scr screen3.scr
   ```

   The default value for this variable is `ldb.scr`.

sm_ldb_is_loaded

Tests whether an LDB is loaded

```
int sm_ldb_is_loaded(char *ldbname);
```

`ldbname`
The name of the LDB to test.

Returns

0	The LDB is not loaded.
1	The LDB is loaded.

Description

`sm_ldb_is_loaded` takes the name of an LDB and tests whether it is loaded into memory or not. It returns a value of true (1) or false (0).

sm_ldb_load

Loads an LDB into memory

```
int sm_ldb_load(char *ldbname);
```

`ldbname`
The name of the LDB to load.

Returns

≥0	The handle of the loaded LDB.
-1	Failure. Panther was unable to load the LDB for one of these reasons: - Unable to open the specified file. - Unable to read the file. - The file type is invalid.

Description

`sm_ldb_load` loads a screen into memory as an LDB. Multiple LDBs can be loaded into memory; of these, one or more can be active at any time. Once an LDB is loaded, you can activate it by calling `sm_ldb_state_set`; only active LDBs are open to read and write operations.

You can load multiple instances of the same LDB. For example, you might do this to prevent data from multiple invocations of the same screen from overwriting each other. Because Panther assigns a unique handle to each loaded LDB, you can reference these LDBs either collectively by their common name, or individually by their separate handles.

See Also

`sm_ldb_state_set`, `sm_ldb_unload`

sm_ldb_name

Gets the name of an LDB of the specified handle

```
char *sm_ldb_name(int ldbhandle);
```

`ldbhandle`
    The handle of the LDB to look up.

Returns

- Success: A pointer to the name of the LDB specified by `ldbhandle`.
- Failure: Null pointer.

Description

`sm_ldb_name` takes the integer handle of an LDB and returns a pointer to the LDB's name.

sm_ldb_next_handle

Gets the handle of a previously loaded LDB with the same name as the specified LDB

```
int sm_ldb_next_handle(int ldbhandle);
```

`ldbhandle`
: The handle of a loaded LDB whose name is sought among previously loaded LDBs.

Returns

≥0	Success: The handle of a previously loaded LDB with the same name as `ldbhandle`.
-1	No LDB was loaded before `ldbhandle`.
-2	`ldbhandle` is not a valid handle.

Description

`sm_ldb_next_handle` takes a handle of a loaded LDB and looks for a previously loaded instance of the same LDB. If an earlier instance exists, the function returns with its handle. You can call this function iteratively to ascertain how many instances of an LDB are loaded into memory and their order of precedence.

sm_ldb_pop

Pops LDBs off the LDB save stack

```
void int sm_ldb_pop(void);
```

Returns

0	Success.
-1	The stack is empty.

Description `sm_ldb_pop` removes all loaded LDBs from memory. It then restores to memory the LDBs in the LDB save stack’s topmost—that is, most recently pushed—list. If any LDBs were active at the time they were unloaded, `sm_ldb_pop` restores them to active status. If the stack is empty, `sm_ldb_pop` removes all loaded LDBs from memory and returns with `-1`.

See Also `sm_ldb_push`

sm_ldb_push

Pushes all LDBs onto a save stack

```
void int sm_ldb_push(void);
```

Returns	

0	Success: one or more LDBs are pushed.
-1	No LDBs are currently loaded.
-2	A memory allocation error occurred.

Description

`sm_ldb_push` makes all loaded LDBs unavailable to the application. It writes their identities and status—whether active or not—to a list that it pushes onto the LDB save stack. Each call to `sm_ldb_push` pushes another list of LDBs onto the stack; the stack stores these lists in first-in/last-out order. The number of lists you can save depends on the amount of memory available on your system. To restore the last-pushed list of LDB's to memory, call `sm_ldb_pop`.

See Also

`sm_ldb_pop`

sm_*ldb_*putfield

Reads data into an LDB entry

```
int sm_ldb_putfield (int field_number, char *ldbname,
    char *buffer);
int sm_i_ldb_putfield (char *field_name, int occurrence,
    char *ldbname, char *buffer);
int sm_n_ldb_putfield (char *field_name, char *ldbname,
    char *buffer);
int sm_o_ldb_putfield (int field_number, int occurrence,
    char *ldbname, char *buffer);
int sm_ldb_h_putfield (int field_number, int ldbhandle,
    char *buffer);
int sm_i_ldb_h_putfield (char *field_name, int occurrence,
    int ldbhandle, char *buffer);
int sm_n_ldb_h_putfield (char *field_name, int ldbhandle,
    char *buffer);
int sm_o_ldb_h_putfield (int field_number, int occurrence,
    int ldbhandle, char *buffer);
```

`field_name, field_number`
: The LDB field to read the data in `buffer`.

`occurrence`
: The occurrence to read the data.

`ldbname`
: The name of the LDB that contains the field.

`ldbhandle`
: The handle of the LDB that contains the field.

`buffer`
: The buffer that contains the data to read.

Returns

0	Success.
-1	Unable to find the specified field.
-2	Unable to find the specified LDB.
-3	The occurrence number is out of range.

Description

`sm_ldb_putfield` reads the contents of the specified buffer into an entry or array occurrence in the specified LDB. This function and its variants let you specify an LDB by name or by handle. The LDB must be among one of the LDBs loaded into memory. If multiple instances of the same LDB are loaded, you can get the value from the desired instance by specifying its handle; if you specify the LDB by name, Panther gets the value from the last-loaded instance.

sm_ldb_*state_get

Gets the current state of the LDB

```
int sm_ldb_state_get(char *ldbname, int state_type);
int sm_ldb_h_state_get(int ldbhandle, int state_type);
```

`ldbname`
: The name of the LDB whose state you want to get.

`ldbhandle`
: The integer handle of the LDB whose state you want to get.

`state_type`
: Specifies the state to get with one of these constants:

  `LDB_ACTIVE`
  : A Yes/No flag that specifies whether the LDB is active. Only active LDBs participate in LDB write-through.

  `LDB_READ_ONLY`
  : A Yes/No flag that specifies whether the LDB is read-only. Screens can read from this LDB on screen entry but cannot modify it on exit; consequently, a read-only LDB cannot be used to transfer values from one screen to another.

Returns

0	`state_type` is set to No.
1	`state_type` is set to Yes.
-1	Unable to find `ldbname`.

Description

`sm_ldb_state_get` lets you determine whether a loaded LDB is active or whether it is read-only. Call this function before changing an LDB's state through `sm_ldb_state_set`.

sm_ldb_*state_set

Changes the state of the LDB

```
int sm_ldb_state_set (char *ldbname, int state_type,
    int new_value);
int sm_ldb_h_state_set (int ldbhandle, int state_type,
    int new_value);
```

`ldbname`
: The name of the LDB whose state you want to set.

`ldbhandle`
: The integer handle of the LDB whose state you want to set.

`state_type`
: Specifies the state to set with one of these constants:

  `LDB_ACTIVE`
  : A Yes/No flag that specifies whether the LDB is active. Only active LDBs participate in LDB write-through.

  `LDB_READ_ONLY`
  : A Yes/No flag that specifies whether the LDB is read-only. The default for newly activated LDBs is set to No. Screens can read from this LDB on screen entry but cannot modify it on exit; consequently, a read-only LDB cannot be used to transfer values from one screen to another.

`new_value`
: A value of 1 (Yes) or 0 (No) to set for `state_type`.

Returns

0	Success.
1	No change: the LDB was already set to the specified state.
-1	Unable to find `ldbname`.

Description `sm_ldb_state_set` lets you change the status of an LDB in one of two ways:

- Allow or disallow participation in LDB write-through. If a loaded LDB has its active state (`LDB_ACTIVE`) set to Yes, screens can, at a minimum, read its data; if the LDB's `LDB_READ_ONLY` state is set to No, screens can also write data to it. For more information about LDB write-through, refer to "Using Local Data Blocks" on page 25-7 in *Application Development Guide*.

- Set the LDB data to be read-only. If an active LDB is read-only—`LDB_READ_ONLY` is set to Yes—a screen can read that LDB's data but cannot use it to propagate data to other screens. By default, newly activated LDBs have `LDB_READ_ONLY` set to No.

Note: You can call `sm_ldb_state_set` only on LDBs that are already loaded into memory. To load an LDB at runtime, call `sm_ldb_load`.

See Also `sm_ldb_load`, `sm_ldb_state_get`

sm_ldb_*unload

Unloads LDBs from memory

```
int sm_ldb_unload(char *ldbname);
int sm_ldb_h_unload(int ldbhandle);
```

`ldbname`
: The name of the LDB to unload.

`ldbhandle`
: The integer handle of the LDB to unload.

Returns

0	Success.
-1	Failure. Panther is unable to find the specified LDB.

Description

`sm_ldb_unload` unloads LDBs and free the memory allocated for it. If the LDB is loaded more than once, use `sm_ldb_unload` to unload all instances; to unload a specific instance, supply its handle with `sm_ldb_h_unload`.

sm_leave

Prepares to temporarily leave a Panther application

```
void sm_leave(void);
```

Environment C only

Description `sm_leave` lets you leave a Panther application temporarily—for example, to escape to the command interpreter or execute some graphics functions. When you call this function before leaving, `sm_leave` performs these tasks:

- Clears the physical screen, but not the internal screen image.
- Resets the operating system channel.
- Resets the terminal with the `RESET` sequence found in the video file.

Example

```
#include <smdefs.h>

   /* Escape to the UNIX shell for a directory listing */

   sm_leave();
   sm_system("ls -l");
   sm_return();
   sm_c_off();
   sm_d_msg_line("Hit any key to continue", BLINK | WHITE);
   sm_getkey();
   sm_d_msg_line("", WHITE);
   sm_rescreen();
```

See Also `sm_return`

sm_list_objects_count

Counts the widgets contained by an application object

```
int sm_list_objects_count(int list_id);
```

`list_id`
: An integer handle to the list of widgets in an application object, obtained from `sm_list_objects_start`.

Returns

- ≥0 The number of objects listed for the specified container.
- `PR_E_OBJID`: Unable to find the specified list handle.

Description

`sm_list_objects_count` returns the number of objects specified in an object contents list. This list, created by `sm_list_objects_start`, initially contains the object IDs of all widgets in the container object; thus, a call to `sm_list_objects_count` immediately after the list is created yields the total number of widgets in a container object. Each call to `sm_list_objects_next` reduces by one the number of objects in the list; so a call to `sm_list_objects_count` that is preceded by calls to `sm_list_objects_next` yields the number of objects that remain on the object contents list.

Note: `sm_list_objects_count` does not check whether the widgets in an object contents list are still in existence; it is therefore possible to return a count that includes invalid object IDs for widgets that were destroyed after the list's creation.

See Also

`sm_list_objects_end`, `sm_list_objects_start`

sm_list_objects_end

Destroys an object contents list

```
void sm_list_objects_end(int list_id);
```

`list_id`
: An integer handle to the object contents list to destroy, obtained from `sm_list_objects_start`.

Description `sm_list_objects_end` destroys an object contents list created by `sm_list_objects_start` and deallocates the memory associated with it. All subsequent attempts to access this list yield an error (`PR_E_OBJID`). Always pair this function with `sm_list_objects_start`.

See Also `sm_list_objects_start`

sm_list_objects_next

Traverses the widgets contained by an application object

```
int sm_list_objects_next(int list_id);
```

`list_id`
An integer handle to the list of widgets in an application object, obtained from `sm_list_objects_start`.

Returns

≥1 The object ID of the next widget listed for the specified container object.

- `PR_E_ERROR`: The list is empty.
- `PR_E_OBJID`: Unable to find the specified list handle.

Description

`sm_list_objects_next` returns a handle to the next widget in the object contents list created by `sm_list_objects_start`. When this list is created, it contains the object IDs of all widgets within the container object. The first call to `sm_list_objects_next` on a given list returns the object ID of the first widget on the list; each subsequent call returns the object ID of the next-listed widget; it also removes the last-returned object ID and thereby reduces the number of listed objects by one.

When the list is completely traversed, the function returns `PR_E_ERROR`. You can use this error code to test whether a list is fully traversed; or use `sm_list_objects_count` to set a counter for traversing the list.

For example, the following code creates an objects contents list for all members in a grid and traverses the list:

```
proc traverse_grid(grid_name)
vars grid_list, ct, member_ct, member_id

// create list of all members in grid
grid_list = sm_list_objects_start(sm_prop_id(grid_name))

if grid_list > 0
{
    // get count of listed object IDs
    member_ct = sm_list_objects_count(grid_list)

    for ct = 1 while ct <= member_ct
    // traverse list
    {
        member_id = sm_list_objects_next(grid_list)
        // use member's object ID to perform some action on it
    }
    call sm_list_objects_end(grid_list)
    return 1
}
return 0
```

`sm_list_objects_next` does not check whether the widgets identified in an object contents list are still in existence; it is therefore possible to return invalid object IDs for widgets that were destroyed after the list's creation.

When you are finished traversing an object contents list, call `sm_list_objects_end` on the list to destroy it and deallocate its memory.

See Also `sm_list_objects_count`, `sm_list_objects_end`, `sm_list_objects_start`

sm_list_objects_start

Constructs a list of widgets contained by an application object

```
int sm_list_objects_start(int obj_id);
```

`obj_id`
: An integer handle that identifies the container object whose contents are to be listed, obtained through `sm_prop_id` or through the JPL `id` property.

Returns

≥1 A handle to the object contents list.

- `PR_E_OBJID`: The container object does not exist.
- `PR_E_MALLOC`: Memory allocation error occurred.

Description

`sm_list_objects_start` creates a list of all widgets that are currently contained by the specified object; the list identifies widgets by their object IDs. The function returns a handle to the list so you access its contents. The container object can be a screen (including one used as an LDB), grid widget, box widget, selection group, synchronized scrolling group, or table view widget.

All widgets within the container object are included in the list, even if they are themselves contained by other widgets. Two exceptions apply: the list that is generated for a box and or grid widget excludes any selection groups and synchronized scrolling groups that are inside the container object.

Widgets that accept data are listed in order of their position within the container object—from left to right, top to bottom. You can traverse an object contents list and thereby access the widgets in it by calling `sm_list_objects_next`; you can also count the listed objects with `sm_list_objects_count`. For an example, refer to `sm_list_objects_next`.

An ID to an object contents list is always returned, even when the list is empty. When you are done examining the list, be sure to free the memory allocated for it by calling `sm_list_objects_end`.

See Also

`sm_list_objects_count`, `sm_list_objects_end`, `sm_list_objects_next`, `sm_prop_id`

sm_*lngval

Gets the long integer value of a field

```
long sm_lngval(int field_number);
long sm_e_lngval(char *field_name, int element);
long sm_i_lngval(char *field_name, int occurrence);
long sm_n_lngval(char *field_name);
long sm_o_lngval(int field_number, int occurrence);
```

`field_name`, `field_number`
: The field whose value is sought.

`element`
: The element in `field_name` that contains the data to get.

`occurrence`
: The occurrence in the specified field that contains the data to get.

Environment
: C only

Returns
: • The long value of the field.

Description
: `sm_lngval` returns the contents of the specified field as a long integer. It recognizes only digit characters and a leading plus or minus sign.

Example

```
#include <smdefs.h>

   /* Retrieve the number of fish in one particular sea
    * (a big number) from the screen. */

   #define MEDITERRANEAN 4
   long fish;

   fish = sm_e_lngval("seas", MEDITERRANEAN);
```

See Also
: `sm_intval`, `sm_ltofield`

sm_load_screen

Preloads a screen into memory

```
int sm_load_screen(char *screen_name);
```

`screen_name`
: The name of the screen.

Returns

0	Success.
-1	Screen file's format is incorrect.
-2	Screen cannot be found.
-3	Insufficient memory available to load the screen.

Description

`sm_load_screen` loads a screen into memory without displaying it. Use this function to provide fast response times when the screen is displayed later. This function calls `sm_svscreen`, but it improves efficiency by also processing any GUI extensions necessary for display. Also, unlike `sm_svscreen,` it can be called from JPL.

See Also

`sm_svscreen`, `sm_unload_screen`, `sm_window`

sm_log

Writes a message to an error log

```
int sm_log(char *msg);
```

`msg`

Message to be printed to the log.

Scope Server

Description `sm_log` writes messages to an error log file.

For COM and EJB applications, a file named `server.log` must exist in the component's application directory. When this file is created, in addition to the messages logged with this function, messages are automatically logged when service components are created or destroyed. All messages that would normally appear on the message line or message window are also logged.

During development you should always enable error logging by creating `server.log`. In production `server.log` should not be present as logging is a substantial load on the system.

sm_lstore

Copies everything from screen to LDB

```
int sm_lstore(void);
```

Returns

0	Success.
-3	Insufficient memory.

Description

`sm_lstore` copies data from the screen to local data block entries with matching names.

Panther automatically calls `sm_lstore` when it brings up a new screen or before it closes a window. You should explicitly call this function only for special circumstances.

See Also `sm_allget`

sm_ltofield

Writes a long integer value to a field

```
int sm_ltofield(int field_number, long value);
int sm_e_ltofield(char *field_name, int element, long value);
int sm_i_ltofield(char *field_name, int occurrence, long value);
int sm_n_ltofield(char *field_name, long value);
int sm_o_ltofield(int field_number, int occurrence, long value);
```

`field_name, field_number`
: The field to receive `value`.

`element`
: The element in `field_name` to receive `value`.

`occurrence`
: The occurrence in the specified field to receive `value`.

`value`
: A long integer to put into the specified field.

Environment C only

Returns

0	Success.
-1	The field is not found.

Description The long integer passed to this function is converted to user-readable format and placed in `field_number`. If the number is longer than the field, it is truncated without warning, on the right or left depending on the field's justification.

Example

```
#include <smdefs.h>

   /* Set the number of fish in the sea to a
    * smallish number. */

   #define MEDITERRANEAN 4

   sm_i_ltofield("seas", MEDITERRANEAN, 14L);
```

See Also `sm_itofield`, `sm_lngval`

sm_m_flush

Flushes the status line

```
void sm_m_flush(void);
```

Description

`sm_m_flush` forces Panther to display updates to the status line. This is useful if you want to display the status of an operation with `sm_d_msg_line` without flushing the entire display like `sm_flush`.

Example

```
#include <smdefs.h>

   /* Process a big pile of records, providing
    * status as we go.
    */
   char buf[80];
   int k;

   k = 0;
   do {
      sprintf(buf, "Processing record %d", k + 1);
      sm_d_msg_line(buf, REVERSE | WHITE);
      sm_m_flush();
   } while (process(records[k++]) >= 0);
```

See Also

`sm_flush`

sm_*mail_attach

Sends an attachment with the email message

```
int sm_mail_attach(int obj_id, char *pathname, char *filename, int
    delete);
int sm_n_mail_attach(char *name, char *pathname, char *filename,
    int delete);
```

`obj_id`
: The object ID of the mail object.

`name`
: The name of the message.

`pathname`
: The path to the file.

`filename`
: The name to use when the file is saved by the message recipient. If the null string or null pointer is passed, the `filename` will be taken from `pathname`.

`delete`
: If not 0, the file will be deleted when the message is sent or deleted.

`name`
: The name assigned to the mail object.

Returns 0 Success.

Description `sm_mail_attach` adds an attachment to the mail message.

See Also `sm_mail_new`

sm_*mail_file_text

Specifies the file containing the text of the email message

```
int sm_mail_file_text(int obj_id, char *filename);
int sm_n_mail_file_text(char *name, char *filename);
```

`obj_id`
: The object ID of the mail object.

`filename`
: The file containing the mail message.

`name`
: The name assigned to the mail object.

Returns 0 Success.

Description `sm_mail_file_text` takes the message text from the specified text file.

sm_mail_message

Sends a simple email message

```
int sm_mail_message(char *to, char *subject, char *text);
```

`to`
: The recipient of the mail message.

`subject`
: The subject of the mail message.

`text`
: The text of the mail message.

Returns

0 Success.

Description

`sm_mail_message` mails a simple email message containing `text` with `to` as the email address.

The default values of `PR_MAIL_FROM`, `PR_MAIL_CC`, `PR_MAIL_BCC` and `PR_MAIL_RECEIPT` will be used if they are set.

sm_mail_new

Returns the object ID of a new email message

```
int sm_mail_new(char *name);
```

`name`
: An optional name to be assigned to the mail object that is to be created.

Returns

≥1 Integer handle to the new mail object.

- `PR_E_PROP_VAL`: `name` is not valid or an object with that name already exists.
- `PR_E_MALLOC`: unable to allocate memory for internal structures.
- `PR_E_OBJECT`: unexpected internal error.

Description

`sm_mail_new` returns the object ID of a new message. If `name` is supplied, it can be used to set properties of the message. Before the message is sent, the following properties can be set:

`PR_MAIL_SUBJECT`
: Text of the `Subject:` line.

`PR_MAIL_TEXT`
: Text of the message. There are several functions that also can be used to set the message text.

`PR_MAIL_FROM`
: Information for the `From:` line of mail messages. Some Mail Transfer Agents ignore this property.

`PR_MAIL_TO`
: Information for the `To:` line of mail messages.

`PR_MAIL_CC`
: Information for the `CC:` line of mail messages.

`PR_MAIL_BCC`
: Information for the `BCC:` line of mail messages.

`PR_MAIL_REPLYTO`
: Information for the `Reply-to:` line of mail messages. Ignored when using MAPI.

`PR_MAIL_RECEIPT`
: Whether to ask for a receipt when the mail is first read. This seemingly does not work for most Mail Transfer Agents (Outlook and Outlook Express in particular).

`PR_NAME`
: Name of the message.

See Also `sm_*mail_send`, `sm_*mail_attach`

sm_*mail_send

Sends an email message

```
int sm_mail_send(int obj_id);
int sm_n_mail_send(char *name);
```

`obj_id`
: The object ID of the mail object.

`name`
: The name assigned to the mail object.

Returns
0 Success.
Error code from `smuprapi.h`

Description `sm_mail_send` sends the message identified by `obj_id` or `name` and deletes it.

See Also `sm_mail_new`

sm_*mail_text

Specifies the field containing the text of the email message

```
int sm_mail_text(int obj_id, char *field_name);
int sm_n_mail_text(char *name, char *field_name);
```

`obj_id`
: The object ID of the mail object.

`name`
: The name assigned to the mail object.

`field_name`
: The field containing the text of the mail message.

Returns 0 Success.

Description `sm_mail_text` takes the message text from the specified field. If the field is not word wrapped, each occurrence will be placed on a new line.

See Also `sm_*mail_file_text`

sm_*mail_widget

Sends an image of a Panther widget as an attachment to the mail message

```
int sm_mail_widget(int obj_id, char *widget_name,
    char *attachment_name, int quality);
int sm_n_mail_widget(char *name, char *widget_name,
    char *attachment_name, int quality);
```

`obj_id`
: The mail object ID.

`name`
: The name assigned to the mail object.

`widget_name`
: The widget to be converted to a JPEG file and attached to the mail message. `PR_APPLICATION` will send the complete MDI frame.

`attachment_name`
: The name to use when the file is saved by the message recipient. If the null string or null pointer is passed, the name used depends on the widget.

`quality`
: An integer from 0 to 100. Low numbers give poorer image quality but reduce the file size.

Returns

- 0 Success.
- Error code from `smuprapi.h`, most likely `PR_E_OBJECT` or `PR_E_OBJID`.

Description

`sm_*mail_widget` can only be used in `prodev` and `prorun`. The screen image of a widget is converted to a JPEG file and attached to the mail message.

sm_menu_bar_error

Returns the last error returned by a menu function

```
int sm_menu_bar_error(void);
```

Returns

0	`MNERR_OK`: Success.
-1	`MNERR_SCRIPT`: Script not found, or script or menu name not supplied.
-2	`MNERROR_EMPTY_SCOPE`: Menu not installed at specified scope.
-3	`MNERR_NOT_SUPPORTED`: Menus not supported.
-4	`MNERR_MENU`: Menu name not found.
-5	`MNERR_ITEM`: Item name not found.
-6	`MNERR_DATA`: Invalid data.
-7	`MNERR_MALLOC`: Memory allocation error.
-8	`MNERR_NULL`: Property has a value of null string pointer.
-9	`MNERR_READ_ONLY`: Property is read-only.
-10	`MNERR_LOCATION`: Invalid memory location.

Description

`sm_menu_bar_error` returns the error generated by the last call to a menu function. This is particularly useful for calls to `sm_menu_get` and `sm_mnitem_get` and their variants. These functions return the value of the specified property when successful; otherwise, they return `-1` for failure of the `_get_int` variants, and `0` for the `_get_str` variants. `sm_menu_bar_error` returns the actual cause of failure. It also lets you determine whether a return of `-1` indicates the property's actual value or an error condition.

Because Panther retains the error code only for the last call to one of the menu functions, call `sm_menu_bar_error` immediately afterward to evaluate the call's return status.

Example

```
/*enable and disable menu tear-offs*/

int ToggleTearOffs(void)
{
    int errorCode;
   switch(sm_menu_get_int(MNL_SCREEN, "menucom", "main", MN_TEAR))
    {
    case 0:                  /*enable tear-offs */
        sm_menu_change
               (MNL_SCREEN, "menucom", "main", MN_TEAR, 1, NULL);
```

```
            break;

        case 1:                   /*disable tear-offs */
            sm_menu_change
                   (MNL_SCREEN, "menucom", "main", MN_TEAR, 0, NULL);
            break;

        case -1:                 /* if error returned, find out why */
            errorCode = sm_menu_bar_error();
            menuErrorHandler(errorCode);
            break;
        }
    }
```

sm_menu_change

Sets a menu's properties

```
int sm_menu_change(int mem_location, char *script, char *menu,
    int prop, int intval, char *strval);
```

`mem_location`

The menu's memory location, one of these constants:

`MNL_ANY`
`MNL_APPLIC`
`MNL_SCREEN`
`MNL_FIELD`

If set to `MNL_ANY`, Panther looks for the menu in all memory locations. If the menu is installed in more than one location, the call fails and returns `MN_ERR_LOCATION`.

`script`

The name of a memory-resident script that contains the menu to change. The script must already be loaded into memory at `mem_location` by `sm_mnscript_load`. If you supply `NULL`, Panther searches among the most recently loaded script in `mem_location` for the specified menu.

`menu`

The menu to change. If set to `NULL`, Panther uses the first menu in `script`.

`prop`

The property to change. Table 5-11 lists properties that you can change and their constants.

`intval`

The integer value to set for `prop`. Supply `0` if `prop` takes a string value.

`strval`

The string value to set for `prop`. Supply `NULL` if `prop` takes an integer value.

Environment C only

Returns

0	`MNERR_OK`: Success.
-1	`MNERR_SCRIPT`: Script not found, or script or menu name not supplied.
-3	`MNERR_NOT_SUPPORTED`: Menus not supported.
-4	`MNERR_MENU`: Menu name not found.

-6	`MNERR_DATA`: Invalid data.
-8	`MNERR_NULL`: Null string argument.
-9	`MNERR_READ_ONLY`: Property is read-only.
-10	`MNERR_LOCATION`: Invalid memory location.

Description

`sm_menu_change` sets a menu property. Menu properties are derived from a memory-resident script. Because `sm_menu_change` changes the specified script, all instances of menus from this script get the requested property change.

Specify the property to change through one of the constants in Table 5-11. Menu-specific properties begin with a prefix of `MN`. Properties that begin with `MNI` set defaults for new items that are added to the menu at runtime. If you call `sm_menu_change` to reset item property defaults, the changes only affect items that are added after this call; it leaves existing menu items unchanged. To reset item properties for individual items, call `sm_mnitem_change`.

Table 5-11 Menu properties that can be changed at runtime

Property	**Type***	**Description**
`MN_EXTERNAL`	`int`	A value of `PROP_ON` or `PROP_OFF` specifies whether to find this menu's definition in another script.
`MN_NAME`	`str`	The name of this menu. The function does not check for duplicate names.
`MN_TEAR`	`int`	A value of `PROP_ON` or `PROP_OFF` enables or disables this submenu as a tear-off menu.
`MN_TITLE`	`str`	A title to display with popup menus.
`MNI_ACCEL_ACTIVE`	`int`	A value of `PROP_ON` or `PROP_OFF` specifies whether menu item accelerators are active.
`MNI_ACTIVE`	`int`	A value of `PROP_ON` or `PROP_OFF` allows or disallows access to menu items. If `MNI_ACTIVE` is set to `PROP_OFF`, menu items are greyed out.
`MNI_INDICATOR`	`int`	A value of `PROP_ON` or `PROP_OFF` specifies whether to show the toggle indicator on items.

Table 5-11 Menu properties that can be changed at runtime *(Continued)*

Property	Type*	Description

`MNI_SEP_STYLE`	`int`	The default style used by separator-type items, specified by one of these integer constants:
		
		`SEP_SINGLE`
		`SEP_DOUBLE`
		`SEP_NOLINE`
		`SEP_SINGLE_DASHED`
		`SEP_DOUBLE_DASHED`
		`SEP_ETCHEDIN`
		`SEP_ETCHEDOUT`
		`SEP_ETCHEDIN_DASHED`
		`SEP_ETCHEDOUT_DASHED`
		`SEP_MENUBREAK`
		`SEP_TYPE_MASK`

`MNI_SHOW_ACCEL`	`int`	A value of `PROP_ON` or `PROP_OFF` specifies whether menu items display the accelerator key next to their labels.

* For integer-type properties, supply an argument for the intval parameter and set the strval parameter to `NULL`; for string-properties, supply an argument for the strval parameter and set the intval parameter to 0.

Example

```
/*enable and disable menu tear-offs*/

int ToggleTearOffs(void)
   {
      int errorCode;
      switch
         (sm_menu_get_int(MNL_SCREEN, "menucom", "main", MN_TEAR)
         {
            /*enable tear-offs */
            case 0: sm_menu_change
               (MNL_SCREEN, "menucom", "main", MN_TEAR, 1, NULL);
               break;

         /*disable tear-offs */
            case 1: sm_menu_change
               (MNL_SCREEN, "menucom", "main", MN_TEAR, 0, NULL);
               break;

         /* if error returned, find out why */
            case -1:
               errorCode = sm_menu_bar_error();
```

```
                        menuErrorHandler(errorCode);
                        break;
                  }
            }
```

See Also `sm_mnitem_change`

sm_menu_create

Defines a menu at runtime

```
int sm_menu_create(int mem_location, char *script, char *menu);
```

`mem_location`
: The memory location in which to load this menu, one of the following constants:

  `MNL_APPLIC`
  `MNL_SCREEN`
  `MNL_FIELD`

`script`
: The name of a memory-resident script to contain the menu. The script can be one previously loaded into memory at `mem_location` by `sm_mnscript_load`; otherwise, Panther creates a script in memory with the name that you supply.

`menu`
: The name of the menu to create. The menu name must be unique in `script`.

Returns

0	`MNERR_OK`: Success.
-3	`MNERR_NOT_SUPPORTED`: Menus not supported.
-6	`MNERR_DATA`: Menu name already exists or not supplied.
-7	`MNERR_MALLOC`: Memory allocation error.

Description

`sm_menu_create` defines a menu and loads it into memory as part of the specified script. After you create this menu, you can set its properties and create items for it through `sm_menu_change` and `sm_mnitem_create`, respectively. Like other menus that are loaded into memory, you can attach this menu to an application component—screen or widget—and make it available for display through `sm_menu_install`.

sm_menu_delete

Removes a menu from the specified script

```
int sm_menu_delete(int mem_location, char *script, char *menu);
```

`mem_location`
: The menu's memory location, one of the following constants:

  `MNL_APPLIC`
  `MNL_SCREEN`
  `MNL_FIELD`

`script`
: The name of a memory-resident script that contains the menu. The script must already be loaded into memory at `mem_location` by `sm_mnscript_load`. If you supply `NULL`, Panther searches in the most recently loaded script in `mem_location` for the specified menu.

`menu`
: The name of the menu to delete. If you supply `NULL`, Panther uses the first menu in `script`.

Returns

0	`MNERR_OK`: Success.
-1	`MNERR_SCRIPT`: Script not found, or script or menu name not supplied.
-3	`MNERR_NOT_SUPPORTED`: Menus not supported.
-4	`MNERR_MENU`: Menu name not found.

Description

`sm_menu_delete` removes a menu from memory at runtime and frees the memory allocated for it. This function also destroys all items in the menu and frees the memory associated with them. After you call this function, you can restore this menu only by reloading its script, provided the script's source file already contains the menu definition.

See Also

`sm_menu_create`

sm_menu_get*

Gets a menu's property

```
int sm_menu_get_int(int mem_location, char *script, char *menu,
    int prop);
char *sm_menu_get_str(int mem_location, char *script, char *menu,
    int prop);
```

`mem_location`
: The menu's memory location, one of the following constants:

```
MNL_APPLIC
MNL_SCREEN
MNL_FIELD
```

`script`
: The name of a memory-resident script that contains the menu. The script must already be loaded into memory at `mem_location` by `sm_mnscript_load`. If you supply `NULL`, Panther searches in the most recently loaded script in `mem_location` for the specified menu.

`menu`
: The menu's name. If you supply `NULL`, Panther uses the first menu in `script`.

`prop`
: The property to get. Table 5-12 lists the properties that you can get and their constants.

Returns

- A pointer to the property's current value, returned either as an integer or as a pointer to a string value.

`NULL` Error returned by a `_get_str` variant. Call `sm_menu_bar_error` to get the error code.

`-1` Error returned by a `_get_int` variant. Call `sm_menu_bar_error` to get the error code.

Description

`sm_menu_get_int` and `sm_menu_get_str` returns the current setting of the specified property. Use the `_int` variant for those properties that have an integer value—for example, `MN_TEAR`; use the `_str` variant for properties that take string values, such as `MN_NAME` and `MN_TITLE`.

Table 5-12 Menu properties

Property	Type*	Description
`MN_EXTERNAL`	`int`	A value of `PROP_ON` or `PROP_OFF` specifies whether to find this menu's definition in another script.
`MN_NAME`	`str`	The name of this menu.
`MN_NUM_ITEMS`	`int`	Number of items in this menu.
`MN_TEAR`	`int`	A value of `PROP_ON` or `PROP_OFF` enables or disables this submenu as a tear-off menu.
`MN_TITLE`	`str`	A title to display with popup menus.
`MNI_SHOW_ACCEL`	`int`	A value of `PROP_ON` or `PROP_OFF` specifies whether menu items display the accelerator key next to their labels.
`MNI_ACCEL_ACTIVE`	`int`	A value of `PROP_ON` or `PROP_OFF` specifies whether menu item accelerators are active.
`MNI_ACTIVE`	`int`	A value of `PROP_ON` or `PROP_OFF` allows or disallows user access to menu items. If `MNI_ACTIVE` is set to `PROP_OFF`, menu items are greyed out.
`MNI_INDICATOR`	`int`	A value of `PROP_ON` or `PROP_OFF` specifies whether to show the toggle indicator on items

`MNI_SEP_STYLE`	`int`	The default style used by separator-type items, specified by one of these constants:
		
		`SEP_SINGLE`
		`SEP_DOUBLE`
		`SEP_NOLINE`
		`SEP_SINGLE_DASHED`
		`SEP_DOUBLE_DASHED`
		`SEP_ETCHEDIN`
		`SEP_ETCHEDOUT`
		`SEP_ETCHEDIN_DASHED`
		`SEP_ETCHEDOUT_DASHED`

* *For integer-type properties, use `sm_menu_get_int`; for string-properties, use `sm_menu_get_str`.*

Example

```
/*enable and disable menu tear-offs*/

int ToggleTearOffs(void)
   {
      int errorCode;
      switch
         (sm_menu_get_int(MNL_SCREEN, "menucom", "main", MN_TEAR)
         {
            /*enable tear-offs */
            case 0: sm_menu_change
               (MNL_SCREEN, "menucom", "main", MN_TEAR, 1, NULL);
                break;

         /*disable tear-offs */
            case 1: sm_menu_change
               (MNL_SCREEN, "menucom", "main", MN_TEAR, 0, NULL);
               break;

         /* if error returned, find out why */
            case -1:
               errorCode = sm_menu_bar_error();
               menuErrorHandler(errorCode);
               break;
         }
   }
```

sm_menu_install

Makes a menu available for display

```
void int sm_menu_install(int scope, int mem_location,
    char *script, char *menu);
```

`scope`

Specifies the menu's scope within the application with one of these constants:

`MNS_APPLIC`

Associates `menu` with the application and displays it. An application menu displays with all screens unless you install another menu at screen scope (`MNS_SCREEN`). Under Motif, the application menu can display on the base window along with the active screen's menu if you set the `baseWindow` and `formMenus` resources to true. You can install an application menu only from a script that is loaded into application (`MNL_APPLIC`) memory.

`MNS_SCREEN`

Associates `menu` with the current screen and displays it. The menu displays when its screen is invoked or reexposed. You can install a screen menu from a script that is loaded into application (`MNL_APPLIC`) or screen (`MNL_SCREEN`) memory.

`MNS_SCRN_POPUP`

Associates `menu` with the current screen and makes it available for display as a popup that the user invokes when the cursor is outside a field or in field that has no menu associated with it. You can install a screen popup menu from a script that is loaded into application (`MNL_APPLIC`) or screen (`MNL_SCREEN`) memory.

`MNS_FIELD`

Associates a menu with the current field, and makes it available for display as a popup that the user invokes while in that field. You can install a field menu from a script in any memory location.

`mem_location`

Specifies the memory location in which `script` is loaded. A script's memory location determines the scope at which you can install its menus—for example, you can install a screen menu only from a script that is loaded into screen (`MNL_SCREEN`) or application (`MNL_APPLIC`) memory. You load a menu script into memory with sm_mnscript_load with one of the arguments in the following table. The table shows which scope arguments are valid for each memory location:

Memory location	**Valid scopes**
MNL_APPLIC	All
MNL_SCREEN	MNS_SCREEN, MNS_SCRN_POPUP, MNS_FIELD
MNL_FIELD	MNS_FIELD
MNL_ANY	Panther searches for the menu's script in all memory locations that are valid for the menu's scope, starting with the "lowest" location. For example, if you want to install a screen-level menu, (MNS_SCREEN), Panther first looks in screen memory, (MNL_SCREEN), then in application memory (MNL_APPLIC).

Refer to sm_mnscript_load for more information about these arguments.

script

The name of a memory-resident script that contains the menu to install. The script must already be loaded into memory at mem_location by sm_mnscript_load. If you supply NULL, Panther searches for menu in the script most recently loaded in mem_location. A NULL value requires you to supply a non-NULL value for menu.

menu

Specifies a menu definition in script to install. If you supply an empty string, Panther installs the first menu definition in script. Make sure that menu names among all scripts loaded at the same memory location are unique; otherwise, results can be unpredictable.

If you supply NULL, Panther uses the first menu in script. A NULL value requires you to supply a non-NULL value for script.

Returns

0	MNERR_OK: Success.
-1	MNERR_SCRIPT: Script not found, or script or menu name not supplied.
-3	MNERR_NOT_SUPPORTED: Invalid scope or menus not supported.

-4	`MNERR_MENU`: Menu name not found.
-7	`MNERR_MALLOC`: Memory allocation error.
-10	`MNERR_LOCATION`: Invalid memory location for specified scope.

Description

`sm_menu_install` finds a menu in the specified script and memory location and reads its definition. If the menu contains external references, Panther resolves these; it then makes the menu available for display.

Except for Motif versions, Panther applications can display only one menu bar at a time. For example, if an application contains multiple screens and each screen has its own menu, Panther displays only the menu bar of the active screen. Under Motif, an application menu and screen menu can display simultaneously.

The scope at which you install a menu determines when Panther displays it; its memory location determines whether you can have identical instances of the same menu.

Menus display according to their scope assignment as follows:

- An application menu displays at all times unless a screen menu is installed. Note that under Motif, an application menu bar can display along with a screen menu.
- A screen menu displays when its screen is invoked or reexposed. This menu also displays with successive screens that lack their own menus: with sibling and child windows; and, if invoked as a form, with other forms invoked later.
- A screen popup menu displays when invoked from an area of the screen that has no field, or when the cursor is in a field that has no menu of its own associated with it.
- A field menu displays as a popup that the user invokes while on that field.

You can install a menu at any scope that is the same or higher than the scope of its caller. For example, the application's startup routines in `jmain.c` can only install a menu at application scope, while a screen's entry procedure can install a menu at all scopes except field (`MNS_FIELD`); a field's entry procedure can install menus at all scopes, including field.

Panther installs a screen menu with the current screen, a field menu with the current field. If another menu is already installed at the specified scope, Panther removes the previous menu. If the same menu is already installed from the same memory location, Panther does not try to reinstall it.

Installing Menus with Shared Content

Because a script can be loaded only once into a given memory location, all menus installed from that location are identical. Panther provides only one memory location at the application level (`MNL_APPLIC`). So, all scripts in application memory are unique, and all instances of a menu installed from application memory are the same: changes in one are immediately propagated to all others.

You can install the same menu from application memory for different screens and fields; if you do, all instances of this menu are always the same. If you install the same menu from screen memory for different fields on that screen; all popup menus of those fields are identical.

For example, the following JPL procedure in an application's startup screen loads a menu script into application memory; it then installs the menu `scr_mn` for the startup screen from application memory:

```
if sm_mnscript_load(MNL_APPLIC, "mnscript_myprog") == MNERR_OK

   call sm_menu_install \
        (MNS_SCREEN, MNL_APPLIC,"mnscript_myprog", "scr_mn")
else
{
   msg emsg "No menu found for application. Goodbye"
   call jm_exit
}
return
```

Subsequently, other screens in this application can install their own instances of this menu with the following call:

```
call sm_menu_install \
     (MNS_SCREEN, MNL_APPLIC, "mnscript_myprog", "scr_mn")
```

All screens that display the `scr_mn` menu display the same menu. Thus, if one screen makes a menu option inactive, that option is inactive when other screens display that menu.

Installing Menus with Unique Content

Conversely, you can install multiple copies of the same menu for screens and widgets, where each copy is unique. Because screens and widgets can load menu scripts into their private memory locations, each location can maintain its own copy of a menu; changes to one have no effect on the others.

To install unique copies of the same menu for several screens, repeat these steps for each screen:

1. Load the menu script into screen memory—call `sm_mnscript_load` with an argument of `MNL_SCREEN`.
2. Install the menu from screen memory—call `sm_menu_install` with arguments of `MNS_SCREEN` and `MNL_SCREEN`.

Similarly, you can make sure that several widgets on a screen have unique copies of the same popup menu. Repeat these steps for each field:

1. Load the menu script into field memory for the widget—call `sm_mnscript_load` with an argument of `MNL_FIELD`.
2. Install the menu from the widget's memory—call `sm_menu_install` with arguments of `MNS_FIELD` and `MNL_FIELD`.

External Menus

A menu definition can specify submenus whose contents are defined outside the current script—that is, the submenu's External property is set to Yes. For maximum flexibility, the external flag contains no information about this menu's script name. Consequently, when you install a menu, Panther resolves external references by searching first among scripts in the same memory location, then among scripts in the next highest memory location, and so on.

For example, given a menu installed from screen memory, Panther tries to resolve each of its external references first by searching among other scripts in screen memory; if no match is found in screen memory, Panther continues the search among the scripts loaded into application memory. If no menu is found in any memory location, Panther displays an empty submenu.

Removing Menus from Memory

You can explicitly remove any instance of a menu by calling `sm_menu_remove`. Otherwise, the menu remains installed until its screen or widget is removed from memory—for example, when a screen with its own menu is removed from the form or window stack. Panther automatically removes all menus and frees their memory when the application exits.

sm_menu_remove

Removes a menu from display

```
int sm_menu_remove(int scope);
```

`scope`
: Specifies which menu to remove from display:

  `MNS_APPLIC`
  : Removes the application menu.

  `MNS_SCREEN`
  : Removes the current screen's menu, either installed with the current screen or inherited from another screen.

  `MNS_FIELD`
  : Removes the current field's menu.

Returns

0	`MNERR_OK`: Success.
-2	`MNERROR_EMPTY_SCOPE`: Menu not installed at specified scope.
-3	`MNERR_NOT_SUPPORTED`: Invalid scope or menus not supported.

Description

`sm_menu_remove` makes a menu unavailable for display at the specified scope. Because the script remains loaded, any subsequent changes to the menu's properties become visible when you reinstall it.

This function has no effect on other instances of the menu that are installed from the same memory location.

See Also

`sm_menu_install`

sm_message_box

Displays a message in a dialog box

```
int sm_message_box(char *text, char *title, unsigned int options,
   char *icon);
```

`text`
: The text of the message. The text can contain format options shown in "Description." For Motif, the text has a maximum size of 75 characters.

`title`
: The title of the dialog box. A null pointer or empty string specifies no title.

`options`
: A bit mask that specifies message box display and behavior. Arguments that set different bits can be OR'd together. Table 5-13shows the flags that you can set on this mask.

`icon`
: Specifies the icon to use in the dialog box. The icon specified here overrides any icon set through `options`. This argument is ignored in character-mode.

Returns An integer that indicates which button was pushed:

1	`SM_IDOK`: OK
2	`SM_IDCANCEL`: Cancel
3	`SM_IDABORT`: Abort
4	`SM_IDRETRY`: Retry
5	`SM_IDIGNORE`: Ignore
6	`SM_IDYES`: Yes
7	`SM_IDNO`: No
8	`SM_IDHELP`: Help
9	`SM_IDYESALL`: Yes to All
10	`SM_IDOKALL`: OK to All
11	`SM_ID_NOALL`: No to All

Description `sm_message_box` creates a dialog box that displays a message and requests the user to select a button. Panther prevents further interaction with the application until the function returns with the user's selection.

The message text is a single string that wraps within the window. The text can contain these `%` format options:

`%K`*`keyname`*
: Displays the specified key, where *`keyname`* is a logical key constant. When Panther displays the message, it replaces *`keyname`* with the key label string defined for that key in the key translation file. If there is no label, the `%K` is stripped out and the constant remains. Key constants are defined in `smkeys.h`

`%B`
: Beeps the terminal with `sm_bel` before the message displays. This escape character must precede the message text.

`%N`
: Creates a new line.

You control message box display and behavior by setting one or more flags in Table 5-13. You can set one flag from each group. Flag settings from different groups can be OR'd together.

Table 5-13 Message box settings

Flag settings (by group)	Display/Action
Button Combinations	
`SM_MB_OK`	OK
`SM_MB_OKCANCEL`	OK, Cancel
`SM_MB_ABORTRETRYIGNORE`	Abort, Retry, Ignore
`SM_MB_YESNOCANCEL`	Yes, No, Cancel
`SM_MB_YESNO`	Yes, No
`SM_MB_RETRYCANCEL`	Retry, Cancel
`SM_MB_YESALLNOCANCEL`	Yes, Yes to All, No, Cancel
`SM_MB_OKALL`	OK, OK to All
`SM_MB_OKHELP`	OK, Help
`SM_MB_OKCANCELHELP`	OK, Cancel, Help
`SM_MB_ABORTRETRYIGNOREHELP`	Abort, Retry, Ignore, Help

Table 5-13 Message box settings *(Continued)*

Flag settings (by group)	Display/Action
`SM_MB_YESNOCANCELHELP`	Yes, No, Cancel, Help
`SM_MB_YESNOHELP`	Yes, No, Help
`SM_MB_RETRYCANCELHELP`	Retry, Cancel, Help
`SM_MB_YESALLNOALLCANCEL`	Yes, Yes to all, No, No to all, Cancel
System Icon Display	
`SM_MB_ICONNONE`	No icon
`SM_MB_ICONSTOP`	Stop
`SM_MB_ICONQUESTION`	Question
`SM_MB_ICONWARNING`	Warning
`SM_MB_ICONINFORMATION`	Information
Default Button	
`SM_MB_DEFBUTTON`*n*	Sets the button in the *n*th position as the default button.
Modality	
`SM_MB_APPLMODAL`	Confines user interaction to message box until message is acknowledged; user can interact freely with other applications.
`SM_MB_SYSTEMMODAL`	Confines user interaction to message box until message is acknowledged.

The following sections describe these settings in more detail.

Button Combinations

User options are controlled through the message box buttons. Table 5-13 shows the permissible combinations and the constants that set them.

Your message file defines the labels of message box buttons. You can edit this file and modify the label text. For more information on button label text, refer to “Customizing Push Button Labels for Message Boxes” on page 45-23 in *Application Development Guide*.

System Icon You can use the `options` parameter to set a flag for the system icon you want to display in the message window, if any. The actual icon that appears is platform-specific. In character mode, Panther searches in the message file for the tag that corresponds to the specified icon and its associated text; this text appears in front of the title text. For information on modifying message file tags, refer to “Using Message Files” on page 45-2 in *Application Development Guide*.

Default Buttons The `options` parameter can set the default button. The default button is specified by position—for example, you can set the third button as the default. You cannot set the Help button as the default button.

Modality Panther requires the user to respond to the message before continuing interaction with the application. You can extend this restriction to the entire system, and thereby prevent interaction with other applications, by setting `SM_MB_SYSTEMMODAL` on the `options` parameter. The default modality setting is `SM_MB_APPLMODAL`, which constrains user interaction only within the Panther application.

Example

```
proc clean_exit()
    {
       vars btnPush
       btnPush = sm_message_box("Save changes before exiting?",\
                  "", SM_MB_YESNOCANCEL | SM_MB_ICONQUESTION,"")

   if (btnPush == SM_IDCANCEL)
       {
           return
       }
       if (btnPush == SM_IDYES)
       {
          call save_changes()
       }
       if (btnPush == SM_IDNO)
       {
          call sm_jclose()
       }

}
```

sm_mncrinit6

Initializes support for Panther's menu subsystem

```
void sm_mncrinit6(void);
```

Environment C only

Description `sm_mncrinit6` is usually called automatically when you enable menus in your application. This function is called and menu support is enabled if you set `MENUS` to 1 in the main function.

`sm_mncrinit6` sets a global variable to point to a control function. All screen manager functions that need menu support check the variable and, if it is non-zero, call indirectly with the request.

Call this function explicitly only if you write your own executive. You must call `sm_mncrinit6` in the main function before the call to `sm_initcrt`.

sm_*mnitem_change

Sets a menu item's property

```
int sm_mnitem_change(int mem_location, char *script, char *menu,
    int item_no, int prop, int intval, char *strval);
int sm_n_mnitem_change(int mem_location, char *script, char *menu,
    char *item_name, int prop, int intval, char *strval);
```

`mem_location`
: The memory location of the item's menu, one of the following constants:

  `MNL_ANY`
  `MNL_APPLIC`
  `MNL_SCREEN`
  `MNL_FIELD`

  If you supply `MNL_ANY`, Panther looks for the menu in all memory locations. If the menu is installed in more than one location, the function call fails and returns `MN_ERR_LOCATION`.

`script`
: The name of a memory-resident script that contains the menu to change. The script must already be loaded into memory at `mem_location` by `sm_mnscript_load`. If you supply `NULL`, Panther searches in the most recently loaded script in `mem_location` for the specified menu.

`menu`
: The name of the item's menu, as listed in the Submenu field of the menu bar editor or with the `MENU` keyword in an ASCII menu file. If you supply `NULL`, Panther uses the first menu in `script`.

`item_no, item_name`
: Specifies the menu item to change by its number or name:

- `sm_mnitem_change` identifies the item by its numeric offset within the menu, where the first menu item is `0`.
- `sm_n_mnitem_change` identifies the item by its name.

`prop`
: The property to change, one of the constants listed in Table 5-14.

`intval`
: The integer value to set for `prop`. If the property takes a string value, supply 0.

`strval`
: The string value to set for `prop`. If the property takes an integer value, supply `NULL`.

Returns

0	`MNERR_OK`: Success.
-1	`MNERR_SCRIPT`: Script not found, or script or menu name not supplied.
-3	`MNERR_NOT_SUPPORTED`: Menus not supported.
-4	`MNERR_MENU`: Menu name not found.
-5	`MNERR_ITEM`: Item name not found.
-6	`MNERR_DATA`: Invalid data.
-7	`MNERR_MALLOC`: Memory allocation error.
-8	`MNERR_NULL`: Null string argument.
-9	`MNERR_READ_ONLY`: Property is read-only.

Description

`sm_mnitem_change` sets the property of a menu item. Menu item properties are derived from a memory-resident script. Because `sm_mnitem_change` changes the specified script, all instances of items from this script get the property change.

Menu Item Property Constants

Table 5-14 lists menu item property constants and the values you can set these to. Integer and string properties are listed in separate groups.

Table 5-14 Menu item properties that can be changed at runtime

Constant	Property values
Integer properties:	
`MNI_ACCEL`	An accelerator keystroke that specifies the keyboard equivalent for selecting this menu item, valid only for action and toggle menu items. You cannot set this property for main menu items. Accelerator keys for edit-type items such as Edit Cut or Edit Paste are set by the GUI platform—for example, in Windows, through the Panther initialization file; on Motif, in the Panther file. To change edit item accelerators, modify the appropriate GUI file.
`MNI_ACCEL_ACTIVE`	A value of `PROP_ON` or `PROP_OFF` specifies whether the menu item accelerator is active.
`MNI_ACTIVE`	A value of `PROP_ON` or `PROP_OFF` allows or disallows user access to this menu item. If `MNI_ACTIVE` is set to `PROP_OFF`, the menu item is greyed out.

Table 5-14 Menu item properties that can be changed at runtime *(Continued)*

Constant	Property values
`MNI_DISPLAY_ON`	Specifies whether to display the menu item on the menu and/or the tool bar. Supply one of these arguments: `DISPLAY_MENU`: Menu only (default) `DISPLAY_TOOL`: Tool bar only `DISPLAY_BOTH`: Menu and tool bar `DISPLAY_NEITHER`: Neither
`MNI_INDICATOR`	A value of `PROP_ON` or `PROP_OFF` specifies whether to show the toggle indicator.
`MNI_IS_HELP`	A value of `PROP_ON` or `PROP_OFF` specifies whether to display this item as the rightmost item on the menu bar.
`MNI_MNEMONIC`	A zero-based offset into the item's label that specifies which character users can type to select this item, provided the menu is displayed. A value of -1 specifies no mnemonic for this item.
`MNI_ORDER*`	The order in which this item appears on the toolbar. The default value is 100. You can enter any value between 0 and 200, inclusive. If all toolbar items are set to the same value, they appear in the same order as they do in the menu.

`MNI_SEP_STYLE`	The style used by an item separator, specified by one of these constants:
	
	`SEP_SINGLE`
	`SEP_DOUBLE`
	`SEP_NOLINE`
	`SEP_SINGLE_DASHED`
	`SEP_DOUBLE_DASHED`
	`SEP_ETCHEDIN`
	`SEP_ETCHEDOUT`
	`SEP_ETCHEDIN_DASHED`
	`SEP_ETCHEDOUT_DASHED`

`MNI_SHOW_ACCEL`	A value of `PROP_ON` or `PROP_OFF` specifies whether a menu item displays the accelerator key next to the item label.

Table 5-14 Menu item properties that can be changed at runtime *(Continued)*

Constant	Property values
String properties:	
`MNI_ACT_PIXMAP*`	The name of an image file whose contents are shown for an active toolbar item—that is, accessible but not pressed. Refer to Table 25-1 on page 25-15 in *Using the Editors* for valid file types, and for information about path and extension options.
`MNI_ARM_PIXMAP*`	The name of an image file whose contents are shown for an armed toolbar item—that is, in its pressed state. If this property is blank, Motif uses the `MNI_ACT_PIXMAP` property for the item's armed state. Windows uses a modified version of the Active Pixmap property to display a toolbar item's armed state and ignores this property.
`MNI_CONTROL`	A control string that specifies the action that occurs when this item is selected.
`MNI_EXT_HELP_TAG`	A help context identifier that specifies the help to invoke from an external help program.
`MNI_HOT_PIXMAP*`	The name of an image file whose contents are shown when a pointer moves over an active toolbar item. (Windows only)
`MNI_INACT_PIXMAP*`	The name of an image file whose contents are shown for an inactive or unavailable (grayed) item. If this property is blank, Motif displays an empty toolbar item. Windows uses a grayed version of the Active Pixmap property to display a toolbar item's inactive state if a pixmap is not specified.
`MNI_HELP_SCREEN`	The name of a Panther screen to invoke as a help screen.
`MNI_LABEL`	A string expression to display as this item's label.
`MNI_MEMO`	A string expression for this menu item's Memo Text property.
`MNI_NAME`	The menu item's name. This function does not check for duplicate names.
`MNI_STAT_TEXT`	A string expression to display on the screen's status line when this item has focus.
`MNI_SUBMENU`	Name of the submenu to invoke when this item is selected.

Table 5-14 Menu item properties that can be changed at runtime *(Continued)*

Constant	Property values
`MNI_TM_CLASS`	The transaction manager class assigned to this menu item. This property determines how the item behaves in each of the transaction manager modes. Refer to "Using Styles and Classes" on page 23-5 in *Using the Editors* for more information on transaction manager classes.
`MNI_TOOL_TIP*`	The balloon help to display when the cursor remains over the toolbar item.
* Ignored in character-mode.	

Calling from JPL `sm_mnitem_change` and `sm_n_mnitem_change` have too many parameters to allow installation by `sm_install`; consequently, they are not directly accessible to JPL modules. (Refer to "Installing Prototyped Functions" on page 44-9 in *Application Development Guide* for function installation requirements.) A number of wrapper functions that call `sm_mnitem_change` and `sm_n_mnitem_change` are declared and installed in `funclist.c`. You can call these functions from JPL to modify menu items.

Table 5-15 lists the provided wrapper functions and their parameter declarations. Each wrapper function is narrowly defined to look for a menu in a discrete memory location—application, screen, or field—or to look in all memory locations (the `change_i_any` and `change_s_any` variants). Also, the `change_i` variants set only integer properties; the `change_s` variants set only string properties. All parameters are identical in type and purpose to those declared for `sm_mnitem_change` and `sm_n_mnitem_change`.

Table 5-15 Wrapper functions for changing menu item properties from JPL

Function names	Parameter declarations
To modify integer properties, call:	

`sm_n_mnitem_change_i_any*`
`sm_n_mnitem_change_i_app`
`sm_n_mnitem_change_i_screen`
`sm_n_mnitem_change_i_field`
	`(char *script, char *menu, char *item_name, int prop, int intval)`

Table 5-15 Wrapper functions for changing menu item properties from JPL *(Continued)*

Function names	Parameter declarations

`sm_mnitem_change_i_any*`
`sm_mnitem_change_i_app`
`sm_mnitem_change_i_screen`
`sm_mnitem_change_i_field`
	`(char *script, char *menu, int item_no, int prop, int intval)`

To modify string properties, call:	

`sm_n_mnitem_change_s_any*`
`sm_n_mnitem_change_s_app`
`sm_n_mnitem_change_s_screen`
`sm_n_mnitem_change_s_field`
	`(char *script, char *menu, char *item_name, int prop, char *strval)`

`sm_mnitem_change_s_any*`
`sm_mnitem_change_s_app`
`sm_mnitem_change_s_screen`
`sm_mnitem_change_s_field`
	`(char *script, char *menu, int item_no int prop, char *strval)`

* Panther looks for the menu in all memory locations. If the menu is installed in more than one location, the function call fails and returns `MN_ERR_LOCATION`.

sm_*mnitem_create

Inserts a new item into a menu

```
int sm_mnitem_create(int mem_location, char *script, char *menu,
   int next_item_no, int item_type, char *item_name);
int sm_n_mnitem_create(int mem_location, char *script, char *menu,
   char *next_item_name, int item_type, char *item_name);
```

`mem_location`
: The memory location of the item's menu, one of the following constants:

  `MNL_APPLIC`
  `MNL_SCREEN`
  `MNL_FIELD`

`script`
: The name of a memory-resident script that contains the item's menu. The script must already be loaded into memory at `mem_location` by `sm_mnscript_load`. If you supply `NULL`, Panther searches in the most recently loaded script in `mem_location` for the specified menu.

`menu`
: The name of the item's menu, as listed in the Submenu field of the menu bar editor or with the `MENU` keyword in an ASCII menu file. If you supply `NULL`, Panther uses the first menu in `script`.

`next_item_no, next_item_name`
: Specifies the new item's position by the number or name of the item to follow it:

  - `sm_mnitem_create` identifies the next item by its numeric offset within the menu, where the first menu item is `0`. Supply `-1` to append the new item to the end of the menu.
  - `sm_n_mnitem_create` identifies the next item by its name. Supply `NULL` to append the new item to the end of the menu.

`item_type`
: The item's type. Supply one of the constants described in Table 5-16.

`item_name`
: The name to assign this item. Item names must be unique within the same menu. Supply `NULL` to create an unnamed item.

Environment C only

Returns

0	MNERR_OK: Success.
-1	MNERR_SCRIPT: Script not found, or script or menu name not supplied.
-3	MNERR_NOT_SUPPORTED: Menus not supported.
-4	MNERR_MENU: Menu name not found.
-5	MNERR_ITEM: Item name not found.
-6	MNERR_DATA: Item name already exists.
-7	MNERR_MALLOC: Memory allocation error.

Description sm_mnitem_create inserts a new menu item into a menu. After you create this item, you can set its properties through sm_mnitem_change. The menu displays this item at the next delayed write.

Table 5-16 lists menu item type constants.

Table 5-16 Menu item type constants

Item type constants	Item behavior
MI_SEPARATOR	Draws a separator between the previous and next menu items, according to the specified separator style MNI_SEP_STYLE).
MI_SUBMENU	Invokes another menu. If a MI_SUBMENU-type item is on the menu bar, its submenu displays as a pulldown; otherwise, the submenu displays to its right.
MI_ACTION_BTTN	Invokes an action through a control string.
MI_TOGGLE_BTTN	Invokes an action through a control string and toggles the indicator on or off.
MT_WINDOWS_OPT	Invokes the windows menu of the current platform—for example, under Windows, the Windows menu with Arrange Icons, Tile, and Cascade. This item is ignored in character mode.
MT_WINDOWS_LIST	Invokes a menu that lists all open windows.
MT_EDIT_CUT*	Cuts selected text to the clipboard.
MT_EDIT_DELETE*	Deletes the selected text.
MT_EDIT_PASTE*	Pastes the clipboard contents.
MT_EDIT_SELECT*	Selects the current widget's contents.

Table 5-16 Menu item type constants *(Continued)*

Item type constants	Item behavior
`MT_EDIT_COPY*`	Copies selected text to the clipboard.
`MT_EDIT_CLEAR*`	Replaces the selected text with blank spaces.

*Under Windows and Motif, use edit-type items only on a pulldown or popup menu. Windows and Motif inactivate edit-type menu items when they appear on a menu bar.

sm_*mnitem_delete

Removes an item from a menu

```
int sm_mnitem_delete(int mem_location, char *script, char *menu,
    int item_no);
int sm_n_mnitem_delete(int mem_location, char *script, char *menu,
    char *item_name);
```

`mem_location`
: The memory location of the item's menu, one of the following constants:

```
MNL_APPLIC
MNL_SCREEN
MNL_FIELD
```

`script`
: The name of a memory-resident script that contains the item's menu. If you supply `NULL`, Panther searches in the most recently loaded script in `mem_location` for the specified menu.

`menu`
: The name of the item's menu, as listed in the Submenu field of the menu bar editor or with the `MENU` keyword in an ASCII menu file. If you supply `NULL`, Panther uses the first menu in `script`.

`item_no, item_name`
: Specifies the menu item to delete by its number or name:

- `sm_mnitem_delete` identifies the item by its numeric offset within the menu, where the first menu item is `0`.
- `sm_n_mnitem_delete` identifies the item by its name.

Returns

0	`MNERR_OK`: Success.
-1	`MNERR_SCRIPT`: Script not found, or script or menu name not supplied.
-3	`MNERR_NOT_SUPPORTED`: Menus not supported.
-4	`MNERR_MENU`: Menu name not found.
-5	`MNERR_ITEM`: Item name not found.

Description

`sm_mnitem_delete` removes an item from a menu and frees the memory associated with it. Panther updates the menu display at the first delayed write.

sm_*mnitem_get

Gets a menu item's property

```
int sm_mnitem_get_int(int mem_location, char *script, char *menu,
   int item_no, int prop);
int sm_n_mnitem_get_int(int mem_location, char *script,
   char *menu, char *item_name, int prop);
char *sm_mnitem_get_str(int mem_location, char *script,
   char *menu, int item_no, int prop);
char *sm_n_mnitem_get_str(int mem_location, char *script,
   char *menu, char *item_name, int prop);
```

`mem_location`
: The memory location of the item's menu, one of the following constants:

```
MNL_APPLIC
MNL_SCREEN
MNL_FIELD
```

`script`
: The name of a memory-resident script that contains the item's menu. The script must already be loaded into memory at `mem_location` by `sm_mnscript_load`.

`menu`
: The name of the item's menu, as listed in the Submenu field of the menu bar editor or with the `MENU` keyword in an ASCII menu file.

`item_no, item_name`
: Specifies the menu item by its number or name:

- `sm_mnitem_get` identifies the item by its numeric offset within the menu, where the first menu item is `0`.
- `sm_n_mnitem_get` identifies the item by its name.

`prop`
: The property to get. Supply one of the constants described in Table 5-17.

Returns

- The property's current value, returned either as an integer or as a pointer to a string value. Because this function stores a returned string in a pool of buffers that it shares with other functions, copy or process this data immediately.

`NULL`	Error returned by `_get_str` variants. Call `sm_menu_bar_error` to get the error code.
-1	Error returned by `_get_int` variants. Call `sm_menu_bar_error` to get the error code.

Description

`sm_mnitem_get_int` and `sm_mnitem_get_str` return the current setting of the specified property. Use the `_int` variant for those properties that have an integer value—for example, `MNI_SEP_STYLE`; use the `_str` variant for properties that take string values, such as `MNI_NAME` and `MNI_ACCEL`.

Table 5-17 lists the menu item property constants that you can supply as arguments to the `prop` parameter and the values that these return. Integer and string properties are listed in separate groups.

Table 5-17 Menu item properties

Constant	Property values
Integer properties:	
`MNI_ACCEL`	An accelerator keystroke that specifies the keyboard equivalent for selecting this menu item, valid only for action and toggle menu items.
`MNI_ACCEL_ACTIVE`	A value of `PROP_ON` or `PROP_OFF` specifies whether the menu item accelerator is active.
`MNI_ACTIVE`	A value of `PROP_ON` or `PROP_OFF` allows or disallows user access to this menu item. If `MNI_ACTIVE` is set to `PROP_OFF`, the menu item is greyed out.
`MNI_DISPLAY_ON`	Specifies whether to display the menu item on the menu and/or the tool bar. Supply one of these arguments: `DISPLAY_MENU`: Menu only (default). `DISPLAY_TOOL`: Tool bar only. `DISPLAY_BOTH`: Menu and tool bar. `DISPLAY_NEITHER`: Neither.
`MNI_INDICATOR`	A value of `PROP_ON` or `PROP_OFF` specifies whether to show the toggle indicator.
`MNI_IS_HELP`	A value of `PROP_ON` or `PROP_OFF` specifies whether to display this item as the rightmost item on the menu bar.

Table 5-17 Menu item properties *(Continued)*

Constant	Property values
`MNI_MNEMONIC`	A zero-based offset into the item's label that specifies which character users can type to select this item, provided the menu is displayed. A value of -1 indicates that the item has no mnemonic set.
`MNI_ORDER*`	The order in which this item appears on the toolbar. The default value is 100. You can enter any value between 0 and 200, inclusive. If all toolbar items are set to the same value, they appear in the same order as they do in the menu.

`MNI_SEP_STYLE`	The style used by an item separator, specified by one of these constants:
	
	`SEP_SINGLE`
	`SEP_DOUBLE`
	`SEP_NOLINE`
	`SEP_SINGLE_DASHED`
	`SEP_DOUBLE_DASHED`
	`SEP_ETCHEDIN`
	`SEP_ETCHEDOUT`
	`SEP_ETCHEDIN_DASHED`
	`SEP_ETCHEDOUT_DASHED`

`MNI_SHOW_ACCEL`	A value of `PROP_ON` or `PROP_OFF` specifies whether a menu item displays the accelerator key next to the item label.
`MNI_TM_CLASS`	The transaction manager class assigned to this menu item. This property determines how the item behaves in each of the transaction manager modes. Refer to "Using Styles and Classes" on page 23-5 in *Using the Editors* for more information on transaction manager classes.

Table 5-17 Menu item properties *(Continued)*

Constant	Property values

`MNI_TYPE`	The menu item's type, specified by one of the following constants:
	
	`MI_SEPARATOR`
	`MI_SUBMENU`
	`MI_ACTION_BTTN`
	`MI_TOGGLE_BTTN`
	`MT_WINDOWS_OPT`
	`MT_WINDOWS_LIST`
	`MT_EDIT_CUT`
	`MT_EDIT_DELETE`
	`MT_EDIT_PASTE`
	`MT_EDIT_SELECT`
	`MT_EDIT_COPY`
	`MT_EDIT_CLEAR`

String properties:	
`MNI_ACT_PIXMAP*`	The name of an image file whose contents are shown for an active toolbar item—that is, accessible but not pressed. Refer to Table 25-1 on page 25-15 in *Using the Editors* for valid file types. File paths and extensions are optional; for more information, refer to "Filename Extensions" on page 25-16 in *Using the Editors*.
`MNI_ARM_PIXMAP*`	The name of an image file whose contents are shown for an armed toolbar item—that is, in its pressed state. If this property is blank, Motif uses the `MNI_ACT_PIXMAP` property for the item's armed state. Windows uses a modified version of the Active Pixmap property to display a toolbar item's armed state and ignores this property.
`MNI_CONTROL`	A control string that specifies the action that occurs when this item is selected.
`MNI_EXT_HELP_TAG`	A help context identifier that specifies the help to invoke from an external help program.
`MNI_HOT_PIXMAP*`	The name of an image file whose contents are shown when a pointer moves over an active toolbar item. (Windows only)

Table 5-17 Menu item properties *(Continued)*

Constant	Property values
`MNI_INACT_PIXMAP *`	The name of an image file whose contents are shown for an inactive or unavailable (grayed) item. If this property is blank, Motif displays an empty toolbar item. Windows uses a grayed version of the Active Pixmap property to display a toolbar item's inactive state if a pixmap is not specified.
`MNI_HELP_SCREEN`	The name of a Panther screen to invoke as a help screen.
`MNI_LABEL`	A string expression to display as this item's label.
`MNI_MEMO`	A string expression for this menu item's Memo Text property.
`MNI_NAME`	The menu item's name.
`MNI_STAT_TEXT`	A string expression to display on the screen's status line when this item has focus.
`MNI_SUBMENU`	Name of the submenu to invoke when this item is selected.
`MNI_TOOL_TIP*`	The balloon help to display when the cursor remains over the toolbar item.

* Ignored in character-mode.

sm_mnscript_load

Loads a menu script into memory and makes its menus available for installation

```
int sm_mnscript_load(int mem_location, char *script);
```

`mem_location`

Specifies where to load this script into memory. You can load a script only once into a given memory location. The script's memory location determines the scope at which its menus can be installed and whether you can install identical instances of the same menu.

`MNL_APPLIC`

Loads the menu script into application memory. Menus in application memory can be installed at any scope—application, screen, and field. All instances of a menu installed from application memory are always identical; changes in one are immediately propagated to the others.

`MNL_SCREEN`

Loads the menu script into the current screen's memory. Each screen maintains its own memory location. You can install menus for a screen and its widgets from that screen's memory.

`MNL_FIELD`

Loads the menu script into the current field's memory. Each field maintains its own memory location. You can install a popup menu for a field from its own memory location.

`script`

The name of the menu script to load into memory.

Returns

0	`MNERR_OK`: Success.
-1	`MNERR_SCRIPT`: Script not found, or script or menu name not supplied.
-3	`MNERR_NOT_SUPPORTED`: Menus not supported.
-7	`MNERR_MALLOC`: Memory allocation error.
-10	`MNERR_LOCATION`: Invalid memory location.

Description

`sm_mnscript_load` loads the specified script into application, screen, or field memory. All menus that are defined in that script are subsequently available for installation and display through `sm_menu_install`.

`sm_mnscript_load` lets you load a menu into any memory location that is the same or higher than its caller, as shown in Table 5-18:

Table 5-18 Valid menu script load locations

`sm_mnscript_load` **caller**	**Valid memory locations**
Application	`MNL_APPLIC`
Screen	`MNL_SCREEN` `MNL_APPLIC`

Widget	`MNL_FIELD`
	`MNL_SCREEN`
	`MNL_APPLIC`

For example, the application's startup routines in `jmain.c` can only load menu scripts into application memory, while a screen's entry procedure can load scripts into application memory and into its own memory.

A menu script's memory location determines the scope at which its menus can be installed:

- Application memory menus can be installed at all scopes: application, screen, and field. Instances of a menu installed from application memory all share the same content; changes to one are propagated to all.
- Screen memory menus can be installed at screen and field scopes. All copies of a screen menu installed from screen memory are unique; copies of a field menu installed from screen memory all share the same content within that screen.
- Field memory menus can be installed only at field scope. All instances of a field menu installed from field memory are unique.

See Also `sm_mnscript_unload`

sm_mnscript_unload

Removes a script from memory and destroys all menus installed from it

```
int sm_mnscript_unload(int mem_location, char *script);
```

`mem_location`
: The memory location that contains the menu script, one of the following constants:

  `MNL_APPLIC`
  `MNL_SCREEN`
  `MNL_FIELD`

`script`
: The menu script to unload. An argument of `NULL` unloads the script last loaded in `mem_location`.

Returns

0	`MNERR_OK`: Success.
-1	`MNERR_SCRIPT`: Script not found.
-10	`MNERR_LOCATION`: Invalid memory location.

Description

`sm_mnscript_unload` removes `script` from the specified memory location and destroys all menus that are installed from it. If any of those menus are currently displayed, Panther removes them immediately. If a menu is referenced as an external menu, Panther displays an empty menu in its place.

See Also

`sm_mnscript_load`

sm_ms_inquire

Gets information about the mouse's current state

```
int sm_ms_inquire(int request);
```

`request`
Specifies the data to get, one of the following constants:

`MOUSE_LINE`
The line of the physical display on which the mouse click occurred.

`MOUSE_COLM`
The column of the physical display on which the mouse click occurred.

`MOUSE_SHIFT`
The state of the Shift, Control, and Alt keys during the mouse click. Panther returns this information in an integer bit mask. For bit settings, refer to the Description.

`MOUSE_BUTTONS`
The state of all mouse buttons, left, middle, and right. Panther returns this information in an integer bit mask. For states that are recognized by Panther and their corresponding bit settings, refer to the Description.

`MOUSE_FIELD`
The number of the field in which the mouse click occurred. If the mouse click occurs outside a field, the function returns -1.

`MOUSE_FORM_LINE`
The number of the Panther screen line on which the mouse click occurred.

`MOUSE_FORM_COLM`
The number of the Panther screen column on which the mouse click occurred.

Returns

- The data specified by `request`.

-1 Unable to get the requested data.

Description

`sm_ms_inquire` gets information about the mouse's current state—the position of the last mouse click on the physical or Panther screen, whether other keys are pressed in combination with it, and which mouse buttons have been pressed and how recently.

This function's returns an integer value whose bits are set according to the supplied argument, `MOUSE_SHIFT` or `MOUSE_BUTTONS`.

Mouse Events with Keyboard Modifiers

`MOUSE_SHIFT` sets the three lowest-order bits in the return value to indicate which of three keys—Shift, Ctrl, and Alt—are pressed at the same time as the mouse click. `sm_ms_inquire` can set these bits as follows, from lowest- to highest-order bit:

1	Shift key is down
1	Ctrl key is down
1	Alt key is down

For example, a return value of 2 (`0 1 0`) indicates that the Ctrl key is down, while a return value of 5 (`1 0 1`) indicates that the Alt and Shift keys are both down. The second of these returns can be represented as follows:

Alt	Ctrl	Shift
1	0	1

`MOUSE_BUTTONS` sets nine bits to indicate the state of the left, right, and middle mouse buttons. `sm_ms_inquire` puts the requested data in three segments of three bits each, where each segment represents one of three mouse buttons—left, right, and middle. The three lowest-order bits contain left button data; if the mouse has only one button, only these bit settings are significant. The three middle bits contain right button data, and the three highest-order bits contain data for the middle button, if any.

Each bit within a three-bit segment can be set as follows, from lowest- to highest-order bit:

0/1	Up/down
1	Just pressed
1	Just released

For example, the bit settings returned for a just-initiated point and click operation—left button is down and just pressed—can be represented as follows:

Middle Button			Right Button			LeftButton		
0	0	0	0	0	0	0	1	1

A click and drag operation that is in progress—right button is down—can be represented like this:

Middle Button			Right Button			LeftButton		
0	0	0	0	0	1	0	0	0

Only four combinations of bit settings are meaningful to Panther and recognized as valid button states:

- Up — `0 0 0`
- Down — `0 0 1`
- Down and just pressed — `0 1 1`
- Up and just released — `1 0 0`

Example

```
/*find out whether any button is down */

int is_any_button_down(void)
   {
      int retval;
      retval = -1;
      if (sm_ms_inquire(MOUSE_BUTTONS) > -1)
         return retval & 0x49;
      return retval;
   }
```

See Also

`sm_mus_time`

sm_msg

Displays a message at a given column on the status line

```
void sm_msg(int column, int disp_length, char *text);
```

`column`
: The message's start column on the status line. On terminals with onscreen attributes, you might need to adjust the column position to allow for attributes embedded in the status line. `sm_d_msg_line` explains how to embed attributes and function key names in a status line message.

`disp_length`
: The number of characters to display.

`text`
: The contents of the message.

Description

`sm_msg` merges the specified message with the current contents of the status line and displays it at the specified column. This function is called by the function that updates the cursor position display (refer to `sm_c_vis`).

Note: Messages generated by `sm_msg` have the lowest of priority among status line messages; consequently, its display is guaranteed only until the function returns to its caller, or until another message routine is called. Any messages that are subsequently posted to the status line overwrite the `sm_msg`-generated text.

Example

```
#include <smdefs.h>

   /* This code displays a message, then chops out
    * part of it.
    */

   char *text0 = "      ";
   char *text1 = "Message is displayed on the status "
      "line at col 1.";

   sm_msg(1, strlen(text1), text1);
   sm_msg(12, strlen(text0), text0);
```

See Also

`sm_d_msg_line`

sm_msg_del

Removes a class of messages from memory

```
#include <smerror.h>
int sm_msg_del(int class);
```

`class`

Specifies the class of messages to remove, where 0-7 are reserved for user-defined message classes, and the following classes, defined in `smerror.h`, are reserved for Panther:

`DM_MSGS`	`UT_MSGS`
`SM_MSGS`	`WB_MSGS`
`TP_MSGS`	

If the message file is not divided into sections, supply a value of 0.

Returns

0	Success.
-1	Unable to find message class.

Description

`sm_msg_del` removes the specified class of messages from memory. All messages of this class are thereafter inaccessible to the application unless explicitly reloaded through `sm_msg_read`.

See Also

`sm_msg_read`, `sm_msg_get`, `sm_msgfind`

sm_msg_get

Finds a message

```
#include <smerror.h>
char *sm_msg_get(int msg_id);
```

`msg_id`
: Specifies the message to get through its number or name, defined in `smerror.h`.

Returns

- A pointer to the text of `msg_id`'s message.
- A string that contains the message class and number if no message exists for `msg_id`.

Description

`sm_msg_get` gets a message from a message file previously loaded by `sm_msg_read`. Message files are binary files, created through the Panther utility `msg2bin`, whose contents are accessible through Panther library functions like `sm_msg_get`.

Example

```
#include <smdefs.h>
#include <smerror.h>

    /* Assume that an anxious programmer has just
     * typed in the question, "Will my boss like
     * my new program?" This code fragment answers
     * the question.
     */

    sm_n_putfield("answer", rand() & 1 ?
                  sm_msg_get(SM_YES):
                  sm_msg_get(SM_NO));
```

See Also

`sm_msgfind`, `sm_msg_read`, `sm_msg_set`

sm_*msg_read

Reads messages from a message file

```
#include <smerror.h>
int sm_msg_read(char *msg_prefix, int class, int no_replace,
   char *msgfile);
int sm_d_msg_read(char *msg_prefix, int class, int no_replace);
int sm_n_msg_read(char *msg_prefix, int class, int no_replace,
   char *msgfile);
```

`msg_prefix`

Specifies to read messages of this prefix within message class `class`. Panther messages have the following prefixes:

`DM`	Database interface
`DM_TM`	Transaction manager
`FM`	Editor
`JM`	Panther runtime
`JV`	Java
`SM`	Screen manager
`TP`	Three-tier (JetNet/Tuxedo)
`UT`	Utilities
`WB`	Web

To read all messages in `class`, supply `NULL` or empty string "".

`class`

Specifies the class of messages to read, where 0-7 are reserved for user-defined message classes, and the following classes, defined in `smerror.h`, are reserved for Panther:

`DM_MSGS`	`UT_MSGS`
`SM_MSGS`	`WB_MSGS`
`TP_MSGS`	

If the message file is not divided into sections, supply a value of 0.

`no_replace`
: Boolean flag. If set to other than 0 (true), an existing message set of the same class already loaded into memory is not replaced.

`msgfile`
: Specifies the message file.

Environment C only

Returns

0	Success.
1	If `no_replace` is true and the message set was already loaded.
-1	Specified file either can't be found or can't be opened.
-3	Message section not found.
-4	Specified file is not a message file.
-5	File read error or a premature end-of-file.
-6	Memory allocation error.
-7	File had an invalid version number.

Description `sm_msg_read` and its variants let you read a set of messages from a binary message file. The set of messages from the message file that is read is determined by the values of `class` and `msg_prefix`. When Panther reads messages of prefix `msg_prefix` from the file, it numbers them sequentially, starting from `class*4096`. Later, you can access these messages through `sm_msg_get` or `sm_msgfind`.

This function has three variants:

- `sm_d_msg_read` reads from the default message file specified by the environment variable `SMMSGS`.
- `sm_n_msg_read` reads from a named binary message file.
- `sm_msg_read` reads from a message file already loaded into memory.

See Also `sm_msg_del`, `sm_msg_get`, `sm_msgfind`

sm_msg_set

Replaces a message

```
#include <smerror.h>
int sm_msg_set(int msg_id, char *text);
```

`msg_id`
: Specifies the message to set using its number, as defined in `smerror.h`.

`text`
: The replacement message.

Returns

- 0 on success.
- `PR_E_MALLOC` if insufficient memory is available.

Description

`sm_msg_set` a replaces a message file entry. When a message is replaced, there is no way to restore the original message other than copying its text and then calling `sm_msg_set` again. This function first appeared in Panther 4.60.

Example

```
#include <smdefs.h>
#include <smerror.h>

    // To replace SM_5DEF_DTIME with 2 digit time tokens use:

    sm_msg_set(SM_5DEF_DTIME, "%0m/%0d/%2y %0h:%0M");

    // To restore SM_5DEF_DTIME to its value in the default msgfile:

    sm_msg_set(SM_5DEF_DTIME, "%m/%d/%2y %h:%0M");
```

See Also

`sm_msg_get`

sm_msgfind

Finds a message given its number

```
#include <smerror.h>
char *sm_msgfind(int msg_id);
```

`msg_id`
Specifies the message to get.

Returns

- A pointer to the message.
- 0 The message number is out of range.

Description

`sm_msgfind` finds the message specified by `number` and returns the message string. Unlike `sm_msg_get`, this function returns `NULL` if the message number is not found.

Message numbers for Panther messages are defined in `smerror.h`.

Example

```
#include <smdefs.h>
    #include <smerror.h>

    /* print out message #4 */

    sprintf(buf, "The message reads: %s\n", sm_msgfind
          (SM_BADKEY));
    sm_fquiet_err(0, buf);
```

See Also

`sm_msg_get`, `sm_msg_read`

sm_mts_CreateInstance

Creates an object under MTS control

```
int sm_mts_CreateInstance(char *name);
```

`name`
: Name of object to be created.

Environment
MTS

Scope
Server

Returns
- An object id suitable for making `sm_obj_call` calls.
- In case of error, `PR_E_OBJECT` will be returned and a notation with the `HRESULT` written to the log file.

Description
`sm_mts_CreateInstance` can be used in place of `sm_obj_create` to create an object that inherits the transaction context from the calling service object.

For more details see CreateInstance method under IObjectContextMethods in the MTS documentation.

Note that this function requires an object context and therefore cannot be used in the constructor of a service component.

sm_mts_CreateProperty

Creates a named property

```
int sm_mts_CreateProperty(char *group, char *prop);
```

`group`
: Name of group to be created.

`prop`
: Name of object to be created.

Environment MTS

Scope Server

Returns

0 Success. The property was created.
1 The property previously existed.
- Otherwise, `PR_E_OBJECT`. The log file will have further error information.

Description `sm_mts_CreateProperty` creates a named property within the specified group. If the group does not exist, it will be created. It is not necessary to call this function as getting or putting a value to a property will automatically create the property if necessary. See `sm_mts_GetPropertyValue` and `sm_mts_PutPropertyValue`.

The one reason to call this function is to examine the return code. It will be 1 if the property previously existed.

For more details see the CreateProperty method under ISharedPropertyGroup in the MTS documentation.

Note that this function requires an object context and therefore cannot be used in the constructor of a service component.

See Also `sm_mts_CreatePropertyGroup`, `sm_mts_GetPropertyValue`, `sm_mts_PutPropertyValue`

sm_mts_CreatePropertyGroup

Creates a new property group

```
int sm_mts_CreatePropertyGroup(char *group);
```

`group`
: Name of group to be created.

Environment MTS

Scope Server

Returns

0	Success. The group was created.
1	The group previously existed.
●	Otherwise, `PR_E_OBJECT`. The log file will have further error information.

Description `sm_mts_CreatePropertyGroup` creates a named property group. It is not necessary to call this function as getting or putting a value to a property will automatically create the group if necessary. See `sm_mts_GetPropertyValue` and `sm_mts_PutPropertyValue`.

The one reason to call this function is to examine the return code. It will be 1 if the property group previously existed.

For more details see the CreatePropertyGroup method under ISharedPropertyGroupManager in the MTS documentation.

Panther uses the following settings:

- The isolation mode is set to LockMethod. This means that the group will be locked for the entire method call. This ensures that the object can read data, and update it, without interference with other objects.
- The release mode is set to Process. Thus the data will be retained for the life of the package. However, data can only be shared within a single package.

Note that this function requires an object context and therefore cannot be used in the constructor of a service component.

See Also `sm_mts_CreateProperty`, `sm_mts_GetPropertyValue`, `sm_mts_PutPropertyValue`

sm_mts_DisableCommit

Prevents database transactions from being committed

```
int sm_mts_DisableCommit(void);
```

Environment MTS

Scope Server

Returns

- 0: OK
- `PR_E_OBJECT` if there is no context

Description `sm_mts_DisableCommit` prohibits the current transaction from being committed when all interested parties complete their work. If the database updates are not in a consistent state at some point during processing, calling this method prevents those updates from being committed.

This function will not destroy the object, nor will it cause the transaction to be aborted. Only SetAbort or SetComplete by all parties will allow the transaction to be completed.

Transactions are created with EnableCommit on.

For more details see the DisableCommit method under IObjectContextMethods in the MTS documentation.

Note that this function requires an object context and therefore cannot be used in the constructor of a service component.

See Also `sm_mts_EnableCommit`, `sm_mts_SetComplete`, `sm_mts_SetAbort`

sm_mts_EnableCommit

Enables database transactions to be committed

```
int sm_mts_EnableCommit(void);
```

Environment MTS

Scope Server

Returns

- 0: OK
- `PR_E_OBJECT` if there is no context

Description `sm_mts_EnableCommit` allows the current transaction to be committed when all interested parties complete their work.

This function will not destroy the object, nor will it cause the transaction to be committed. Only SetComplete by all parties will allow the transaction to be committed.

Transactions are created with EnableCommit on so this function is only needed if DisableCommit has been called.

For more details see the EnableCommit method under IObjectContextMethods in the MTS documentation.

Note that this function requires an object context and therefore cannot be used in the constructor of a service component.

See Also `sm_mts_DisableCommit`, `sm_mts_SetComplete`, `sm_mts_SetAbort`

sm_mts_GetPropertyValue

Gets the value of a property

```
char *sm_mts_GetPropertyValue(char *group, char *prop);
```

`group`
: Name of the property group.

`prop`
: Name of the property.

Environment	MTS
Scope	Server

Returns

- The value of the property as a string
- 0: An error occurred

Description

`sm_mts_GetPropertyValue` gets the value of the named property within the specified group. If the property or the group does not exist, it will be created.

The value of the property is always converted to a string.

For more details see the get_Value method under ISharedProperty in the MTS documentation.

Note that this function requires an object context and therefore cannot be used in the constructor of a service component.

This function stores the data in a buffer that is shared with other functions, so you must either process the returned string immediately or copy it to another variable for additional processing.

See Also `sm_mts_PutPropertyValue`, `sm_mts_CreateProperty`

sm_mts_IsCallerInRole

Determines if the client calling the component is allowed access

```
int sm_mts_IsCallerInRole(char *role);
```

`role`
Name of the user accessing the COM component.

Environment
MTS

Scope
Server

Returns
- `PV_YES` if the caller is in the role
- `PV_NO` if the caller is not
- `PR_E_OBJECT` if there is no context

Description
`sm_mts_IsCallerInRole` queries whether the current user is in the component package's specified role.

For more details see the IsCallerInRole method under IObjectContextMethods in the MTS documentation.

Note that this function requires an object context and therefore cannot be used in the constructor of a service component.

See Also
`sm_mts_IsSecurityEnabled`

sm_mts_IsInTransaction

Determines if the object is participating in a transaction

```
int sm_mts_IsInTransaction(void);
```

Environment MTS

Scope Server

Returns

- `PV_YES` if the object is in a transaction
- `PV_NO` if the object is not in a transaction
- `PR_E_OBJECT` if there is no context

Description `sm_mts_IsInTransaction` queries whether the current object is taking part in a transaction.

For more details see the IsInTransaction method under IObjectContextMethods in the MTS documentation.

Note that this function requires an object context and therefore cannot be used in the constructor of a service component.

sm_mts_IsSecurityEnabled

Determines if security checking is enabled

```
int sm_mts_IsSecurityEnabled(void);
```

Environment MTS

Scope Server

Returns

- `PV_YES` if security checking is on
- `PV_NO` if security checking is off
- `PR_E_OBJECT` if there is no context

Description `sm_mts_IsSecurityEnabled` queries whether the current object has security checking on.

For more details see the IsSecurityEnabled method under IObjectContextMethods in the MTS documentation.

Note that this function requires an object context and therefore cannot be used in the constructor of a service component.

See Also `sm_mts_IsCallerInRole`

sm_mts_PutPropertyValue

Sets value of named property

```
int sm_mts_PutPropertyValue(char *group, char *prop, char *val);
```

`group`
: Name of the property group.

`prop`
: Name of the property.

`val`
: The property's value, passed as a string.

Environment	MTS
Scope	Server
Returns	• 0: OK • `PR_E_OBJECT` if there is no context

Description

`sm_mts_PutPropertyValue` sets the value of the named property within the specified group. If the property or the group does not exist, it will be created.

For more details see the put_Value method under ISharedProperty in the MTS documentation.

Note that this function requires an object context and therefore cannot be used in the constructor of a service component.

See Also `sm_mts_GetPropertyValue`

sm_mts_SetAbort

Tells MTS to abort the transaction

```
int sm_mts_SetAbort(void);
```

Environment MTS

Scope Server

Returns

- 0: OK
- `PR_E_OBJECT` if there is no context

Description `sm_mts_SetAbort` marks the current transaction invalid. The object will be destroyed upon return to MTS from the current method.

This function can be called even if the object is not in a transaction. The effect is to cause the object to be destroyed. Thus this call says that the object is done with its work and the transaction should be rolled back (when all other parties have finished their work).

For more details see the SetAbort method under IObjectContextMethods in the MTS documentation.

Note that this function requires an object context and therefore cannot be used in the constructor of a service component.

See Also `sm_mts_SetComplete`

sm_mts_SetComplete

Informs MTS that the work is complete and ready to be committed

```
int sm_mts_SetComplete(void);
```

Environment MTS

Scope Server

Returns

- 0: OK
- `PR_E_OBJECT` if there is no context

Description `sm_mts_SetComplete` marks the current transaction complete. The object will be destroyed upon return to MTS from the current method.

This function can be called even if the object is not in a transaction. The effect is to cause the object to be destroyed. Thus this call says that the object is done with its work and the transaction may be committed (if all other parties agree).

For more details see the SetComplete method under IObjectContextMethods in the MTS documentation.

Note that this function requires an object context and therefore cannot be used in the constructor of a service component.

See Also `sm_mts_SetAbort`

sm_mus_time

Gets the system time of the last mouse click

```
double sm_mus_time(void);
```

Environment
: C only

Returns
: The system time in milliseconds.

Description
: `sm_mus_time` reports the number of milliseconds that elapsed since an unspecified time. You can compare this value to the value reported on previous or subsequent mouse clicks—for example, to determine whether two successive mouse clicks should be interpreted as a double mouse click.

See Also
: `sm_ms_inquire`

sm_mw_DismissIntroPixmap

Close the window containing the allpication’s startup image

```
void sm_mw_DismissIntroPixmap(void);
```

Description

`sm_mw_DismissIntroPixmap` closes the window displaying the image specified by the `IntroPixmap` entry in the application’s Windows initialization file.

sm_mw_get_client_wnd

Gets a handle to the client area of the MDI frame

```
#include <smmwuser.h>
HWND sm_mw_get_client_wnd(void);
```

Environment	Windows C
Returns	`HWND` of the client window.
Description	`sm_mw_get_client_wnd` gets a handle to the client area of the MDI frame of an application.

sm_mw_get_cmd_show

Returns the initial state of an application

```
#include <smmwuser.h>
int sm_mw_get_cmd_show(void);
```

Environment	Windows
Returns	`SW_SHOW`, `SW_MAXIMIZE`, `SW_MINIMIZE`
Description	`sm_mw_get_cmd_show` gets the initial state of a Windows application. Use this function to get the nCmdShow parameter of WinMain. This function returns the application's initial state.

sm_mw_get_frame_wnd

Gets a handle to the MDI frame

```
#include <smmwuser.h>
HWND sm_mw_get_frame_wnd(void);
```

Environment	Windows C
Returns	`HWND` of the frame window.
Description	`sm_mw_get_frame_wnd` gets a handle to the MDI frame of an application.

sm_mw_get_instance

Gets a handle to the current instance of a Windows program

```
#include <smmwuser.h>
HINSTANCE sm_mw_get_instance(void);
```

Environment	Windows C
Returns	A handle to the application's instance.
Description	`sm_mw_get_instance` gets a handle to the current instance of a Windows application. Use this function to supply the handle required by Windows API routines such as CreateWindow.

sm_mw_get_prev_instance

Gets a handle to the previous instance of a Windows program

```
#include <smmwuser.h>
HINSTANCE sm_mw_get_prev_instance(void);
```

Environment Windows

Returns

- A handle to the application's previous instance.
- `NULL` if there is no current instance. For WIN32 this value is always `NULL`.

Description `sm_mw_get_prev_instance` gets a handle to the previous instance of a Windows application. Use this function to supply the handle required by Windows API routines such as CreateWindow.

sm_mw_install_msg_callback

Install a message handler to be called by Panther's Windows message loop

```
void sm_mw_install_msg_callback(PiMwMsgCbFunc_t callback,
    LPVOID context);
```

`callback`
: A callback function that will be called when Panther gets a Windows message from its message queue. If `NULL` is passed, the current callback function (if any) will be uninstalled.

  When `callback` is called, it is passed two parameters, an `LPMSG` structure containing the Windows message and `context`. It should return `TRUE` if no more processing should take place on the message, else `FALSE` so that Panther can continue normal message processing.

`context`
: Information to pass when `callback` is called It can be a Window handle; a structure or some other useful pointer. If not needed, you should pass `NULL`.

Environment Windows C/C++

Description `sm_mw_install_msg_callback` installs a function that is called when Panther gets a message from its message queue. It should be used, for example, when an application opens windows outside of the Panther framework.

Example

```
#include <smdefs.h>
#include <smmwuser.h>

extern "C"
{

void OpenMyDialog()
{
    HWND hParent = GetParent(sm_mw_drawingarea());
    CMyDialog* myDialog = new CMyDialog(CWnd::FromHandle(hParent));
}

BOOL CALLBACK
MsgCallback (LPMSG pMsg, LPVOID context)
{
    if (context)                // process messages for our dialog
    {
        CMyDialog* d = (CMyDialog*)context;
        if (IsDialogMessage(d->m_hWnd, pMsg))
            return TRUE;
    }
    return FALSE;
}

CMyDialog::CMyDialog(CWnd* pParent)
                    : CDialog(CMyDialog::IDD, pParent)
{
    Create(CMyDialog::IDD, pParent);
    ShowWindow(SW_SHOW);
}

BEGIN_MESSAGE_MAP(CMyDialog, CDialog)
    //{{AFX_MSG_MAP(CMyDialog)
    ON_WM_ACTIVATE()
    //}}AFX_MSG_MAP
END_MESSAGE_MAP()

void CMyDialog::OnActivate(UINT nState, CWnd* pWndOther,
            BOOL bMinimized)
{
    if (nState == WA_INACTIVE)
    {
        sm_mw_install_msg_callback(NULL, NULL);
    }
    else
    {
        sm_mw_install_msg_callback(MsgCallback, (LPVOID)this);
    }
    CDialog::OnActivate(nState, pWndOther, bMinimized);
}

}
```

sm_mw_PrintScreen

Prints a Panther screen

```
#include <smmwuser.h>
void sm_mw_PrintScreen(int region, int interactive, int reserved1,
    int reserved2);
```

`region`

The region to be printed. The following values are from `smumisc.h`:

Region	**Value**	
`PS_CUR_SCREEN`	0	Print the top screen only.
`PS_MDI_FRAME`	1	Print the MDI frame and contents.
`PS_CUR_TITLE`	2	Print the top screen with title bar.
`PS_MDI_TITLE`	3	Print the MDI frame and contents with title bar.

`interactive`

Specifies whether the default printer will be used or whether a dialog box for user interaction will be posted:

Interaction	**Value**	
`PS_USE_DEFAULT`	0	Use the default printer.
`PS_USE_DIALOG`	1	Display the printer selection dialog box.

`reserved1`

Reserved for future use; set to 0.

`reserved2`

Reserved for future use; set to 0.

Environment Windows only.

Description `sm_mw_PrintScreen` is a control function for printing Panther screens. It sends either the current Panther screen or all the screens in the MDI frame to the printer.

sm_next_sync

Finds the next synchronized array

```
int sm_next_sync(int field_number);
```

`field_number`
Specifies the field for which a synchronized array is sought.

Returns

- The field number of the next synchronized array, if any.
- The field number the function was passed.

Description

Given a field number, `sm_next_sync` finds the next array synchronized with `field_number` and returns the field number of the corresponding element in that array. Panther identifies the next synchronized array as the one to the right, unless `field_number` is in the rightmost synchronized array. In that case, the function returns the corresponding element in the leftmost array that is synchronized with `field_number`–that is, it wraps around the screen.

sm_nl

Positions the cursor to the first unprotected field beyond the current line

```
void sm_nl(void);
```

Description

`sm_nl` moves the cursor to the next line of the screen or to the next occurrence of a scrolling array. If the current field is non-scrolling, the cursor goes to the first unprotected field, if any, on the screen's next line. If all fields below the current one are protected, the cursor wraps to the screen's first unprotected field.

If the cursor is on the last allocated occurrence of a scrolling array and the number of allocated occurrences is less than the maximum, Panther allocates an empty occurrence.

If all fields are protected, the cursor goes to the first column of the next line. If the cursor is on the screen's last line of the form, it wraps to the screen's top left-hand corner (0,0).

`sm_nl` does not immediately trigger field entry, exit, or validation processing. Such processing occurs according to the cursor position when control returns to `sm_input`.

This function is usually bound to NL.

See Also

`sm_backtab`, `sm_home`, `sm_last`, `sm_tab`

sm_*null

Tests whether a field is null

```
int sm_null(int field_number);
int sm_e_null(char *field_name, int element);
int sm_i_null(char *field_name, int occurrence);
int sm_n_null(char *field_name);
int sm_o_null(int field_number, int occurrence);
```

`field_name, field_number`
: Specifies the field to test.

`element`
: The element in `field_name` to test.

`occurrence`
: The occurrence in the specified field to test.

Returns

1	True: the field’s Null Field property is set to Yes and contains a null value.
0	False: the field’s Null Field property is set to No or it does not contain a null value.
-1	The field does not exist.

Description

Use `sm_null` to test whether a field’s value is null or not. This function checks whether a field’s `null_field` property is set to `PV_YES`; if it is, `sm_null` gets the field’s null indicator and compares it to the field’s value.

You can specify the field’s null indicator string through the message file and/or the field’s Null Text property.

sm_obj_call

Calls a method of a service component, Java object or COM control

```
char *sm_obj_call(char *method_spec);
```

`method_spec`
: A string specifying the method and its parameters consisting of the following:

  `object_id`
  : An integer handle identifying the component whose method you want to call. Object handles are returned by `sm_obj_create` for component objects, `sm_prop_id` for ActiveX controls and by `sm_obj_call`.

  `method`
  : The name of the method. Periods are allowed as part of the method specification, as in:

    ```
    Application.Quit
    ```

  `p1, p2, ...`
  : (Optional) A comma-delimited list of the method's parameters. Unused parameters can be omitted, as in:

    ```
    sm_obj_call ("TreeView, \"Add\" , , , , 'First node'")
    ```

Environment COM, EJB, Java

Scope Client

Returns

- The value returned by the component, converted to a string.
- A null string if an error occurred. For a COM error code, call `sm_com_result`. COM error codes are defined in `winerror.h`.

Description `sm_obj_call` calls methods that are part of the component's interfaces. To find which methods are available, refer to the documentation supplied with COM component, use the Panther AxView utility, or use the View⇒Component Interface in the Panther Editor for service components.

This function returns a string; the component itself can return different types of data.

Java Objects

For calling methods of Java objects, the method name can include an optional type-specifier to eliminate ambiguity for overloaded methods (see Working with Java Objects for further information on using type-specifiers.) `@obj()` may be used to pass in Java objects as parameters for the Java method. For primitive and String valued method return types, this function returns the value as a string. Otherwise, a Panther object ID is returned. This object ID should be passed to `sm_obj_delete_id` when the associated Java object is no longer needed in order to allow for garbage collection by the JVM.

COM Components

For COM components, if the typelib cannot be used to determine the parameter's type, `@obj()` can be used to specify the object ID of the parameter. Generally, this syntax will not be necessary. For an example of its usage, see the example under `sm_com_load_picture`.

If you get a "type mismatch" error, refer to the component documentation and check that all the parameters are of the correct type. `@obj()` may be needed if any of the parameters must be passed as objects.

Syntax Changes in JPL, Java and C

The syntax of `sm_obj_call` is different in JPL from that in C and Java. For JPL, `sm_obj_call` can have multiple parameters. For Java and C, `sm_obj_call` accepts one parameter, a string, which Panther parses into multiple parameters. See the *Examples* section.

Examples

```
// This C function calls the InsertNode method of the
// ActiveX treeview control.

char *parent;
char *child;

child = sm_obj_call("treeview->id, \"InsertNode\", parent, \"Child
node\"");

// This Java example calls the SetStyle method.

import com.prolifics.jni.*;
public class SetStyleButton extends ButtonHandlerAdapter
{
public int buttonValidate(FieldInterface f, int item, int context)
    {
    ScreenInterface scrn = f.getScreen();
    WidgetInterface w = scrn.getWidget("tree");
    CFunctionsInterface cfi = w.getCFunctions();
    String s = cfi.sm_obj_call("tree->id, \"SetStyle\", 1, 1, 1,
1");
          return 0;
     }
}

// This is the JPL call for this method. Single quotation
// marks are used surrounding the method in order to pass
// double quotation marks to the method itself.

vars parent
vars child

child = sm_obj_call \
    (treeview->id, "InsertNode", :parent, "Child node")

// These JPL procedures instantiate the cCustomers COM
// component and call its GetCustomer method.

vars id

proc entry
@app->current_component_system=PV_SERVER_COM
id = sm_obj_create("cCustomers")
return

proc GetCustomer
call sm_obj_call(id, "GetCustomer", \
    CompanyName, CustomerID, Phone)
return

// This JPL procedure closes down Microsoft Excel
// that is running as a COM component.

proc close
call sm_obj_call(ExcelID, "Application.Quit")
return
```

See Also sm_obj_get_property, sm_obj_set_property

sm_obj_copy*

Copies a widget

```
#include <smuprapi.h>
int sm_obj_copy(char *target_widget, char *source_widget);
int sm_obj_copy_id(int target_widget_id, int source_widget_id);
```

`target_widget, target_widget_id`
: The widget to receive the copied widget, specified either by name or by an integer handle obtained from `sm_prop_id`. It can specify a screen, a box, a tab card, a tab deck or a grid.

`source_widget, source_widget_id`
: The widget to copy, specified either by its name or by an integer handle obtained from `sm_prop_id`. The widget to copy can be on any screen on the window stack. If the widget is not on the current screen, supply its integer handle; or use the JPL object syntax to specify the source screen. For example, supply this string to copy `cust_id` from the `custqry` screen:

  `@screen("custqry.scr")!cust_id.`

Returns

- ≥1 Object ID of the new widget.
- `PR_E_MALLOC`: Insufficient memory available.
- `PR_E_OBJID`: ID for source widget or target screen does not exist.
- `PR_E_OBJECT`: Named object does not exist.
- `PR_E_OBJ_TYP`: The source widget cannot be copied into the target widget because their widget types are not compatible. For example, only tab cards can be copied into tab decks.
- `PR_E_TOO_BIG`: Widget cannot fit on the target screen.

Description

`sm_obj_copy` creates a copy of the specified widget and puts it in the target widget. The data and all properties of the source widget are copied to the new one, including its position on the screen. If the widget is copied onto the screen of the source widget, the new widget overlays the original. If a widget is copied to a box or to a tab card it will be moved if necessary.

If the source widget is named and the target screen already has a widget with the same name, Panther sets the new widget's name to an empty string to prevent duplicate names.

Copying Groups sm_obj_copy can also copy a synchronized scrolling group or table view group; the function copies an empty group to the target screen—that is, the member widgets are not copied. You can subsequently copy one or more members of the group through additional calls to sm_obj_copy.

Selection groups cannot be copied directly; however if you copy a field that belongs to a selection group to another screen, Panther copies the field and its group to the target screen, provided that the target screen does not already contain a group of the same name; if it does, the copied field is added to the existing group.

See Also sm_obj_delete_id

sm_obj_create

Instantiating an object

```
int sm_obj_create(char *object);
```

`object`
The service component to instantiate.

Environment
COM, EJB, Java

Returns

- Success: object ID
  `PR_E_ERROR`: `@app()->current_component_system` is not set
  `PR_E_OBJECT`: the service component could not be created.

Description
`sm_obj_create` instantiates a service component regardless of whether it is deployed using COM, EJB or Java technologies. Before invoking this function, you must set the `current_component_system` application property in order to select the type of components to create: `PV_SERVER_COM` for COM components, `PV_SERVER_EJB` for Enterprise JavaBeans deployed in a JEE Application Server, such as IBM's WebSphere Application Server, or `PV_SERVER_JAVA` for Java Objects.

The argument to `sm_obj_create` may contain a comma separated list of parameters that is specific to the component system being used. When `sm_obj_create` is called from JPL, these embedded parameters should be passed as separate parameters to `sm_obj_create`.

See Also
`sm_obj_delete_id`

sm_obj_create_licensed

Instantiating a licensed object

```
int sm_obj_create_licensed(char *object, char *license);
```

`object`
: The service component to instantiate.

`license`
: The license needed to instantiate the service component.

Environment
: COM

Returns

- Success: object ID
  `PR_E_ERROR`: @app()->current_component_system is not `PV_SERVER_COM`
  `PR_E_OBJECT`: The service component could not be created

Description
: `sm_obj_create_licensed` instantiates a licensed COM service component. Before invoking this function, you must set `@app()->current_component_system` to `PV_SERVER_COM`. This function first appeared in Panther 5.10.

Example

```
vars objId, license

license = '0FA372A815960ED56DE037A'

objId = sm_obj_create_licensed("FAXOCX.FaxManCtrl.1", license)
```

See Also
: `sm_obj_delete_id`

sm_obj_create_server

Instantiate a COM object

```
int sm_obj_create_server(char *object);
```

`object`
The service component to instantiate.

Environment
COM

Returns

- Success: Object ID

`PR_E_ERROR`: `@app()->current_component_system` is not `PV_SERVER_COM`
`PR_E_OBJECT`: the service component could not be created

Description
`sm_obj_create_server` instantiates a COM service component. It is needed because some COM components cannot be instantiated by `sm_obj_create`. A notable example is the Crystal Reports `CrystalRuntime.Application` component. Before invoking this function, you must set `@app()->current_component_system` to `PV_SERVER_COM`. This function was first released in Panther 5.10.

Example

```
// This JPL code assmes that the screen has a Crystal Reports
// 'Crystal ActiveX Report Viewer Control 11.5' ActiveX field
// named 'myReport'.

vars reportObj, applicObj

@app()->current_component_system = PV_SERVER_COM

applicObj = sm_obj_create_server("CrystalRuntime.Application")

// reportPfad is the file name of a Crystal Reports report file.

proc ViewReport (reportPfad)
{
  if applicObj > 0
  {
    reportObj = sm_obj_call(applicObj, "OpenReport", reportPfad)

    if reportObj > 0
    {
      sm_obj_set_property(myReport->id, "ReportSource", reportObj)
```

```
            sm_obj_call (myReport->id, "ViewReport")

            sm_obj_delete_id (reportObj)
        }
    }
}
```

See Also `sm_obj_delete*`

sm_obj_delete*

Deletes an object

```
int sm_obj_delete(char *object);
int sm_obj_delete_id(int object_id);
```

`object, object_id`
: The object (a widget or a component) to delete, specified either by its name or by an integer handle obtained from `sm_prop_id` for widgets, `sm_obj_create` for components, or from `sm_obj_call`.

Returns

0: Success.
`PR_E_OBJID`: ID for source widget or target screen does not exist.
`PR_E_OBJECT`: Named object does not exist.

Description `sm_obj_delete` and `sm_obj_delete_id` delete objects.

Deleting Widgets The widget to delete can be on any screen on the window stack. If the widget is not on the current screen, supply its integer handle; or use the JPL object syntax to specify the source screen. For example, this JPL statement deletes `cust_id` from the `custqry` screen:

```
call sm_obj_delete("@screen('custqry.scr')!cust_id")
```

Note: This function has no effect on the screen definition; to restore deleted widgets, close and reopen the screen.

Destroying Components After invoking and working with the methods and properties of a component, you should destroy it by calling `sm_obj_delete_id` with the component's object ID. Otherwise, the component will continue to exist until the application terminates (or goes from test mode to edit mode).

Java objects returned by calls by `sm_obj_create` and to `sm_obj_call` should be deleted by `sm_obj_delete_id` to allow garbage collection of these objects by the JVM. (This does not happen automatically when switching from test to edit mode.)

If a COM component is running under MTS, its life cycle can be managed by MTS, depending on whether the component is marked as belonging to a transaction and whether the work in the transaction is complete.

```
call sm_obj_delete_id(cmpt_id)
```

See Also   sm_obj_copy, sm_obj_create

sm_obj_get_property

Gets the value of a property from a service component or ActiveX control

```
char *sm_obj_get_property(int obj_id, char *prop);
```

`obj_id`

An integer handle that identifies the component whose property you want to get. Object handles are returned by `sm_obj_create` for service components, `sm_prop_id` for ActiveX controls and by `sm_obj_call`.

.

`prop`

The designated property. For indexed properties, use brackets to specify the occurrence.

Environment COM, EJB, Java

Scope Client

Returns

- The property's current value, returned as a string.
- A null string if an error occurred. For the error code from COM components, call `sm_com_result`. Error codes are defined in `winerror.h`.

Description `sm_obj_get_property` retrieves property values from a service component or ActiveX control.

Properties can be determined through the AxView utility for COM components, the Properties window for ActiveX controls, or the Component Interface window for service components.

Example

```
#include <smuprapi.h>
{
    id = sm_prop_id("spinner");
    value = sm_obj_get_property(id, "prop");
}

// For an indexed property:
{
    id = sm_prop_id("spinner");
    value = sm_obj_get_property(id, "prop[5]");
}
```

See Also `sm_obj_set_property`

sm_obj_onerror

Installs an error handler

```
void sm_obj_onerror(char *handler);
```

`handler`
: The name of the error handler. This C function or JPL procedure will be passed three parameters:

  `errorNumber`
  : The error number as an integer. Use this value to test for errors.

  `errorHexidecimal`
  : The error number as a string in hexadecimal format, as in `0x80000307`. (This parameter is displayed by the default error handler.)

  `errorMessage`
  : The text description of the error. (This parameter is displayed by the default error handler.)

Environment COM, EJB, Java

Scope Client

Description `sm_obj_onerror` specifies the error handler that will be called if an operation returns a negative exception.

An error handler will only be fired on negative exception codes; for COM/MTS applications, use `sm_com_result` to retrieve positive exceptions.

If you do not want an error handler, you must install your own error handler that simple returns.

To restore the default error handler, use `sm_obj_onerror("")`.

Example

```
// This JPL module specifies an error handler
   // similar to the default error handler.

call sm_obj_onerror (handler)

proc handler (errnum, errhex, errmsg)
```

```
msg emsg "COM Error: " errhex " " errmsg
return
```

sm_obj_set_property

Sets the value of a property of a service component or ActiveX control

```
int sm_obj_set_property(int obj_id, char *prop, char *val);
```

`obj_id`
: An integer handle that identifies the object whose property you want to set. The handle is returned through `sm_obj_create` for service components, `sm_obj_create` or `sm_obj_call` for Java Objects, and `sm_prop_id` for ActiveX controls.

`prop`
: The designated property. Refer to the Description for information about available properties.

`val`
: The value to set for the specified property or property item. Panther converts the value to the type expected by the component.

If the value needs to be sent as an object ID, `@obj()` can be used to specify the object ID. For an example of its usage, see the example under `sm_com_load_picture`.

Environment COM, EJB, Java

Scope Client

Returns On Windows platforms:

- The `HRESULT` from the most recent component function call. Refer to `winerror.h` for values; 0 is the value for `S_OK`.

On UNIX platforms:

0	Success.
-1	Failure.

Description `sm_obj_set_property` sets the value of the specified property of a service component or ActiveX control.

When setting properties for ActiveX controls, this function can be used on all platforms for CLSID and Control Name and only on Windows for properties of the ActiveX control itself.

Properties can be determined through the AxView utility for COM components, the Properties window for ActiveX controls, or the Component Interface window for service components.

In property specifications, periods are allowed. For example:

```
sm_obj_set_property(Excel_Sheet,
    "ActiveSheet.Cells(1,1).Value", text)
```

For indexed properties, use brackets to specify the occurrence. For example:

```
sm_obj_set_property(id, "prop[5]", value)
```

Example

```
#include <smuprapi.h>
    int id;
    int retcode;

{
    id = sm_prop_id("spinner");
    retcode = sm_obj_set_property(id, "Value", "40");
    }
```

See Also

`sm_obj_get_property`

sm_obj_sort

Sorts the object's occurrences

```
int sm_obj_sort(int obj_id, int direction);
```

`obj_id`
: An integer handle that identifies the object to be sorted. For widgets, it can be obtained from `sm_prop_id`.

`direction`
: The direction for the sort: `SORT_ASCENDING` for an ascending sort or `SORT_DESCENDING` for a descending sort.

Returns

- 0: Success. Target has been sorted without error.
  `PR_E_OBJECT`: Cannot find target object.
  `PR_E_MALLOC`: Memory error.
  `PR_E_NO_SORT_FUNC`: Sort order function not specified.
  `PR_E_SORT_FUNC`: Sort order function not found, or error reported by sort order function.
  `PR_E_DATA_FORMAT`: Incompatible data format for sort order.
  `PR_E_ERROR`: Error in performing sort.
- A property API error code.

Description

`sm_obj_sort` sorts the object's occurrences according to the rules specified in the object's Sort Order property. If the Sort Order is set to `PV_CUSTOM`, then the function named in the Sort Order Function property is used.

If the specified object is a member of a synchronized scrolling group, the other arrays in the group will be re-ordered to retain row integrity for the group.

See Also

`sm_obj_sort_auto`

sm_obj_sort_auto

Sorts the object's occurrences according to grid conventions

```
int sm_obj_sort_auto(int obj_id);
```

`obj_id`
: An integer handle that identifies the object to be sorted. For widgets, it can be obtained from `sm_prop_id`.

Returns

0: Success. Target has been sorted without error.

`PR_E_OBJECT`: Cannot find target object.
`PR_E_MALLOC`: Memory error.
`PR_E_NO_SORT_FUNC`: Sort order function not specified.
`PR_E_SORT_FUNC`: Sort order function not found, or error reported by sort order function.
`PR_E_DATA_FORMAT`: Incompatible data format for sort order.
`PR_E_ERROR`: Error in performing sort.

- A property API error code.

Description

`sm_obj_sort_auto` sorts the object's occurrences according to the rules specified for grids in the Windows API. For fields that have their Column Click Action property set to `PV_SORT`, this function is invoked automatically in response to user clicks on the field's grid column heading.

What happens in response to the invocation of `sm_obj_sort_auto` on a given field depends on the settings of the field's runtime properties `column_arrow_direction` and `column_arrow_hidden`.

If `column_arrow_hidden` is set to `PV_NO`, the value of `column_arrow_direction` will be flipped from `PV_UP` to `PV_DOWN`, or vice versa, and the object will be sorted according to the new `column_arrow_direction` value.

If `column_arrow_hidden` is set to `PV_YES`, then the field is sorted according to the current value of `column_arrow_direction`, and then the value of `column_arrow_hidden` is changed to `PV_NO`.

Note that these column arrow properties are in effect, and are manipulated by `sm_obj_sort_auto`, even when an object is not in a grid and when there is no visible representation of the arrow on the screen.

Access to `sm_obj_sort_auto` is provided if you want to invoke it in the context of custom `column_click_func` functions. `sm_obj_sort_auto` should either be called in response to column click events or during screen entry processing, so that the first time a user sees a grid, it is already sorted. In general, `sm_obj_sort_auto` is not useful except for objects in grids. For general-purpose sorting of objects and synchronization groups, use `sm_obj_sort`.

See Also `sm_obj_sort`

sm_occur_no

Gets the current occurrence number

```
int sm_occur_no(void);
```

Returns

≥1	The occurrence number.
0	The cursor is not in a field.

Description `sm_occur_no` returns the number of the occurrence in the current field.

sm_*off_gofield

Moves the cursor into a field, offset from the left

```
int sm_off_gofield(int field_number, int offset);
int sm_e_off_gofield(char *field_name, int element, int offset);
int sm_i_off_gofield(char *field_name, int occurrence,
   int offset);
int sm_n_off_gofield(char *field_name, int offset);
int sm_o_off_gofield(int field_number, int occurrence,
   int offset);
```

`field_name, field_number`
: Specifies the destination field.

`element`
: The destination element in `field_name`.

`occurrence`
: The destination occurrence in the specified field.

`offset`
: The position in the destination field at which to place the cursor. If `offset` is larger than the field's length, or greater than a shiftable field's maximum length, the cursor is placed in the rightmost position.

Returns

0	Success.
-1	The field is not found.

Description

`sm_off_gofield` moves the cursor into the specified field at position `offset`, regardless of the field's justification. If the data specified by `offset` is out of view, Panther shifts the field's contents to make the data visible.

Example

```
#include <smdefs.h>
   #include <ctype.h>
   /* Place cursor over the first embedded blank in */
   /* the "names" field.
    */

char buf[256], *p;
   int length;

   length = sm_n_getfield(buf, "names");
```

```
for (p = buf; p <buf + length; ++p)
{
   if (isspace(*p))
      break;
}
sm_n_off_gofield("names", p - buf);
```

See Also `sm_disp_off`, `sm_gofield`, `sm_sh_off`

sm_option

Sets a behavior variable

```
int sm_option(int option, int newval);
```

`option`
: The behavior variable to change.

`newval`
: The new value, defined in `smsetup.h`, to assign the `option`-specified option. To get an option's current value, supply the value `NOCHANGE`.

Returns

- The old value for the specified option.
- −1: The option is out of range.

Description

`sm_option` lets you change Panther behavior variables at runtime—for example, error window attributes, delayed write options, cursor display, and zoom options. You can set one of these variables:

`CHAR_VAL_OPT`	`IN_VARROW`
`CLOSELAST_OPT`	`IN_WRAP`
`DA_CENTBREAK`	`IND_OPTIONS`
`DECIMAL_PLACES`	`IND_PLACEMENT`
`EMSGATT`	`LISTBOX_SELECTION`
`ENTEXT_OPTION`	`MESSAGE_WINDOW`
`ER_ACK_KEY`	`OCTAL_SUPPORT`
`ER_KEYUSE`	`QUIETATT`
`ER_SP_WIND`	`SB_OPTIONS`
`EXPHIDE_OPTION`	`SCR_KEY_OPT`
`F_EXTREC`	`SMSGBKATT`
`F_EXTOPT`	`SMSGPOS`
`F_EXTSEP`	`STEXTATT`
`FCASE`	`TOOLBAR_DISPLAY`
`GA_CURATT`	`TOOLTIP_DISPLAY`
`GA_CURMASK`	`TXT_SELECT_ATTR`
`GA_SELATT`	`TXT_SELECT_MASK`
`GA_SELMASK`	`ZM_DISPLAY`
`IN_ENDCHAR`	`ZM_SC_OPTIONS`
`IN_HARROW`	`ZM_SH_OPTIONS`
`IN_RESET`	`WWTAB`
`IN_VALID`	`XMIT_LAST`

Note: Use `sm_keyoption` to change the behavior of cursor control keys.

See Also

`sm_keyoption`, `sm_soption`

sm_optmnu_id

Gets the ID of an option menu or combo box

```
int sm_optmnu_id(void);
```

Returns

- An integer handle that uniquely identifies an option menu or combo box.
- `PR_NULL_OBJID`: Unable to identify an option menu or combo box.

Description

`sm_optmnu_id` gets the object ID property of an option menu or combo box that is initialized on popup from an external screen (`initialization` = `PV_FILL_AT_POPUP`); this function can only be called by the external screen's entry function; otherwise, it returns `PR_NULL_OBJID`. (For more information about initializing option menu data, refer to "Using Data from an External Source" on page 14-22 in *Using the Editors*.)

For example, you might have two option menus that are initialized from the same external screen but require different sets of data. The external screen's entry function can call `sm_optmnu_id` to get the ID of its caller and thereby determine which database query fetches the required data:

```
/* get the option menu's ID */
vars opt_id
opt_id = sm_optmnu_id()

dbms declare cursor c1
dbms with cursor c1 alias array1

/* query the database according to option menu name */
if @id(opt_id)->name == "ratings_opt"
{
  dbms declare cursor c1 \
       select rating_code from titles \
       group by rating_code order by 1
}
else if @id(opt_id)->name == "genre_opt"
{
  dbms declare cursor c1 \
       select descr from codes \
       where code_type = 'genre_code' order by 1
}
```

```
dbms with cursor c1 execute
dbms close cursor c1
return
```

sm_*PiMwCopyToClipboard

Copy data from field(s) to the Windows clipboard

```
#include <smmwuser.h>
int sm_n_PiMwCopyToClipboard(const char *fields);
int sm_i_PiMwCopyToClipboard(const char *fields, int from);
int sm_ii_PiMwCopyToClipboard(const char *fields, int from,
    int to);
```

`fields`
: A comma separated list of one or more fields. If copying from a word wrapped field, only one field can be specified. Grids and synchronized scrolling groups can also be specified, in which case, data from all the grid or group members is copied.

`from`
: For a word wrapped field, the starting character; otherwise the starting occurrence.

`to`
: For a word wrapped field, the ending character - `0` (zero) to copy the remaining characters; otherwise the ending occurrence.

Environment Windows interactive. First released in Panther 5.40.

Returns

- `0`: Success.
- `PR_E_ARGS`: problems parsing argument fields or in the values of from or to.
- `PR_E_ERROR`: Unable to access the clipboard or field data.
- `PR_E_MALLOC`: Memory allocation error.

Description For a word wrapped field, the data is just copied, including any new line and tab characters in the field data.

Otherwise, the field data is tab separated with each occurrence being on a separate line. This allows the data to be pasted, for example, into a spreadsheet.

See Also `sm_*PiMwPasteFromClipboard`

sm_*PiMwPasteFromClipboard

Paste data from the Windows clipboard to field(s)

```
#include <smmwuser.h>
int sm_n_PiMwPasteFromClipboard(const char *fields);
int sm_i_PiMwPasteFromClipboard(const char *fields, int from);
int sm_ii_PiMwPasteFromClipboard(const char *fields, int from,
   int to);
```

`fields`

A comma separated list of one or more fields. When pasting to a word wrapped field, only one field can be specified. Grids and synchronized scrolling groups can also be specified, in which case, data is pasted to all the grid or group members.

`from`

For a word wrapped field, the starting character to paste to; otherwise the starting occurrence.

`to`

For a word wrapped field, the ending character - `0` (zero) to replace the remaining characters; otherwise the ending occurrence.

Environment Windows interactive. First released in Panther 5.40.

Returns

- `0`: Success.
- `PR_E_ARGS`: problems parsing argument fields or in the values of from or to.
- `PR_E_ERROR`: Unable to access the clipboard or field data.
- `PR_E_MALLOC`: Memory allocation error.

Description For a word wrapped field, the data is just pasted, including any new line and tab characters in the clipboard data.

Otherwise, the clipboard data is assumed to be tab separated with each occurrence being on a separate line. This allows, for example, pasting data copied to the clipboard from a spreadsheet.

See Also `sm_*PiMwCopyToClipboard`

sm_pinquire

Gets the value of a global string

```
#include <smglobs.h>
char *sm_pinquire(int which);
```

`which`
: Specifies the global string to get through one of these constants:

`P_YES`
: Returns valid affirmative input for a field whose `keystroke_filter` property is set to `PV_YES_NO`. The return is a null-terminated string that contains the lowercase yes value and the uppercase yes value.

`P_NO`
: Returns valid negative input for a field whose `keystroke_filter` property is set to `PV_YES_NO`. The return is a null-terminated string that contains the lowercase no value and the uppercase no value.

`P_DECIMAL`
: Returns a three-character string: the user's decimal point marker, the operating system's decimal point marker, and the null terminator.

`P_DICNAME`
: Returns the repository's file name.

`P_FLDPTRS`
: Returns a pointer to an array of field structures. The implementation of these structures is release-dependent.

`P_TERM`
: Returns the name Panther uses as the terminal identifier, or an empty string if not found.

`P_SPMASK`
: Returns a pointer to a memory-resident, full-size form containing all blanks.

`P_USER`
: Returns a pointer to developer-specified region of memory for the current screen. Each screen maintains its own pointer. This pointer is not set by Panther; it is set and maintained by the application.

`SP_NAME`
: Returns the name of the active screen.

`SP_STATLINE`
: Returns the status line's current text.

`SP_STATATTR`
: Returns attributes of current status line—a pointer to an array of unsigned short integers.

`V_`
: One of the `V_` constants defined in `smvideo.h`, returns video-related information.

Returns

- If the argument corresponds to a global pointer variable, a pointer to the value of that variable.
- 0: Failure.

Description

`sm_pinquire` gets the current value of a global pointer variable. To modify a global string, use `sm_pset`.

Because the objects pointed to by the pointers returned by `sm_pinquire` usually have short duration, use or copy them quickly. This caution does not apply to `P_USER`, which is maintained by the application. The `P_` pointers point to the actual objects in Panther. The `SP_` pointers point to copies of the objects. Because an object's characteristics is implementation dependent, it might change in future releases of Panther. Except for `P_USER`, do not use the pointers returned by `sm_pinquire` to modify objects directly. Use `sm_pset` instead.

Example

```
/* Get next key from user. Return -1 for 'n', 1 for 'y', and
     * 0 if unknown. 'n' and 'y' come from the message file,
     * and so can be changed to reflect the local language.
     */

int get_yes_no()
    {
       unsigned key;
       char *yes;
       char *no;
       key = sm_getkey();
       yes = sm_pinquire(P_YES);
       no = sm_pinquire(P_NO);
       if (key == yes[0] || key == yes[1])
          return(1);
       if (key == no[0] || key == no[1])
```

```
            return(-1);
        return(0);
    }
```

See Also sm_inquire, sm_iset, sm_pset

sm_popup_at_cur

Invokes the current widget's popup menu

```
int sm_popup_at_cur(void);
```

Returns

0	`MNERR_OK`: Success.
-3	`MNERR_NOT_SUPPORTED`: Menu bars are not supported.

Description

`sm_popup_at_cur` invokes the popup menu installed for the field or screen, depending on which one has focus. This function lets users access popup menus via the keyboard. For example, the following control string assignment lets a user invoke a popup menu by pressing the `PF1` key:

```
PF1 = ^sm_popup_at_cur
```

`sm_popup_at_cur` uses one of the following two algorithms for finding and displaying a popup menu:

- If a field has focus, `sm_popup_at_cur` displays the first menu that it finds from the following:
  1. The field's popup menu.
  2. The screen's popup menu.
  3. The menu installed for the screen's menu bar and toolbar.
  4. The application-level menu.
- If the screen has focus, `sm_popup_at_cur` displays the first menu that it finds from the following:
  1. The screen's popup menu.
  2. The menu installed for the screen's menu bar and toolbar.
  3. The application-level menu.

See Also

`sm_menu_install`

sm_prop_error

Gets the error code returned by the last properties API function call

```
#include <smuprapi.h>
int sm_prop_error(void);
```

Returns

- 0: The last function call succeeded.
- `PR_E_ERROR`: Failed for another reason.
- `PR_E_MALLOC`: Insufficient memory.
- `PR_E_OBJID`: Object ID does not exist.
- `PR_E_OBJECT`: Object does not exist.
- `PR_E_ITEM`: Invalid occurrence or element.
- `PR_E_PROP`: Invalid property.
- `PR_E_PROP_ITEM`: Invalid property item.
- `PR_E_PROP_VAL`: Invalid property value.
- `PR_E_CONVERT`: Unable to perform conversion.
- `PR_E_OBJ_TYPE`: Invalid object type.
- `PR_E_RANGE`: Property value is out of range.
- `PR_E_NO_SET`: Property cannot be set.
- `PR_E_BEYOND_SCREEN`: Widget extends beyond screen.
- `PR_E_WW_SCROLLING`: Word wrap must be scrolling.
- `PR_E_NO_SYNC`: Arrays cannot be synchronized.
- `PR_E_TOO_BIG`: Widget too large for screen.
- `PR_E_BAD_MASK`: Invalid edit mask or regular expression
- `PR_E_NO_KEYSTRUCT`: Property requires previous execution of `SELECT`, `NEW`, `COPY`, or `COPY_FOR_UPDATE` command.

Description

`sm_prop_error` gets the error code returned by the last-called properties API function: `sm_prop_get`, `sm_prop_set`, `sm_prop_id`, or one of their variants. This function is especially useful for ascertaining the success or failure of calls to variants that do not return an error code—for example, `sm_prop_get_str`, which returns 0 when an error occurs.

Because Panther internal processing also uses the properties API, you should call this function and retrieve the desired error code immediately.

Note: A negative value returned by `sm_prop_get_int` and its variants usually specifies an error. However, some integer properties accept negative values; in these cases, you can differentiate between a negative property value and an error condition only by calling `sm_prop_error`.

Example

```
/* Act on error code */

switch (sm_prop_error())
    {
        case '0':
            ...
            break;
        case PR_E_ERROR:
            ...
            break;
        case PR_E_MALLOC:
            ...
            break;
        ...
        default:
            ...
    }
```

sm_prop_get*

Gets a property setting

```
#include <smuprapi.h>
int sm_prop_get_int(int obj_id, int prop);
char *sm_prop_get_str(int obj_id, int prop);
double sm_prop_get_dbl(int obj_id, int prop);
int sm_prop_get_x_int(int obj_id, int array_item, int prop);
char *sm_prop_get_x_str(int obj_id, int array_item, int prop);
double sm_prop_get_x_dbl(int obj_id, int array_item, int prop);
int sm_prop_get_m_int(int obj_id, int prop, int prop_item);
char *sm_prop_get_m_str(int obj_id, int prop, int prop_item);
double sm_prop_get_m_dbl(int obj_id, int prop, int prop_item);
```

`obj_id`
: An integer handle that identifies the Panther object whose property you want to get, obtained through `sm_prop_id`. For application properties, supply `PR_APPLICATION`; for the current screen, `PR_CURSCREEN`.

`array_item`
: The widget occurrence or element whose property you want to get.

`prop`
: The property to get. Refer to Chapter 1, “Runtime Properties,” in *Quick Reference* for a full list of property constants.

`prop_item`
: Specifies the item in a multi-item property whose value you want to get. For example, if the `prop` value is `SM_PR_CONTROL_STRING`, supply a logical key name such as XMIT to get that key’s current control string assignment.

Returns

For `sm_prop_get_int` and its variants:

- The property’s current value, returned as an integer
- <0 The property’s negative value or the error code returned by this function. To ascertain whether an error condition exists, call `sm_prop_error`.

For `sm_prop_get_str`, `sm_prop_get_dbl`, and their variants:

- The property’s current value, returned either as a string pointer or a double.

0 Failure. To ascertain the cause of failure, call sm_prop_error.

Description

sm_prop_get has three basic variants: sm_prop_get_str, sm_prop_get_int and sm_prop_get_dbl, which get string, integer, and double properties, respectively. For example, sm_prop_get_str gets string properties such as title, while sm_prop_get_int gets integer properties such as max_occurrences.

sm_prop_get_str stores the returned data in a pool of buffers that it shares with other functions; either process the returned string immediately or copy it to another variable for additional processing.

Each of these variants have _x and _m variants. These let you access properties of occurrences or elements, and offsets into properties that take multiple values, respectively. These variant types are discussed in the following sections.

Elements and Occurrences

You can get properties for individual elements and occurrences in an array by calling sm_prop_get_x_*prop-type*. All variants of this function require an obj_id handle to the array and an array_item argument. Depending on how the obj_id handle was obtained, the function determines whether array_item specifies an offset into the array's elements or its occurrences:

- To set the properties of an array's elements, obtain a handle by supplying sm_prop_id with a widget identifier that has the format *widget-spec*[[]].
- To set the properties of an array's occurrences, obtain a handle by supplying sm_prop_id with a widget identifier that has the format *widget-spec*[].

For example, this call to sm_prop_id gets a handle to the properties of cust_id's elements:

```
int elem_h;
elem_h = sm_prop_id("cust_id[[]]");
```

This call gets a handle to the properties of cust_id's occurrences:

```
int occ_h;
occ_h = sm_prop_id("cust_id[]");
```

Given these two handles, you can use sm_prop_get_x_int to get the mdt property setting for either cust_id's first element or first occurrence as follows:

```
/* get the first element's mdt setting */
int elem_mdt;
elem_mdt = sm_prop_get_x_int(elem_h, 1, PR_MDT);
```

```
/* get the first occurrence's mdt setting */
int occ_mdt;
occ_mdt = sm_prop_get_x int(occ_h, 1, PR_MDT);
```

Multi-item Properties

`sm_prop_get_m_`*`prop-type`* gets one of the settings in a multi-item property such as `PR_DROP_DOWN_DATA` for an option menu, or `PR_CONTROL_STRING` for a screen. For example, this code iteratively calls `sm_prop_get_m_str` to compare the data in each item in option menu `flavors` to the current selection:

```
/* replace current item with contents of "substitute" */
char cur_item[256], new_item[256];
char *option_txt;
int ct, f_id, err;

f_id = sm_prop_id("flavors");

/*get substitute data*/
sm_n_getfield("substitute", new_item);

/*get selection data*/
sm_n_getfield("flavors", cur_item);

/* get offset of current selection */
for (ct = 1; ; ct++)
{
   option_txt = sm_prop_get_m_str(f_id,
                       PR_DROP_DOWN_DATA, ct)
   if (!option_txt)
   {
      err = PR_E_ERROR;
      break;
   }
   if (strcmp(option_txt, cur_item) == 0)
   {
      err = sm_prop_set_m_str(f_id,
                   PR_DROP_DOWN_DATA, ct, new_item);
      break;
   }
}
```

Errors

A return value of 0 from `sm_prop_get_str`, `sm_prop_get_dbl`, or one of its variants usually indicates that the call failed. However, some string and double properties accept `NULL` or 0 values. To determine with absolute certainty whether a call failed and to get its error code, call `sm_prop_error`.

A negative value returned by `sm_prop_get_int` and its variants usually specifies an error. However, some integer properties accept negative values; in these cases, you can differentiate between a negative property value and an error condition only by calling `sm_prop_error`.

See Also `sm_prop_error`, `sm_prop_id`, `sm_prop_set`

sm_prop_id

Returns an integer handle for an application component

```
#include <smuprapi.h>
int sm_prop_id(char *obj_name);
```

`obj_name`

A string that identifies an object in the current application. The string must conform to Panther object name conventions. For information about valid formats, refer to "Properties" on page 19-40 in *Application Development Guide*.

For example, this call to `sm_prop_id` gets a handle to the `cust_id` widget in the `custlist` screen:

```
err = sm_prop_id
      ("@screen('custlist')!@widget('cust_id')");
```

A non-subscripted widget identifier returns a handle to the entire widget. If the widget is an array, you can use this handle to get or set properties for all occurrences and elements. You can also create handle to an array that lets you get or set properties for individual occurrences or elements. To do this, include an empty subscript in the widget's string identifier, using one of these two formats:

- `widget-spec[]` enables access to properties of occurrences in `widget-spec`.
- `widget-spec[[]]` enables access to properties of elements in `widget-spec`.

For example, the handle returned by this call to `sm_prop_id` can be used as an argument to variants of `sm_prop_get_x_prop-type` to get or set properties of elements in `cust_id`:

```
sm_prop_id("@widget('cust_id')[[]]");
```

Refer to Description for more information about obtaining access to the properties of an array's occurrences or elements.

Returns

≥ 1 Integer handle to the specified object.

- `PR_E_ERROR`: Failed for another reason.

- `PR_E_OBJID`: Object ID does not exist.
- `PR_E_OBJECT`: Object does not exist.
- `PR_E_ITEM`: Invalid element or occurrence.

Description

`sm_prop_id` gets an integer handle to an application component—widgets and array elements or occurrences. Use this handle to get and change the component's properties with calls to functions like `sm_prop_get_str` or `sm_prop_set_int`.

Access to Occurrence and Element Properties

You can get three kinds of handles to an array, depending on whether the array's string identifier contains a subscript and the subscript's format:

- A non-subscripted identifier returns a handle that lets you get or set properties for the array as a whole. The following sequence of calls changes the `reverse` property for all elements and occurrences in array `cust_id`:

  ```
  arr_h = sm_prop_id("cust_id");
  sm_prop_set_int(arr_h, PR_REVERSE, PV_YES);
  ```

- An empty subscript of single paired brackets—`[]`–returns a handle to an array that you can supply to _x variants of `sm_prop_get` and `sm_prop_set` to get and set properties of individual occurrences. The following sequence of calls changes the `reverse` property for the first occurrence in array `cust_id`:

  ```
  occ_h = sm_prop_id("cust_id[]");
  sm_prop_set_x_int(occ_h, 1, PR_REVERSE, PV_YES);
  ```

- An empty subscript of double paired brackets—`[[]]`–returns a handle to an array that you can supply to _x variants of `sm_prop_get` and `sm_prop_set`, to get and set properties of individual elements. The following sequence of calls changes the `reverse` property for the first element in array `cust_id`:

  ```
  elem_h = sm_prop_id("cust_id[[]]");
  sm_prop_set_x_int(elem_h, 1, PR_REVERSE, PV_YES);
  ```

See Also

`sm_prop_get`, `sm_prop_set`

sm_prop_name_to_id

Gets the integer ID of a Panther property

```
int sm_prop_name_to_id(char* jpl_prop_str);
```

`jpl_prop_str`
    The JPL mnemonic for the desired property.

Returns

≥1	The integer ID of the specified property.
<0	No match found.

Description

`sm_prop_name_to_id` gets the integer ID for the supplied JPL property name. Access to this ID lets you call C library routines such as `sm_prop_get` and `sm_prop_set` from JPL. JPL only has direct access to its own property mnemonics, while calls to these routines require the property identifiers that are defined in Panther header files.

For example, JPL gets the number of occurrences in an array through `num_occurrences`, while `sm_prop_get` takes `PR_NUM_OCCURRENCES` to specify the same property. With `sm_prop_name_to_id`, you can translate the JPL mnemonic to the integer value of `PR_NUM_OCCURRENCES` and call `sm_prop_get` from JPL:

```
/* get the number of selections in group 'genre' */
vars num_selects

num_selects = sm_prop_get_int( \
                 sm_prop_id("genre"), \
                 sm_prop_name_to_id("num_occurrences"))
```

sm_prop_set*

Sets a property

```
#include <smuprapi.h>
int sm_prop_set_int(int obj_id, int prop, int val);
int sm_prop_set_str(int obj_id, int prop, char *val);
int sm_prop_set_dbl(int obj_id, int prop, double val);
int sm_prop_set_x_int(int obj_id, int array_item, int prop,
    int val);
int sm_prop_set_x_str(int obj_id, int array_item, int prop,
    char *val);
int sm_prop_set_x_dbl(int obj_id, int array_item, int prop,
    double val);
int sm_prop_set_m_int(int obj_id, int prop, int prop_item,
    int val);
int sm_prop_set_m_str(int obj_id, int prop, int prop_item,
    char *val);
int sm_prop_set_m_dbl(int obj_id, int prop, int prop_item,
    double val);
```

`obj_id`
: An integer handle that identifies the Panther object whose property you want to set, obtained through `sm_prop_id`. For application properties, supply `PR_APPLICATION`; for the current screen, `PR_CURSCREEN`.

`array_item`
: The widget occurrence or element whose value you want to set.

`prop`
: The property to set. Refer to Chapter 1, “Runtime Properties,” in *Quick Reference* for a full list of property constants.

`prop_item`
: Specifies the item in a multi-item property whose value you want to set. For example, if `prop` is set to `PR_CONTROL_STRING`, supply a logical key name to get that key’s current control string assignment.

`val`
: The value to set for the specified property or property item. The value’s type—string, integer, or double—must be appropriate to the property itself.

For a list of properties and their valid values, refer to Chapter 1, “Runtime Properties,” in *Quick Reference*.

Returns

- 0: Success.
- `PR_E_MALLOC`: Insufficient memory.
- `PR_E_OBJID`: Object ID does not exist.
- `PR_E_OBJECT`: Object does not exist.
- `PR_E_ITEM`: Invalid occurrence or element.
- `PR_E_PROP`: Invalid property.
- `PR_E_PROP_ITEM`: Invalid property item.
- `PR_E_PROP_VAL`: Invalid property value.
- `PR_E_CONVERT`: Unable to perform conversion.
- `PR_E_OBJ_TYPE`: Invalid object type.
- `PR_E_RANGE`: Property value is out of range.
- `PR_E_NO_SET`: Property cannot be set.
- `PR_E_BEYOND_SCREEN`: Widget extends beyond screen.
- `PR_E_WW_SCROLLING`: Word wrap must be scrolling.
- `PR_E_NO_SYNC`: Arrays cannot be synchronized.
- `PR_E_TOO_BIG`: Widget too large for screen.
- `PR_E_ERROR`: Failed for another reason.
- `PR_E_BAD_MASK`: Invalid edit mask or regular expression
- `PR_E_NO_KEYSTRUCT`: Property requires previous execution of `SELECT`, `NEW`, `COPY`, or `COPY_FOR_UPDATE` command.

Description

`sm_prop_set` has three basic variants: `sm_prop_set_str`, `sm_prop_set_int`, and `sm_prop_set_dbl`, which set string, integer, and double properties, respectively. For example, `sm_prop_set_str` sets string properties such as `title`, while `sm_prop_set_int` sets integer properties such as `max_occurrences`.

Each of these variants have `_x` and `_m` variants. These let you set properties of occurrences or elements, and offsets into properties that take multiple values, respectively. These variant types are discussed in the following sections.

Elements and Occurrences

You can set properties for individual elements and occurrences in an array by calling `sm_prop_set_x_`*`prop-type`*. All variants of this function require an `obj_id` handle to the array and an `array_item` argument. Depending on how the `obj_id handle` was obtained, the function determines whether `array_item` specifies an offset into the array’s elements or its occurrences:

- To set the properties of an array's elements, obtain a handle by supplying sm_prop_id with a widget identifier that has the format *widget-spec*[[]].
- To set the properties of an array's occurrences, obtain a handle by supplying sm_prop_id with a widget identifier that has the format *widget-spec*[].

For example, this call to sm_prop_id gets a handle to the properties of cust_id's elements:

```
int elem_h;
elem_h = sm_prop_id("cust_id[[]]");
```

Alternatively, this call gets a handle to the properties of cust_id's occurrences:

```
int occ_h;
occ_h = sm_prop_id("cust_id[]");
```

Given these two handles, you can use sm_prop_get_x_int to set the foreground color of either cust_id's first element or first occurrence as follows:

```
/*set the first element's foreground color */
sm_prop_set_x int(elem_h, 1, PR_FG_COLOR_NUM, MAGENTA);

/*set the first occurrence's foreground color */
sm_prop_set_x_int(occ_h, 1,PR_FG_COLOR_NUM, MAGENTA);
```

Note: To set properties on the entire array, use a handle obtained by supplying sm_prop_id with a widget string identifier that contains no subscript.

Multi-item Properties

sm_prop_set_m_*prop-type* sets one of the values in a multi-item property such as PR_DROP_DOWN_DATA for an option menu, or PR_CONTROL_STRING for a screen. For example, this code calls sm_prop_set_m_str to set the data for an item in option menu flavors:

```
/* replace current item with contents of "substitute" */
char cur_item[256], new_item[256];
char *option_txt[256];
int ct, f_id, err;

f_id = sm_prop_id("flavors");

/*get substitute data*/
sm_n_getfield("substitute", new_item);

/*get selection data*/
sm_n_getfield("flavors", cur_item);
```

```
/* get offset of current selection */
for (ct = 1; ; ct++)
{
   option_txt = sm_prop_get_m_str(f_id,
                     PR_DROP_DOWN_DATA, ct)
   if (!option_txt)
   {
      err = PR_E_ERROR;
      break;
   }
   if (strcmp(option_txt, cur_item) == 0)
   {
      err = sm_prop_set_m_str(f_id,
               PR_DROP_DOWN_DATA, ct, new_item);
      break;
   }
}
```

See Also `sm_prop_error`, `sm_prop_id`, `sm_prop_set`

sm_pset

Modifies the value of a global string

```
#include <smglobs.h
char *sm_pset(int which, char *newval);
```

`which`

Specifies the global string to modify with one of these constants:

`P_YES`
: Set the affirmative input that is valid for a field whose `keystroke_filter` is set to `PV_YES_NO`. Supply a two-character string that contains the lowercase yes value and the uppercase yes value.

`P_NO`
: Set the negative input that is valid for a field whose `keystroke_filter` is set to `PV_YES_NO`. Supply a two-character string that contains the lowercase no value and the uppercase no value.

`P_DECIMAL`
: Set the user's decimal point marker and the operating system's decimal point marker in a two-character string.

`P_TERM`
: Set the terminal type. You must call `sm_pset` with this argument this before initialization.

`P_USER`
: Set a pointer to a developer-specified region of memory for the current screen. Each screen maintains its own pointer. This pointer is not set by Panther; it is set and maintained by the application.

`SP_NAME`
: Set the name of the active screen.

`SP_STATATTR`
: Set attributes of current status line—a pointer to an array of unsigned short integers.

`SP_STATLINE`
: Set the current text of the status line as a space padded, 255 character string (not including the terminating null).

`V_`
: One of the `V_` constants defined in `smvideo.h`, returns video-related information.

`newval`
: The new value to assign to this global string.

Note: If you supply a `V_` constant for `which`, declare this parameter as a static variable.

Returns

- A pointer to a buffer with the old contents of the array specified by `which`. The buffer's maximum size of 255 bytes, including the null terminator.
- 0: `which` is invalid.

Description

`sm_pset` lets you modify the contents of the `which`-specified global string. To get the value of a global string, use `sm_pinquire`.

Example

```
/* Set things for "German":   Ja == yes, */
/* Nein == no, and ',' is decimal point. */

void
   set_german()
   {
      sm_pset(P_YES,"jJ");
      sm_pset(P_NO,"nN");
      sm_pset(P_DECIMAL,",.");
      sm_ferr_reset(0, "Jetzt spreche ich Deutsch!");
   }
```

See Also

`sm_iset`, `sm_pinquire`

sm_*putfield

Puts a string into a field

```
int sm_putfield(int field_number, char *data);
int sm_e_putfield(char *field_name, int element, char *data);
int sm_i_putfield(char *field_name, int occurrence, char *data);
int sm_n_putfield(char *field_name, char *data);
int sm_o_putfield(int field_number, int occurrence, char *data);
```

`field_name, field_number`
The field to receive the contents of `data`.

`element`
The element in array `field_name` to receive the string.

`occurrence`
The occurrence in the field to receive the string.

`data`
A pointer to the string to put in the specified field or occurrence.

Returns

0 Success.
- 1 Failure.

Description

`sm_putfield` moves the string in `data` into the specified field, if it differs from the existing value. If the string is too long, Panther truncates it without warning. If the string is shorter than the destination field, Panther blank fills it according to the field's justification. If data points to an empty string, the field is cleared. This refreshes date and time fields that take system values.

`sm_putfield` sets the field's `mdt` property to `PV_YES` to indicate that it is modified, and clears its valided property to `PV_NO` to indicate that the field requires validation on exit. If you use variants `sm_n_putfield` or `sm_i_putfield` and `field_name` is absent from the screen, the value of data is put in the corresponding LDB entry.

Example

```
#include <smdefs.h>

    sm_putfield(1, "This string has 29 characters");
```

See Also

`sm_deselect`, `sm_dtofield`, `sm_getfield`, `sm_itofield`, `sm_ltofield`

sm_raise_exception

Sends an error back to the client

```
int sm_raise_exception(int error, char *message);
```

`error`
: Error code to be returned to the client with COM components.

`message`
: Error message to be returned to the client with Enterprise JavaBeans.

Environment	COM, EJB
Scope	Server
Description	`sm_raise_exception` sends an error code and message back to the client. The client's error handler then can decide what to do based on the value sent. For COM applications, Microsoft defines some conventional exception codes for use in COM programming; see `winerror.h`.
See Also	`sm_receive_args`, `sm_return_args`

sm_receive

Executes a JPL receive command

```
int sm_receive(char *receive_args);
```

`receive_args`

A string constant that contains `receive` command arguments, using one of the following formats:

`[ bundle` *`bundleName`*`] [ item` *`itemNo`*`] [ keep] data` *`fieldExpr`*

`{ ARGUMENTS | MESSAGE } ( [`*`receiveArg`*`] )`

Refer to the `receive` command for a description of these arguments.

Returns

0	Success.
-1	Unable to execute the function, or execution aborted prematurely. Refer to the `receive` command for potential error conditions.
-2	Memory allocation failure.

Description

`sm_receive` reads data from a bundle that was written by an earlier call to `sm_send` or the JPL send command—typically, from another screen. `sm_receive` reads the data into its *`field-expr`* arguments in the same order that it was sent. Unless you supply the `keep` argument, the bundle data is discarded after `sm_receive` completes execution.

For more information, refer to the JPL `receive` command.

See Also

`sm_send`

sm_receive_args

Receives a list of in and in/out parameters for a method

```
int sm_receive_args(char *text);
```

`text`
: List of in/out and out parameters, separated by commas, of field names and global JPL variables.

Environment COM, EJB

Scope Server

Returns
- 0 Not an available method.
- 1 Success.
- Otherwise, a value from `smerror.h`

Description `sm_receive_args` receives a list of arguments, and writes them to the in and in/out parameters of a method. The arguments are written to the parameters in the order received.

Example See the example under the JPL verb `receive_args`.

See Also `sm_return_args`

sm_rescreen

Refreshes the data displayed on the screen

```
void sm_rescreen(void);
```

Description

`sm_rescreen` repaints the entire display from Panther's internal screen and attribute buffers. This function erases anything written to the screen by means other than Panther library functions. This function is normally bound to the REFR key and executes automatically within `sm_getkey`.

You might need to call this function explicitly under the following conditions:

- Screen I/O occurs with the flag `sm_do_not_display` turned on.
- Escape from an Panther application to another program through `sm_leave`.

To force writes to the display, use `sm_flush`.

See Also

`sm_flush`, `sm_return`

sm_*resetcrt

Resets the terminal to the operating system's default state

```
void sm_resetcrt(void);
void sm_jresetcrt(void);
void sm_jxresetcrt(void);
```

Environment

C only

Description

sm_resetcrt resets terminal characteristics to the operating system's normal state. Use this function only with your own executive. Call sm_resetcrt when leaving the screen manager environment before program exit.

All the memory associated with the display and open screens is freed. However, the buffers that hold the message file, key translation file, and so on, are not released. A subsequent call to sm_initcrt finds them in place. In character-mode, sm_resetcrt then clears the screen and turns on the cursor, transmits the RESET sequence defined in the video file, and resets the operating system channel.

Panther automatically calls sm_resetcrt through sm_jresetcrt or—in the case of the screen editor—sm_jxresetcrt as part of its exit processing. These two functions should not be called by application programs except in case of abnormal termination.

Example

```
/* If an effort to read the first form results in
    * failure, clean up the screen and leave. */

   if (sm_r_form("first") < 0)
   {
      sm_resetcrt();
      exit(1);
   }
```

See Also

sm_cancel, sm_leave

sm_resize

Notifies Panther of a change in the display size

```
int sm_resize(int rows, int columns);
```

`rows, columns`
: Specifies the new display size, where the maximum value of `rows` and `columns` is 255. If the specified rectangle is larger than the physical display, results can be unpredictable.

Returns

- 0: Success.
- −1: Failure. A parameter is less than 0 or greater than 255.
- Program exit on memory allocation failure.

Description

`sm_resize` lets you change the default display set by the video file's `LINES` and `COLMS` entries. Character-mode applications can run in different-sized windows by setting their individual display sizes at runtime. Also use `sm_resize` to switch between normal and compressed modes—for example, 80 and 132 columns on VT100-compatible terminals.

Example

```
#include <smdefs.h>
   #include <smkeys.h>
   #include <smglobs.h>
   #define WIDTH_TOGGLE PF9

   /* Somewhat irregular code to switch a VT-100
    * between 80- and 132-column mode by pressing PF9. */

   switch (sm_input(IN_DATA))
   {
   ...
   case WIDTH_TOGGLE:
      if (sm_inquire(I_MXCOLMS) == 80)
      {
         printf("\033[?3h");
         sm_resize(sm_inquire(I_MXLINES), 132);
      }
      else
      {
         printf("\033[?3l");
         sm_resize(sm_inquire(I_MXLINES), 80);
      }
      break;
   ...
   }
```

sm_restore_data

Restores previously saved data to the screen

```
int sm_restore_data(char *buffer);
```

`buffer`
: The address of an area initialized by `sm_save_data` that contains the data to restore. Data items are stored in `buffer` as null-terminated character strings. The contents of a scrollable array is preceded by 2 bytes giving the total number of items saved (high order byte first); each item is preceded by two bytes of display attribute, and followed by a null. There is an additional null following all the scrolling data.

Environment C only

Returns

0	Success.
-1	Error, usually memory allocation failure.

Description `sm_restore_data` restores all data items, on- and off-screen, to the current screen from `buffer`, previously initialized by `sm_save_data`. Passing a buffer not returned by `sm_save_data`, or attempting to restore to a screen other than the one saved, can yield unpredictable results.

See Also `sm_save_data`, `sm_sv_free`

sm_return

Prepares for return to a Panther application

```
void sm_return(void);
```

Environment C only

Description Call `sm_return` on returning to a Panther application after a temporary exit. This function sets up the operating system channel, and in character-mode initializes the display with the video file's `SETUP` string.

Note that `sm_return` does not restore the screen to its state before the call to `sm_leave`. To restore the screen to its previous state, call `sm_rescreen`.

Example

```
#include <smdefs.h>

    /* Escape to the UNIX shell for a directory listing */

    sm_leave();
    sm_system("ls -l");
    sm_return();
    sm_c_off();
    sm_d_msg_line("Hit any key to continue", BLINK | WHITE);
    sm_getkey();
    sm_d_msg_line("", WHITE);
    sm_rescreen();
```

See Also `sm_leave`, `sm_resetcrt`

sm_return_args

Returns a method's in/out and out parameters

```
int sm_return_args(char *text);
```

`text`
: List of in/out and out parameters, separated by commas.

Environment	COM, EJB
Scope	Server
Returns	0 Not an available method. 1 Success. ● Otherwise, a value from `smerror.h`
Description	`sm_return_args` is passed a list of arguments to be written to the in/out and out parameters of a method.
Example	See the example under the JPL verb `receive_args`.
See Also	`sm_receive_args`, `sm_raise_exception`

sm_rmformlist

Purges the memory-resident form list

```
void sm_rmformlist(void);
```

Environment C only

Description `sm_rmformlist` erases the memory-resident form list established by `sm_formlist`, and releases the memory used to hold it. It does not release any of the memory-resident JPL modules or screens. Calling this function prevents `sm_r_window`, `sm_jplcall`, and related functions from finding memory-resident objects.

Example

```
/* Hide all the memory-resident screens, perhaps
 * because the disk versions have been modified. */

   sm_rmformlist();
```

See Also `sm_formlist`

sm_rs_data

Restores saved data to some of the screen

```
int sm_rs_data(int first_field, int last_field, char *buffer);
```

`first_field, last_field`
: Specifies the range of data items to restore, where all data between `first_field` and `last_field` are restored to the screen.

`buffer`
: The address of a buffer, initialized by `sm_sv_data`, that stores the data to restore. Data items are stored in `buffer` as null-terminated character strings. The contents of a scrollable array is preceded by 2 bytes giving the total number of items saved (high order byte first); each item is preceded by two bytes of display attribute, and followed by a null. There is an additional null following all the scrolling data.

Environment C only

Returns

0	Success.
-1	Error, usually memory allocation failure.

Description `sm_rs_data` restores all data items between `first_field` and `last_field`, both off- and onscreen, from a buffer initialized by `sm_sv_data`.

The range of fields passed to `sm_rs_data` must match those passed to `sm_sv_data` and `buffer` must be a value returned by that function; otherwise, serious errors may occur. For more information on saving data for later retrieval by `sm_rs_data`, refer to `sm_sv_data`.

See Also `sm_sv_data`

sm_rw_error_message

Returns the last error message generated by report processing

```
char *sm_rw_error_message(void);
```

Returns

- The last report error message.
- `NULL`: No errors occurred.

Description

`sm_rw_error_message` returns the last error message returned by a report that is undergoing execution. The character string that this function returns can be used in one of the Panther message commands or functions, such as the msg command or sm_message_box. If report processing is error-free, this function returns `NULL`.

You can use this function to test report processing within a Panther application—for example, through `sm_rw_runreport` or `sm_rw_play_metafile`.

sm_rw_play_metafile

Displays or prints a report that is in metafile format

```
#include <rwdefs.h>
int sm_rw_play_metafile(char *invocation_str);
```

`invocation_str`

A string that contains the name of the metafile to run and one or more invocation options in this format:

"metafile-name [option]..."

If *`metafile-name`* contains spaces, it must be quoted. For a description of invocation options, refer to “Setting Invocation Options” on page 9-9 in *Reports*.

Returns

0	Success.
-1	A syntax error occurred or the specified file is not in metafile format.

Description

`sm_rw_play_metafile` takes an existing metafile and processes it according to the specified invocation options. Metafiles are the output of a report invocation that specifies `rwmetafile` as its output driver (refer to “Driver Options” on page 9-11 in *Reports*). For example, this statement saves the first two pages of metafile `report1.rwm` to a PostScript file:

```
retcode = sm_rw_play_metafile \
    ("report1.rwm driver=postscript \
     output=report1.ps overwrite topage=2")
```

This function can be used in an report service’s unload handler to process server-generated metafiles. For more information about using this function in unload handlers, refer to “Handling Client Output” on page 9-26 in *Reports*.

sm_rw_runreport

Invokes the report generator from a user-written function

```
#include <rwdefs.h>
int sm_rw_runreport(char *invocation_str);
```

`invocation_str`

A string that contains the name of the report to be invoked, arguments passed to the report, and output and page layout options. The format of the string is identical to the invocation string for the JPL command `runreport`:

`"`*`filename`*`[!`*`reportname`*`] [ ('`*`arg`*`'[, ... ] )] [ `*`option`*` ]..."`

For a description of invocation arguments and options, refer to “Setting Invocation Options” on page 9-9 in *Reports*.

Returns

0	Success.
-1	Failure.

Description `sm_rw_runreport` invokes the report generator from a user-written function that is linked into a Panther application. This function is functionally identical to the JPL command `runreport`.

Example

```
#include <rwdefs.h>

if (sm_rw_runreport("rptfile!myreport
        ('myarg1', 'myarg2') output=myoutput") == -1)
    {
        sm_n_putfield("myrwstatus", "failure");
    }
```

sm_s_val

Validates the current screen

```
int sm_s_val(void);
```

Returns

0	Success.
-1	A field failed validation.

Description

`sm_s_val` validates all fields and their occurrences, on- and offscreen, that are not protected from validation. Calling this function is equivalent to calling `sm_fval` for each field and its occurrences. If an occurrence fails validation, `sm_s_val` repositions the cursor to it and displays an error message. If the occurrence is offscreen, `sm_s_val` scrolls the array until it is visible. The function then stops validation and returns. Fields that follow the invalid occurrence remain unvalidated.

`sm_s_val` validates array occurrences sequentially, whether onscreen or offscreen. Thus, offscreen occurrences that precede the first onscreen occurrence are validated first.

`sm_s_val` also validates groups, grids, and tab cards. A group is validated when its first field would be validated were it not a group member. Fields that are members of a group are not validated individually. A grid is validated after its last field is validated. All fields, grids and tab cards that are on a tab card are validated together and before the card's validation function is called. When a card is validated, offscreen occurrences of fields not on that card may be validated if they are synchronized with fields on the card.

`sm_s_val` validates synchronized arrays by processing parallel occurrences sequentially. The function begins by validating the first occurrence (on- or offscreen) of the array with the lowest base field number, then the first occurrence of the array with the next base field number, and so on. `sm_s_val` completes validation when it processes the last occurrence of the array with the highest base field number.

For more information about field validation processing, refer to `sm_fval`.

Example

```
proc screen_exit()

    if (sm_s_val()) // found invalid field data, returned -1
    {
        msg err_reset \
           "Erroneous data; please correct and save again"
        return
    }
    ...
    return
```

See Also `sm_fval`, `sm_n_gval`, `sm_validate`

sm_save_data

Saves screen contents

```
char *sm_save_data(void);
```

Environment C only

Returns

- The address of a memory area that contains the screen's data.
- 0: Insufficient memory.

Description `sm_save_data` saves the current screen's data for external access or subsequent retrieval and returns the address of the save area. `sm_save_data` ignores selections from the following widgets: radio buttons, toggle buttons, check boxes, and list boxes.

To restore the saved data, use `sm_restore_data`. Use `sm_sv_free` to discard a save area.

You can get the size of the data with this statement:

```
length = ((unsigned int *)buffer)[-1];
```

See Also `sm_restore_data`, `sm_sv_data`, `sm_sv_free`

sm_sb_delete

Deletes a status bar section

```
int sm_sb_delete(int sectno);
```

`sectno`
The number of the section to be deleted.

Returns	
0	Success
-1	Failure

Description This function deletes a section of the status bar as specified by its index in the array of status bar sections. The initial section, the message line, always has section number `0` and cannot be deleted. Hence, the argument to this function must always be `>=1`.

See Also `sm_sb_format`, `sm_sb_gettext`, `sm_sb_insert`, `sm_sb_settext`

sm_sb_format

Sets a format string for a status bar section

```
int sm_sb_format(int sectno, char* format)
```

`sectno`
: The number of the section for which to specify a format string.

`format`
: A format string for a section.

Returns

0	Success
-1	Failure

Description This function sets the format string for a status bar section. This is relevant only for sections of type `SBS_SYSTEM_TIME` and `SBS_ELAPSED_TIME`, as described above.

The second argument, `format`, represents a format string. Valid date/time format strings are described in the documentation for the function `sm_sdtime`. Note that an `SBS_ELAPSED_TIME` section displays a clock that starts at midnight when the section is created. So a format string for a section of that type should be chosen so that it is meaningful in that context.

sm_sb_gettext

Get contents of a status bar section

```
char* sm_sb_gettext(int sectno);
```

`sectno`
The number of the section being queried.

Returns

- The section's contents
- -1 Error

Description

This function gets the contents of a status bar section. The text returned is as shown on the status bar, and may differ from the text set with `sm_sb_settext` if that text contained formatting tokens.

See Also

`sm_sb_settext`

sm_sb_insert

Inserts a status bar section

```
int sm_sb_insert(int sectno, int type, int length);
```

`sectno`
: The index, in the array of sections, of the section to be added.

`type`
: The type of the new section, one of the following constants:

```
SBS_TEXT
SBS_SEPARATOR
SBS_SYSTEM_TIME
SBS_ELAPSED_TIME
SBS_OVERLAY
SBS_CAPS
SBS_NUM
SBS_SCROLL
```

`length`
: The length of the section to be added.

Returns

- The section number given to the new section

-1 Failure

Description

This function inserts a new section on the status bar. When the status bar is initially created it contains a single section of type `SBS_MSGLINE`. This initial section is the one written to by the various Panther functions that send messages to the status line. The initial `SBS_MSGLINE` section occupies the `0` position in the array of status bar sections. Newly added sections must be placed to the right of the initial section, hence the value of the first argument to `sm_sb_insert` cannot be `0`. If you supply a negative value to the first parameter, the newly added section will be the rightmost, no matter how many sections already exist.

The newly added section must be one of several pre-defined types, as specified by the second argument to `sm_sb_insert`. You cannot add a second section of type `SBS_MSGLINE`. Hence, the valid values for the second argument are as follows:

`SBS_TEXT`
: This type of section is used to display text. Text is written to such a section using the funtion `sm_sb_settext`.

`SBS_SEPARATOR`
: This type of section is used to mark a boundary between two other sections. In character mode, it is equivalent to `SBS_TEXT`, and you can write whatever character you wish to it, to mark the section boundary. In a GUI `SBS_SEPARATOR` sections aren't displayed in a recessed style.

`SBS_SYSTEM_TIME`
: This type of section displays the system time. The format for the time displayed is set by the function `sm_sb_format`. The default format shows the time in a 12-hour clock, with an AM/PM indicator.

`SBS_ELAPSED_TIME`
: This type of section displays the time elapsed since the section was created. The format for the time displayed is set by the function `sm_sb_format`. The default format shows the time in the form `'00:00:00'`.

`SBS_OVERLAY`
: This type of section displays the state of Panther's insert/overstrike mode. The length parameter is ignored if this is the type specified. In character mode the length defaults to 3, and will either display `'OVR'` or be blank. In a GUI the `'OVR'` indicator may be grayed out rather than blanked.

`SBS_CAPS`
: This type of section displays the CAPS LOCK state of the keyboard. The length parameter is ignored if this is the type specified. This type is not supported in character mode.

`SBS_NUM`
: This type of section displays the NUM LOCK state of the keyboard. The length parameter is ignored if this is the type specified. This type is not supported in character mode.

`SBS_SCROLL`
: This type of section displays the SCROLL LOCK state of the keyboard. The length parameter is ignored if this is the type specified. This type is not supported in character mode.

Other than for `SBS_TEXT` and `SBS_SEPARATOR`, you can have only one section of each type on the status bar. Calls to `sm_sb_insert` that specify a type that already exists on the status bar will have no effect. You can insert any number of `SBS_TEXT` or `SBS_SEPARATOR` sections.

The `length` parameter is the length, in characters, of the section to be added. The length specified should be greater than or equal to the length of any text that might be placed in that section. This parameter is ignored for some section types, as noted above.

Note that the length of the status bar as a whole remains constant, and that the message line section initially occupies all of it. In Motif, the message line section is always 255 characters long, so any sections placed after it will appear displaced by 255 character positions. As a result, in Motif sections added to the status bar will probably not be visible unless the window containing the status bar is very wide. To compensate for this you can, in Motif, add a trailing `SBS_SEPARATOR` section that's wide enough to force the section to the right of the message line section to become visible.

Since in the GUIs the screen space allocated to status line sections is font-dependent, you may need to experiment with different lengths to get the status line sections to appear the way you want them.

See Also `sm_sb_delete`, `sm_sb_format`, `sm_sb_gettext`, `sm_sb_settext`

sm_sb_settext

Set contents of a status bar section

```
int sm_sb_settext(int sectno, char* text);
```

`sectno`
The number of the section to update

`text`
The text to place in that section

Returns

0	Success
-1	Failure

Description

This function assigns contents to a section of the status bar. The text specified as the second argument to this function may contain formatting tokens such as `%A` and `%K`. See the description of the JPL command `msg` for descriptions of the valid formatting and key value display tokens.

See Also

`sm_sb_gettext`

sm_sdtime

Gets the formatted system date and time

```
char *sm_sdtime(char *format);
```

`format`
Specifies the format to use with an expression that starts with `y` or `n`, followed by any combination of date/time tokens and literal text. `y` indicates a 12-hour clock; `n` or any other character indicates a 24-hour clock. This character is required even if the format does not include time tokens. The table in Description shows the date/time tokens that you use to build a format expression.

Returns

- A pointer to a string that contains the current date/time in the specified format.
- Empty: `format` is invalid.

Description

`sm_sdtime` gets the current date and/or time from the operating system and returns it in the `format`-specified format.

The following table lists the tokens you use to build a format expression. All tokens are prefixed by the percent sign (%) and are case-sensitive.

Table 5-19 Date/time format options

Unit	Description	Token
year	4 digit (e.g., 1990)	%4y
	2 digit (e.g., 90)	%2y
month	1 or 2 digit (1 - 12)	%m
	2 digit (01 - 12)	%0m
	full name (e.g., January)	%*m
	3 character name (e.g., Jan)	%3m
day	1 or 2 digit (1 - 31)	%d
	2 digit (01 - 31)	%0d

Table 5-19 Date/time format options *(Continued)*

Unit	Description	Token
day of the week	full name (e.g., Sunday)	%*d
	3 character name (e.g., Sun)	%3d
	numeric day of the week (1-7)	%.d
day of the year	digit (1 - 366)	%+d
hour	1 or 2 digit (1 - 12 or 0 - 23)	%h
	2 digit (01 -12 or 00 -23)	%0h
minute	1 or 2 digit (0 - 59)	%M
	2 digit (00 - 59)	%0M
second	1 or 2 digit (0 - 59)	%s
	2 digit (00 - 59)	%0s
AM or PM	for use with a 12-hour clock	%p
literal percent	use `%` as a literal character	%%

default formats from the message file (refer to "Date/Time Defaults" on page 45-13 in *Application Development Guide*)	`SM_0DEF_DTIME`
	`SM_1DEF_DTIME`
	`...`
	`SM_9DEF_DTIME`
		`%0f`
		`%1f`
		`...`
		`%09f`

At runtime, Panther strips off the first character of `format`. If the character is `y`, it uses a 12-hour clock; otherwise, it uses the 24-hour clock. Next, it examines the rest of `format`, replacing any tokens with the appropriate values. All non-token characters are treated as literal values.

The message file contains the text for day and month names, AM and PM, and the tokens for the default formats. You can modify these. Refer to "Date/Time Defaults" on page 45-13 in *Application Development Guide* for details.

`sm_sdtime` uses a 256-byte static buffer that it shares with other date and time formatting functions. Because Panther does not check for overflow, process the returned string or copy it to a local variable immediately.

Example

```
#include <smdefs.h>
    /* Put current date MONTH-DAY-YEAR in the field "time". */
    char *format;
    format = "n%m-%0d-%2y";
    sm_n_putfield("time", sm_sdtime(format));
```

See Also `sm_udtime`

sm_select

Selects an occurrence in a selection group

```
int sm_select(char *selection_group, int group_occurrence);
```

`selection_group`
The name of a selection group.

`group_occurrence`
The number of the occurrence in `selection_group` to select.

Returns

1	Occurrence is already selected.
0	Occurrence not previously selected.
-1	Invalid reference to group or occurrence.

Description

`sm_select` lets you select an occurrence within a selection group. If the group's `num_of_selections` property is set to `PV_1` (allows only one selection), Panther first deselects the current selection before it selects `group_occurrence`. For more information about selection widgets, refer to Chapter 20, "Selection Widgets," in *Using the Editors*.

To deselect an occurrence, call `sm_deselect`.

See Also

`sm_deselect`

sm_send

Executes a JPL send command

```
int sm_send(char *send_args);
```

`send_args`

A string constant that contains `send` command arguments:

`[ bundle` *`bundle-name`* `] [ append ] data` *`data-expr`*`[,...]`

For a description of these arguments, refer to the `send` command.

Returns

0	Success.
-1	Unable to execute the function, or execution aborted prematurely. Refer to the `send` command for potential error conditions.
-2	Memory allocation failure.

Description

`sm_send` executes a JPL send command exactly as if called from JPL. `sm_send` writes screen data to a buffer that is accessible to other screens through calls to `sm_receive` or the JPL `receive` command. `sm_send` can send one or more values from fields and array occurrences on a screen. It can also send character string constants as well as parts of arrays or the current occurrence of an array.

Panther writes the data that you specify in `sm_send` to a temporary buffer, or *bundle*, which you can optionally name. Panther by default maintains up to ten bundles; you can set the number of available bundles using the `max_bundles` property. If you omit a bundle name, Panther writes the data to an unnamed bundle; this data is accessed by the next call to `sm_receive` or `receive` that also omits a bundle name argument or specifies it as an empty string.

For more information, refer to the `send` command.

See Also

`sm_receive`

sm_set_help

Puts an application into help mode

```
void sm_set_help(void);
```

Description

`sm_set_help` puts Panther into help mode. When Panther is in help mode, mouse-clicking on any object in the Panther application invokes the help that is associated with that object. On GUI platforms, help mode changes the mouse pointer shape to the symbol associated with help—for example, on Windows and Motif, a question mark with an arrow. In character mode, the status line displays `Help Mode`.

While Panther is in help mode, the user can click on any application object that can have help associated with it—a screen, toolbar or menu item, or widget. Help on an object is accessible whether or not the object is inactive or focus-protected. Clicking on the screen is equivalent to using the logical key FHLP; clicking on a widget is equivalent to HELP.

Panther exits help mode and restores the mouse pointer shape after a mouse click occurs.

sm_setbkstat

Sets background text for status line

```
void sm_setbkstat(char *message, int display_attr);
```

`message`
: Specifies the message to display as background text.

`display_attr`
: The display attributes to use for `message`, one of the constants defined in `smattrib.h`.

  Foreground colors can be used alone or OR'd with one or more highlights, a background color, and a background highlight. If you do not specify a highlight or a background color, the attribute defaults to white against a black background. Omitting a foreground color causes the attribute to default to black.

Description

`sm_setbkstat` saves the contents of `message` for display on the status line when there is no other message with a higher priority to display. The highest priority messages are those passed to `sm_d_msg_line`, `sm_ferr_reset`, and `sm_fquiet_err`; the next highest are those attached to a widget or screen through its `status_line_text` property. Background status text has lowest priority.

`sm_setstatus` sets the background status to an alternating ready/wait flag; turn this feature off before calling `sm_setbkstat`.

`sm_d_msg_line` shows how to embed attributes and function key names in messages.

See Also

`sm_d_msg_line`, `sm_setstatus`

sm_setsibling

Specifies to open the next screen as a sibling of the current window

```
void sm_setsibling(void);
```

Description

`sm_setsibling` forces sibling status onto the next screen opened as a window. Usually, you can open a screen as a sibling window by prepending the screen name with double ampersands (`&&`) in a control string—for example, in a widget's Control String property or as an argument to `sm_jwindow`. This operation fails if the specified screen is already open as the current window or as a sibling of the current window. If you want to open multiple instances of the same screen as sibling windows, precede each call that opens these windows with a call to `sm_setsibling`.

Also, you can use this function to set sibling status for a screen to be opened with `sm_r_window`, `sm_r_at_cur`, or one of their variants. Otherwise, Panther opens all windows opened by these functions as stacked windows.

To change stacked windows into siblings and vice-versa, set their `sibling` property to `PV_YES` and `PV_NO`, respectively.

Note: `sm_setsibling` temporarily sets a static variable that is immediately unset after the next window-open operation, even if the operation fails. All subsequent window-open operations revert to their default behavior.

sm_setstatus

Turns alternating background status message on or off

```
void sm_setstatus(int mode);
```

`mode`

Specifies whether to turn the alternating status message on or off:

1	Turns the status message on.
0	Turns the status message off.

Environment

Character-mode

Description

When alternating messages are turned on, one message—typically `Ready`–displays on the status line while Panther awaits input, and another—normally `Wait`–when it is not. If `mode` is 0, the messages are turned off.

The status flags are replaced temporarily by messages passed to `sm_ferr_reset` and related functions. They overwrite messages posted by `sm_d_msg_line` and `sm_setbkstat`.

You can edit the text of alternating messages in the message file, where they are stored as `SM_READY` and `SM_WAIT`. You can also embed attribute changes and function key names in these messages, as described in `sm_d_msg_line`.

Example

```
#include <smdefs.h>
   #include <smerror.h>
   #define PAUSE (sm_flush(), sleep(3))
   char buf[100];

   /* Tell people what you're going to tell them. */
   sprintf (buf, "You will soon see %s alternating "
      "with %s below.",
      sm_msg_get(SM_READY), sm_msg_get(SM_WAIT));
   sm_do_region(3, 0, 80, WHITE, buf);

   /* Now tell them. */
   sm_setstatus(1);
   PAUSE;          /* Shows WAIT */
```

```
sm_input(IN_DATA);     /* Shows READY */

/* Finally, tell them what you told them. */
sprintf(buf, "That was %s alternating with %s "
   "on the status line.",
   sm_msg_get(SM_READY), sm_msg_get(SM_WAIT));
sm_ferr_reset (0, buf);
```

See Also `sm_setbkstat`

sm_sh_off

Gets the cursor location relative to the start of a shifting field

```
int sm_sh_off(void);
```

Returns

≥0	The difference between the current cursor position and the start of shiftable data in the current field.
-1	The cursor is not in a field.

Description

`sm_sh_off` returns the difference between the start of data in a shiftable field and the current cursor location. If the current field is not shiftable, it returns the difference between the field's leftmost column and the current cursor location.

Example

```
#include <smdefs.h>

    /* Fancy test to see whether the current field is shifted
     * to the left. */

    if (sm_sh_off() != sm_disp_off())
       sm_ferr_reset(0, "Ha! You shifted!");
```

See Also

`sm_disp_off`, `sm_off_gofield`

sm_shell

Executes a system call

```
int sm_shell(char *cmdstr, int wait);
```

`cmdstr`
: The operating system command to execute; its syntax is system-dependent.

`wait`
: Used only in character mode, specifies whether to display an acknowledgement message before returning to the Panther application:

- 1 (Yes): Display a message that the user must acknowledge before the Panther application resumes execution.
- 0 (No): Return immediately to the Panther application after `cmdstr` executes. Panther refreshes the screen and resumes screen processing.

Returns System-dependent.

Description In character mode, `sm_shell` clears the screen and displays any output from the specified program; on GUI platforms, display output is system-dependent.

Return values are system-dependent. For example, UNIX systems typically supply `sm_shell` with the executed command's return value and reason for exiting by shifting and OR'ing two values together; under Windows, the WinExec Windows API function is used and a return value greater than 31 indicates success.

On Windows, the command string must contain a filename. For example, to run batch files, specify the file extension, as in `run.bat`. To generate a directory listing using the `dir` command, specify:

```
command.com /c dir
```

Example

```
# On a UNIX system, check a directory listing.
    call sm_shell("ls -l", 1)
    #open a file...
```

See Also `jm_system`, `sm_launch`

sm_shrink_to_fit

Removes trailing empty array elements and shrinks the screen

```
void sm_shrink_to_fit(void);
```

Description

`sm_shrink_to_fit` lets you dynamically reduce the current screen size according to the number of array elements that contain data at runtime. This function removes the trailing elements in all arrays on a screen and then shrinks the screen to a size just large enough to accommodate the displayed data. If there is no data in the array, then the entire array is removed. Only the currently displayed copy of the screen in memory is altered.

`sm_shrink_to_fit` never minimizes screen size at the expense of the screen's first or last line. For example, given a five-line screen with a five-element array in which only four elements have data, `sm_shrink_to_fit` leaves the last empty element alone because it occupies the screen's last line.

Example

```
/* Put ^shrink in the auto control */
   /* to have window shrink to fit before */
   /* user gets a chance to see it! */

int
   shrink(ignored_data)
   char *ignored_data;
   {
      sm_shrink_to_fit();
      return (0);
   }
```

sm_slib_error

Gets the system return for the last call to `sm_slib_load`

```
int sm_slib_error(void);
```

Environment	Windows
Description	`sm_slib_error` gets the system-specific error value set when a DLL is loaded by `sm_slib_load`.
See Also	`sm_slib_load`

sm_slib_install

Installs a function from a DLL into a Panther application

```
int sm_slib_install(char *fnc_spec, int language,
   int return_type);
```

`fnc_spec`
: A string that includes the name of the function to install and a comma-delimited list of its argument types enclosed in parentheses:

```
"func-name(param-list)"
```

Panther supports string and integer arguments, specified by `s` and `i`, respectively. Specify any combination of strings and integers from zero to five arguments. Panther also supports functions with six integer arguments.

For example, this statement installs the Windows library function FindWindow, which expects two string arguments:

```
err = sm_slib_install
   ("FindWindow(s,s)", SLIB_C, SLIB_INTFNC);
```

`language`
: Specifies which language calling convention to use when pushing this function's arguments onto the code stack. The convention that you specify must conform to the order in which the function expects to find its arguments stacked. Supply one of these identifiers:

`SLIB_C`
: Arguments are pushed onto the stack in left-to-right order. Windows functions usually use this convention.

`SLIB_PASCAL`
: Arguments are pushed onto the stack in right-to-left order. Microsoft Visual C++ does not support this convention.

`return_type`
: The installed function's return type, specified by one of these arguments:

```
SLIB_INTFNC
SLIB_STRFNC
SLIB_DBLFNC
```

`SLIB_ZROFNC`
: `SLIB_ZROFNC` specifies to ignore the installed function's return value and always to return 0.

Environment Windows

Returns

0	Success.
-1	Cannot find function in the loaded libraries.
-2	Invalid argument.

Description `sm_slib_install` installs the specified function from a shared library previously installed by `sm_slib_load`. Panther searches for the function in all libraries loaded by `sm_slib_load`, starting with the one most recently loaded. This function is installed as a prototyped function and can be called directly from JPL modules.

Note: In the Windows distribution, Panther automatically loads the DLLs `KERNEL32` and `USER32` in that order. All functions in these libraries are available for installation.

See Also `sm_slib_load`

sm_slib_load

Loads a dynamic link library (DLL)

```
int sm_slib_load(char *lib_name);
```

`lib_name`
: The name of the dynamic library to load. The name can include its path. If `lib_name` is already loaded, Panther moves the library to the top of the stack of loaded libraries.

Environment Windows

Returns

- 0 Success.
- -1 Unable to load `lib_name`. Call `sm_slib_error` to get the system-specific error code.

Description `sm_slib_load` makes the functions and other resources in `lib_name` available for installation. Resources can include bitmaps and icons. The library must be sharable—on Windows, a dynamic link library (DLL). To install a function from a loaded library, call `sm_slib_install`. After a function is installed, it can be called directly from a JPL module.

If the argument supplied for `lib_name` omits a path, Panther searches for the library in these locations:

1. Panther's working directory
2. Windows directory
3. Windows system directory
4. The executable's startup directory
5. `SMPATH`
6. The list of directories mapped in a network

Note: In the Windows distribution, Panther automatically loads the DLLs `KEYBOARD`, `KERNEL`, and `USER`, in that order. All functions in these libraries are available for installation.

All loaded libraries are automatically unloaded on program exit.

See Also `sm_slib_install`

sm_soption

Sets a string variable

```
char *sm_soption(int option, char *newval);
```

`option`

Specifies the variable to set with one of these constants:

`SO_EDITOR`
: Editor to use in JPL windows. Equivalent to `SMEDITOR`.

`SO_LPRINT`
: The operating system command that is invoked through the local print key (LP). Equivalent to `SMLPRINT`. Set this option only for character-mode applications.

`SO_PATH`
: Search path for libraries. Equivalent to `SMPATH`.

`SO_LDBLIBNAME`
: An LDB library to open. Equivalent to `SMLDBLIBNAME`. Set this option in `jmain.c` or `jxmain.c` before the call to `sm_ldb_init`.

`SO_LDBNAME`
: An LDB screen to open. Equivalent to `SMLDBNAME`.

`SO_RBCONFIG`
: Specifies the middleware configuration file for a running application, required to enable a native client connection. Equivalent to `SMRBCONFIG`. Set `newval` to the full pathname of the middleware configuration file—for example, `/usr/myapp/broker.bin`.

`SO_RBHOST`
: Provides the middleware API with the network addresses of the machines to which a workstation client can connect. Equivalent to `SMRBHOST`.

`SO_RBPORT`
: Provides the middleware API with the port numbers associated with the machines (`SMRBHOST`) to which a workstation client can connect. Equivalent to `SMRBPORT`.

`newval`
: The new value to assign to the `option`-specified variable.

Returns

- A pointer to the old value for the specified option or an empty string if the specified option was not set.
- 0: The option is invalid or a malloc error occurred.

Description

`sm_soption` lets you change at runtime various application-level variables that are typically defined in the application's environment, its initialization file, or setup files.

Example

```
char *default_lp;
   default_lp = sm_soption(SO_LPRINT, "lp -dny %s");
```

See Also

`sm_option`

sm_strdup

Allocate memory and copy a string to that memory

```
char *sm_strdup(char *string);
```

`string`
The string to be duplicated.

Returns .

- The value returned by sm_`fmalloc`.
- the null pointer if `string` is the null pointer or if the call to `sm_fmalloc` failed.

Description .If `string` is not the null pointer, its length is found, `sm_fmalloc` is called and if it did not return the null pointer, `string` is copied to the allocated memory. Memory allocated by calling `sm_strdup` should be freed by calling `sm_ffree`.

See Also `sm_ffree`, `sm_fmalloc`

sm_*strip_amt_ptr

Strips non-digit characters from a string

```
char *sm_strip_amt_ptr(int field_number, char *inbuf);
char *sm_e_strip_amt_ptr(char *field_name, int element,
   char *inbuf);
char *sm_i_strip_amt_ptr(char *field_name, int occurrence,
   char *inbuf);
char *sm_n_strip_amt_ptr(char *field_name, char *inbuf);
char *sm_o_strip_amt_ptr(int field_number, int occurrence,
   char *inbuf);
```

`field_name, field_number`

The field with the string to strip. You must also set parameter `inbuf` to `NULL`. For example, this JPL statement puts the unformatted contents of field `sale_amt` into variable `amt`:

```
vars amt
amt = sm_n_strip_amt_ptr("sale_amt", @NULL)
```

If `inbuf` contains the string to strip, supply this parameter with an argument of `NULL`.

`element`

The element with the string to strip.

`occurrence`

The occurrence with the string to strip.

`inbuf`

Contains the string to strip. For example, this JPL statement strips the supplied string of its currency symbol and comma and puts `123489.12` into `amt`:

```
amt = sm_strip_amt_ptr(@NULL, "$123,489.12")
```

To use the data in `field_name`/`field_number`, supply `NULL`.

Returns

- A pointer to a string that contains the stripped text.
- 0 if `inbuf` is set to `NULL` and the field number is invalid.

Description

`sm_strip_amt_ptr` strips all leading zeros and non-digit characters from the string, except for an optional leading minus sign and decimal point. If you supply a value for `inbuf` `sm_strip_amt_ptr` processes its contents. Otherwise, it uses the field data.

This function identifies the decimal character to preserve from the widget's decimal_symbol property. If this property is not set, `sm_strip_amt_ptr` uses the character that is set by the message file's `SM_DECIMAL` entry (refer to "Decimal Symbols" on page 45-23 in *Application Development Guide*).

Note: `sm_strip_amt_ptr` stores its return value in a pool of buffers that it shares with other functions. Consequently, you should use this data immediately.

See Also

`sm_amt_format`, `sm_dblval`

sm_sv_data

Saves partial screen contents

```
char *sm_sv_data(int first_field, int last_field);
```

`first_field, last_field`
: Specifies the area to save. All data between first `first_field` and `last_field`, inclusive, is saved to the specified address.

Environment C only

Returns

- The address of an area containing the saved data.
- 0: The current screen has no fields, `first_field` or `last_field` is invalid, or insufficient free memory.

Description `sm_sv_data` saves the current screen's data from all fields numbered from `first_field` to `last_field` for external access or subsequent retrieval. Use `sm_rs_data` to restore the saved data to the screen.

Data items are stored as null-terminated character strings. The contents of a scrollable array is preceded by 2 bytes giving the total number of items saved (high order byte first); each item is preceded by two bytes of display attribute, and followed by a null. There is an additional null following all the scrolling data.

See Also `sm_rs_data`, `sm_save_data`, `sm_sv_free`

sm_sv_free

Frees a buffer that contains saved screen data

```
void sm_sv_free(char *buffer);
```

`buffer`
: The address of the buffer to free.

Environment C only

Description `sm_sv_free` releases the save area at `buffer`, created by `sm_save_data` or `sm_sv_data`. Once released, this data is no longer accessible.

`sm_save_data` and related functions record up to 10 save area addresses. If you save more than 10 times during a Panther session, Panther frees existing buffers on a first-in/first-out basis. Consequently, you should use this function only if you need to manipulate the save buffers manually.

See Also `sm_save_data`, `sm_sv_data`

sm_svscreen

Registers a list of screens on the save list

```
int sm_svscreen(char **screen_list, int count);
```

`screen_list`
: Specifies the screens to add to the save list.

`count`
: The number of screens to add to `screen_list`.

Environment C only

Returns

0	Success.
1	Failure: Insufficient memory.

Description `sm_svscreen` adds screens to the Panther-managed list of screens that are saved in memory. You can call this function to add screens to this list anywhere in your code; however, these screens and the data entered in them are saved in memory only when you close the screens for the first time. Consequently, access to the saved screens is more efficient only on subsequent opens of those screens.

If a screen is already on the save list, Panther leaves that list entry unchanged. You can remove screens from the list with `sm_unsvscreen`. To check whether a screen is on the save list, use `sm_issv`.

This function saves processing time at the expense of memory. It is especially useful with read-only screens that use large amounts of external data, for example, from databases or other files. For instance, use this function to save in memory a help screen that gets its data from a database and is repeatedly opened.

Example

```
/*   sm_issv */
/*   sm_svscreen */
/*   sm_unsvscreen */
   char *screens[] =
   {
      "start.scr",
      "demo.scr",
      "help.scr"
   };
```

```
int num_screens = sizeof(screens) / sizeof(char *);

void
save_screens()
{
   /* Put 'screens' onto the save list. */
   sm_svscreen(screens, num_screens);
}

void
release_screens()
{
   /* Remove 'screens' from the save list. */
   sm_unsvscreen(screens, num_screens);
}

void
release_screen(name)
char *name;
{
   char *temp[1];
   if (sm_issv(name))
   {
      temp[0] = name;
      sm_unsvscreen(temp, 1);
   }
}
```

See Also `sm_issv`, `sm_unsvscreen`

sm_tab

Moves the cursor to the next unprotected field

```
void sm_tab(void);
```

Description

`sm_tab` moves the cursor to first enterable position in the next tab-accessible field on the screen. If the cursor is in a field with a next-field property and one of the fields specified by the property is tab-accessible, the cursor moves to that field’s first enterable position. This function is normally bound to the TAB key.

This function does not immediately trigger field entry, exit, or validation processing. Such processing occurs based on the cursor position when control returns to `sm_input`.

Example

```
#include <smkeys.h>

    /* This moves the cursor to the next field. */
    sm_tab();
```

See Also

`sm_backtab`, `sm_home`, `sm_last`, `sm_nl`

sm_tm_clear

Clears all fields in the table view

```
#include <tmusubs.h>
int sm_tm_clear(int suppress);
```

`suppress`

A flag that, if set to other than 0, indicates that before-image processing should be suppressed while the clearing is being done.

Returns

0	Success.
<0	Failure.

Description

`sm_tm_clear` clears all fields in the current table view. A positive value of `suppress` indicates that before-image processing should be suppressed while the clearing is being done.

sm_tm_clear_model_events

Empties the transaction event stack

```
#include <tmusubs.h>
void sm_tm_clear_model_events(void);
```

Description

`sm_tm_clear_model_events` clears the transaction event stack. Events can be pushed onto the event stack by the transaction manager, a transaction model, or a user event function. The events generated by the transaction manager and those by the standard transaction models can be found in the include file `tmusubs.h`.

This function can be used by transaction models or by transaction event functions associated with a table view.

For more information on the event stack, refer to Chapter 35, “Generating Transaction Manager Events,” in *Application Development Guide*.

See Also

`sm_tm_push_model_event`, `sm_tm_pop_model_event`

sm_tm_command

Executes a transaction command

```
#include <tmusubs.h>
int sm_tm_command(char *cmd_string);
```

`cmd_string`

Contains one of the following transaction commands and its associated parameters:

`CHANGE`	`CONTINUE_DOWN`	`COPY_FOR_VIEW`	`REFRESH`
`CLEAR`	`CONTINUE_TOP`	`FETCH`	`SAVE`
`CLOSE`	`CONTINUE_UP`	`FINISH`	`SELECT`
`CONTINUE`	`COPY`	`FORCE_CLOSE`	`START`
`CONTINUE_BOTTOM`	`COPY_FOR_UPDATE`	`NEW`	`VIEW`

When specifying a command, the table view name is case sensitive; however, the command name and the optional parameters following the table view name are not case sensitive.

Returns

- `STATUS` of the current transaction.

-1 Unable to execute command because transaction is already in progress.

Description

`sm_tm_command` executes the specified transaction manager command. Before the command is processed, a test is performed to see if the specified command is available with the current mode.

By definition, a command is in progress from the moment `sm_tm_command` is called until the moment it returns. As it processes most commands, `sm_tm_command` invokes transaction event functions and transaction models. These, in turn, should not invoke transaction manager commands, because the transaction manager cannot process its commands recursively. This implies that they should not close the active screen (which triggers a `FINISH` command), or cause any other screen to be displayed that contains table views (which triggers a `CHANGE` command).

For the transaction command `START`, the command keyword is followed by the transaction name and can also be followed by a table view name.

```
int sm_tm_command(START transactionName [tableViewName]);
```

For the transaction command `CHANGE`, the command keyword is followed by the transaction name.

```
int sm_tm_command(CHANGE transactionName);
```

For other transaction commands, the transaction name is set by the previous `START` or `CHANGE` command and the parameter following the command is interpreted as a table view name.

If there is an additional `scope` parameter, it specifies a portion of the table view tree. The command is then applied only to those table views.

```
int sm_tm_command(command[ tableViewName][ scope]);
```

The *`scope`* parameter must be preceded by a table view name and takes one of these arguments:

`TV_AND_BELOW`
: Applies the command to the specified table view and all table views below it on the tree. If no parameter is specified, the transaction manager acts as though `TV_AND_BELOW` was supplied.

`BELOW_TV`
: Applies the command to the table views below the specified table view.

`TV_ONLY`
: Applies the command to the specified table view only.

`SV_ONLY`
: Applies the command only to the table views of the specified server view.

Special processing occurs for the `FETCH` command. For `FETCH`, the *`scope`* parameter is either `FETCH_SIMPLE` or `FETCH_SPECIAL` which specifies the type of fetch processing.

Example

```
int sm_tm_command("SELECT titles BELOW_TV");
```

See Also

For the syntax of transaction manager commands, refer to Chapter 8, “Transaction Manager Commands.”

sm_tm_command_emsgset

Initiates error message processing for a transaction manager error code

```
#include <tmusubs.h>
int sm_tm_command_emsgset(char *caller_id, int code);
```

`caller_id`
: A string used for identification; in Panther transaction models this is set to the module name followed by the function name where the event was triggered.

`code`
: One of the transaction manager `DM_TM_ERR_XXXX` return codes.

Returns `STATUS` value of the current transaction.

Description `sm_tm_command_emsgset` reports an error to the transaction manager error processor (`sm_tm_error`). `code` is one of the `DM_TM_ERR_XXX` return codes returned from `sm_tm_command`. The error severity level is set to `TM_EMSG`. The error text generated corresponds to the error message for code.

If the `TM_STATUS` value of the current transaction is 0, this function sets `TM_STATUS` to -1. If both `TM_STATUS` and `TM_MSG` values of the current transaction are 0, this function sets `TM_MSG` to the value of `code`.

sm_tm_command_errset

Initiates error processing for a transaction manager error code

```
#include <tmusubs.h>
int sm_tm_command_errset(char *caller_id, int code);
```

`caller_id`
: A string used for identification; in Panther transactions models this is set to the module name followed by the function name where the event was triggered.

`code`
: One of the transaction manager `DM_TM_ERR_XXXX` return codes.

Returns `STATUS` value of the current transaction.

Description `sm_tm_command_errset` reports an error to the transaction manager error processor (`sm_tm_error`). `code` is one of the `DM_TM_ERR_XXX` return codes returned from `sm_tm_command`. The error severity level is set to `TM_ERROR`. The error text generated corresponds to the error message for code.

If the `TM_STATUS` value of the current transaction is 0, this function sets `TM_STATUS` to -1. If both `TM_STATUS` and `TM_MSG` values of the current transaction are 0, this function sets `TM_MSG` to the value of `code`.

sm_tm_continuation_validity

Checks to see if the CONTINUE events are permitted for the current table view

```
#include <tmusubs.h>
int sm_tm_continuation_validity(int report);
```

`report`
: Controls whether an error message is generated. If this parameter is non-zero, the message is generated.

Returns

- `TM_OK` if the `TM_CONTINUE_UP`, `TM_CONTINUE_TOP`, `TM_CONTINUE_DOWN`, and `TM_CONTINUE_BOTTOM` events are permitted.
- `TM_FAILURE` if these events are not permitted. If `report` is non-zero, an error message is also generated.

Description

This function is used in the standard transaction models for two-tier applications as part of the transaction manager processing for the `SELECT` and `VIEW` commands. It checks the value of the `fetch_directions` property for the current table view to see if the `TM_CONTINUE_UP`, `TM_CONTINUE_TOP`, `TM_CONTINUE_DOWN`, and `TM_CONTINUE_BOTTOM` events are permitted.

If the `fetch_directions` property is specified as `Down only-all modes` (`PV_CONT_NEVER`), only `TM_CONTINUE` fetches additional data. `TM_CONTINUE_DOWN` is not permitted.

If the `fetch_directions` property is specified as `Up/Down-view mode` (`PV_CONT_VIEW_ONLY`), the `TM_CONTINUE_UP`, `TM_CONTINUE_TOP`, `TM_CONTINUE_DOWN`, and `TM_CONTINUE_BOTTOM` events are allowed in addition to `TM_CONTINUE` if the current transaction mode is `view`.

If the `fetch_directions` property is specified as `Up/Down-all modes` (`PV_CONT_ALWAYS`), the `TM_CONTINUE_UP`, `TM_CONTINUE_TOP`, `TM_CONTINUE_DOWN`, and `TM_CONTINUE_BOTTOM` events are allowed in addition to `TM_CONTINUE` in `view` and `update` mode. Data must be re-fetched in order for updates to be displayed from the continuation file used with these events.

If the table view's `fetch_directions` property is specified as `default` (`PV_CONT_DEFAULT`), the screen's Fetch Directions property is consulted. If the screen's Fetch Directions property is specified as `default`, this is the equivalent of `Down only-all modes`.

sm_tm_dbi_checker

Tests for common database errors during transaction manager processing

```
#include <tmusubs.h>
int sm_tm_dbi_checker(int event);
```

`event`

TM_TEST_ERROR to check for database errors, TM_TEST_ONE_ROW to check that one row was affected by the processing, or TM_TEST_SOME_ROWS to check that one or more rows was affected by the processing.

Returns

- TM_FAILURE:
  - If a database error is recognized.
  - If no database error is recognized but event is TM_TEST_ONE_ROW and more than one row has been affected by database interface processing.
  - If no database error is recognized but event is TM_TEST_SOME_ROWS and no rows have been affected by database interface processing.
- TM_OK if no database error has been recognized, nor an error because of an event condition as described above.

Description

sm_tm_dbi_checker tests the Panther database variables @dmretcode and @dmengerrcode for any errors in database processing. If it finds an error, it logs it and sets error messages.

If no database errors are encountered but event is TM_TEST_ONE_ROW, sm_tm_dbi_checker returns the error status TM_FAILURE if @dmrowcount is not 1.

Similarly, if event is TM_TEST_SOME_ROWS, sm_tm_dbi_checker returns the error status TM_FAILURE if @dmrowcount is 0.

Example

```
/* The following example taken from the standard
   transaction model for JDB shows the processing for
   these events. */

   case TM_TEST_ERROR:
   case TM_TEST_ONE_ROW:
   case TM_TEST_SOME_ROWS:
      retcode = sm_tm_dbi_checker(event);
      break;
```

sm_tm_error

Reports an error condition

```
#include <tmusubs.h>
void sm_tm_error(char *caller_id, char *text, char *user_use,
    int severity);
```

`caller_id`
: A string used for identification; in Panther transactions models this is set to the module name followed by the function name where the event was triggered.

`text`
: Null-terminated character string containing a message.

`user_use`
: Null-terminated character string for user's message.

`severity`
: Severity level of the error.

Description

`sm_tm_error` reports an error to the transaction manager error processor. The error is identified by the `caller_id`, `text`, `user_use` (if one exists) and `severity`. Errors are written to an error log file. If an error log file is not specified or if `severity` is less than the severity limit, nothing is written.

The character string parameters can contain white space but the first `NULL` character indicates the end of the string.

See Also

`sm_tm_errorlog`, `sm_tm_msg_count_error`, `sm_tm_msg_emsg`, `sm_tm_command_emsgset`, `sm_tm_command_errset`

sm_tm_errorlog

Controls error log processing

```
#include <tmusubs.h>
int sm_tm_errorlog(int call_type, int call_type_code,
   char *log_file);
```

`call_type`
: Determines which aspect of error log processing is affected. One of the following constants: `TM_ERR_KEEP`, `TM_ERR_SUPPRESS`, `TM_ERR_FILE`, `TM_ERR_NEW_COMMAND`, as defined below.

`call_type_code`
: A value that is used depending on the argument supplied for `call_type`.

`log_file`
: The name of the file in which the error log is maintained; ignored unless `call_type` is `TM_ERR_FILE`.

Returns

0	Success.
-1	Failure.

Description

`sm_tm_errorlog` controls error log processing according to the value of `call_type`.

`TM_ERR_KEEP` specifies the existence of the error log. A `call_type_code` value of 0 clears the error log when a new command begins processing. A value of 1 indicates not to clear out the log. The last parameter is ignored.

`TM_ERR_SUPPRESS` specifies which errors to suppress depending on a severity level, where `call_type_code` determines which errors to suppress. Any errors passed to `sm_tm_error` with a severity greater than 0 and less than or equal to `call_type_code` are not logged. If `call_type_code` is 0, there is no suppression.

`TM_ERR_FILE` specifies the error log file that is named by `log_file`. The file is appended to if it exists, but a call to this function with `TM_ERR_KEEP` might override this. If there is no call to `sm_tm_errorlog`, there is no `sm_tm_error` error logging.

If `call_type_code` is 0, the file is not flushed or closed unnecessarily after it is written to. If `call_type_code` is 1, the file is closed after each entry is written.

If `log_file` is a null pointer or empty string, there is no further error logging until a subsequent call to `sm_tm_errorlog` reinstates it.

`TM_ERR_NEW_COMMAND` specifies that processing of a new command is starting.

See Also `sm_tm_error`

sm_tm_event

Returns the event number for the specified transaction manager event name

```
#include <tmusubs.h>
int sm_tm_event(char *event_name);
```

`event_name`
: One of the names in the table of request and event numbers defined in `tmusubs.h`.

Returns

- The number for the specified event.
- 0 Error: `event_name` is not found.

Description

`sm_tm_event` returns the event number corresponding to the specified transaction manager event name. As part of its processing, `event_name` is converted from lower case letters to upper case.

Example

```
int process_event (event_name, caller_id)
   char* event_name, caller_id;
   {
      int event_num;
      if ((event_num = sm_tm_event(event_name)) != 0)
         switch (event_num)
         {
            case TM_SELECT: ...
               break;
            case TM_PRE_SELECT: ...
               break;
            case TM_POST_SELECT: ...
               break;
            ...
            case TM_NOTE_FAILURE:
            case TM_NOTE_UNSUPPORTED:
               sm_tm_failure_message(event_num, caller_id,
                  "Unsupported or failed event");
               break;
            default:
               sm_tm_failure_message(event_num, caller_id,
                  "Unsupported event");
         }
      else /* returned '0', event_name unrecognized */
      {
         char buf[255];
```

```
            sprintf(buf, "Bad TM event name: %s", event_name);
            sm_tm_error(caller_id, buf, "", TM_WARNING);
            buf[0]='\0';
        }
    }
```

sm_tm_event_name

Returns the transaction manager event name for the specified event number

```
#include <tmusubs.h>
char *sm_tm_event_name(int event_number);
```

`event_number`
One of the request and event numbers defined in `tmusubs.h`.

Returns

- A pointer to a string that contains the name of the specified event number.
- A string that contains the event number if the number does not correspond to one of the events

Description

`sm_tm_event_name` returns the event name corresponding to the specified event number. Because this function stores the returned data in a pool of buffers that it shares with other functions, copy or process this data immediately.

Example

```
# JPL Procedure called as an event function that displays
   # each event name as it is processed.

proc getname(event)
vars retname
retname = sm_tm_event_name(event)
msg_emsg "Event name is " retname
return TM_PROCEED
```

sm_tm_failure_message

Specifies an error message to report for a transaction manager error

```
#include <tmusubs.h>
int sm_tm_failure_message(int type, char *caller_id, char *text);
```

`type`
: The event calling this function. This event must be `TM_NOTE_FAILURE` or `TM_NOTE_UNSUPPORTED`.

`caller_id`
: Identifier for the calling program. If this is not supplied, the generated `caller_id` has embedded in it the previous event name or number and the previous transaction model or transaction event function name.

`text`
: The text for the error message. If this is not supplied, a generic error message is generated.

Returns

- `TM_OK`

Description

When the transaction manager generates either the `TM_NOTE_FAILURE` or the `TM_NOTE_UNSUPPORTED` event, the standard transaction models call `sm_tm_failure_message` to generate an error message for the previous event.

`sm_tm_failure_message` checks the value of `TM_STATUS` and sets it to -1 if the value is 0.

Example

```
/* The following example taken from the standard
   transaction model for JDB shows the processing for
   these events. */

   case TM_NOTE_FAILURE:
   case TM_NOTE_UNSUPPORTED:
      retcode = sm_tm_failure_message(event, "", "");
      break;
```

sm_tm_handling

Processes a handling method property

```
#include <tmusubs.h>
int sm_tm_handling(int prop);
```

`prop`
    One of the property constants.

Returns

- `TM_PROCEED` if the property value specifies something other than function invocation or doing nothing, or if `dm_tm_listing_sql` is being done.
- `TM_FAILURE` if there is an error calling an invoked function or if an invoked function returns non-zero.
- `TM_OK` otherwise.

Description

`sm_tm_handling` analyzes and, in some cases, processes the handling specified (indirectly) by the `prop` parameter for the specified table view. If the parameter is `DM_SEL_FUNC_NAME`, `DM_INS_FUNC_NAME`, `DM_UPD_FUNC_NAME` or `DM_DEL_FUNC_NAME`, the handling is that of the corresponding `PR_xxx_HANDLING` property. If `prop` is `DM_CONTINUE_FUNC_NAME`, the choice of behavior is based on `DM_SELECT_HANDLING`, but the function invoked, if any, is the `CONTINUE` function. If `prop` is `DM_SAVE_FUNC_NAME`, the function (if any) specified by that property is invoked if any of the insert, update or delete handling properties specify function invocation. Otherwise `TM_OK` is returned.

If the property value specifies to do nothing, this routine simply returns `TM_OK`.

If the property value specifies to invoke a function, this routine calls it; the function has no parameters. If the function returns a non-zero integer, and the `TM_STATUS` member of `sm_tm_curinfo` is zero, this routine puts that value there; it similarly proposes a generalized error message, if none has been set up. The return value from this routine is `TM_FAILURE` (for an error on the function call or a non-zero return from the function), or `TM_OK` for a successful call with a zero value returned by the function. Absence of the corresponding function name value is an error, except for `CONTINUE` and `SAVE` functions. However, inability to find a function whose name has been specified as a property value is always an error. For this and other serious processing errors in function invocation, this routine reports an error and displays an error message.

Otherwise this routine returns `TM_PROCEED`.

Since the handling properties are stored internally as strings, the property API access functions are used to get the more convenient `PV_` integers.

If `dm_tm_listing_sql` is being done, this routine always returns `TM_PROCEED`.

sm_tm_inquire

Gets an integer attribute of the current transaction

```
#include <tmusubs.h>
int sm_tm_inquire(int attribute);
```

`attribute`
: Specifies the integer attribute of the current transaction to get with one of the constants shown in the Description section.

Returns

≥1	The current value of `attribute`.
0	The current transaction is null.
-1	Invalid argument supplied for `attribute`.

Description

`sm_tm_inquire` gets the value of an integer attribute of the current transaction. This includes the data in the current transaction structure itself and data that can be found indirectly—for example, information about the current table view.

Supply one of the following constants to specify the desired transaction attribute:

`TM_AT_OR_BELOW`
: Traversal specifier.

`TM_CANCEL_ON_DISCARD`
: Gets cursor-related behavior that is associated with the transaction event `TM_FINISH`. The default setting is 1, which ensures that all cursor-associated locks are released when the `FINISH` command executes. For behavior that is backward compatible, call `sm_iset` and supply a value of 0—`sm_iset(TM_CANCEL_ON_DISCARD, 0)`. This change affects all outstanding and subsequent transactions for the current database connection.

`TM_CONTINUATION`
: Value of `fetch_directions` property for current table view: `PV_CONT_DEFAULT`, `PV_CONT_ALWAYS`, `PV_CONT_NEVER`, `PV_CONT_VIEW_ONLY`.

`TM_CURRENT_COMMAND`
: Identifies the current transaction event—for example, `TM_SELECT` during `CLOSE` processing of a `SELECT` command.

`TM_CURRENT_MODE`
Current transaction mode.

`TM_CURRENT_OCC`
Current occurrence number of current table view.

`TM_CURRENT_REQUEST`
Current request being processed. Use `sm_tm_event_name` to get the string equivalent.

`TM_EMSG_USED`
Error message indicator.

`TM_FULL`
Full or partial command indicator.

`TM_HOOK_IN_USE`
Indicates whether a transaction model or transaction event function is in use. Values include:

`TM_NOTHING_IN_USE`	Nothing in use
`TM_MODEL_IN_USE`	Transaction model in use
`TM_UHOOK_IN_USE`	Event function in use

`TM_LINK`
Link from a table view to its parent table view.

`TM_MSG`
User specified message code to use for exit condition after a call to `sm_tm_command`.

`TM_OCC`
Occurrence number being processed.

`TM_OCC_COUNT`
The number of occurrences in the table view.

`TM_OCC_TYPE`
Code reflecting the nature of change, if any, of an occurrence from its before-image. The codes are listed in “Determining How Screen Data Has Changed” on page 36-26 in the Application Development G uide.

`TM_PARENT_OCC`
Current occurrence of parent of current table view.

`TM_PARENTING_OCC`
: Occurrence that was valid in parent when table view last fetched.

`TM_PREVIOUS_EVENT`
: Indicates the previous transaction manager event. Used when writing an error handler to log the event which generated the error.

`TM_PREVIOUS_HOOK_IN_USE`
: Indicates whether the transaction model or an event function was used in the previous event. Used when writing an error handler. Values include:

`TM_NOTHING_IN_USE`	Nothing in use
`TM_MODEL_PREV_IN_USE`	Transaction model used for previous event
`TM_UHOOK_PREV_IN_USE`	Hook function used for previous event

`TM_QUERY_ACTION`
: Return code from `TM_QUERY`. Models return:

`TM_DISCARD_ACTION`	Discard changes
`TM_EXIT_ACTION`	Return to screen without discarding changes

`TM_SAVE_COUNT`
: When supplied this argument, `sm_tm_inquire` returns the number of rows that the transaction manager asked the transaction model to save to the database.

Note: The value returned by `sm_tm_inquire(TM_SAVE_COUNT)` is not equivalent to the number of SQL statements issued, inasmuch as multiple SQL statements can be issued for each row.

If an error occurs during save processing, `sm_tm_inquire` returns 0 and a `DBMS ROLLBACK` is executed.

The following example is in `smwizard.jpl`:

```
// If new row was added, allow user to work with it.
// Otherwise, place TM back into INITIAL mode.

if (sm_tm_inquire(TM_SAVE_COUNT) > 0)
{
```

```
         call sm_tm_command("COPY_FOR_UPDATE")
     }
     else
     {
         call sm_tm_command("FORCE_CLOSE")
     }
```

`TM_STATUS`
: Error indicator.

`TM_SV_SEL_COUNT`
: For `SELECT` and `VIEW`, this argument is set to 1 if the Count Select property indicates that an initial query be performed to determine the number of rows in the select set.

`TM_SV_SEL_REQUEST`
: Request that gave rise to the current select cursor for the table view (either `SELECT` or `VIEW`).

`TM_USER_VALUE`
: Reserved for user use.

`TM_VALUE`, `TM_VALUE2`
: General purpose integer.

`TM_XA_TRANSACTION_BEGUN`
: For the Tuxedo middleware adapter, this argument tests whether the transaction model has started an XA transaction.

See Also `sm_tm_iset`, `sm_tm_pcopy`, `sm_tm_pinquire`, `sm_tm_pset`

sm_tm_iset

Sets the value of a transaction attribute

```
#include <tmusubs.h>
int sm_tm_iset(int attribute, int value);
```

`attribute`
: Specifies the integer attribute of the current transaction to change with one of the constants shown in Description.

`value`
: `attribute`'s new value.

Returns

0	Success.
-1	Invalid argument supplied for `attribute`.
-2	Unable to make the requested change.

Description

`sm_tm_iset` changes the value of an integer attribute of the current transaction. This includes not only data in the current transaction structure itself, but also data that can be found indirectly, such as data relating to the current table view.

Supply one of the following constants to specify the transaction attribute to be changed:

`TM_CANCEL_ON_DISCARD`
: Sets cursor-related behavior that is associated with the transaction event `TM_FINISH`. The default setting is 1, which ensures that all cursor-associated locks are released when the `FINISH` command executes. For behavior that is backward compatible, supply a value of 0. This change affects all outstanding and subsequent transactions for the current database connection.

`TM_EMSG_USED`
: If set to 1, no error message is displayed when `sm_tm_command` returns to its caller. Indicates that the error message was displayed by `sm_tm_command`.

`TM_MSG`
: User specified message code to use for exit condition after a call to `sm_tm_command`.

`TM_OCC`
: Occurrence number being processed.

`TM_OCC_COUNT`
: The number of occurrences in the table view.

`TM_PROPOSE_MSG`
: A conditional value for `TM_MSG`; used only if there is no existing value.

`TM_PROPOSE_STATUS`
: A conditional value for `TM_STATUS`; used only if there is no existing value.

`TM_QUERY_ACTION`
: Return code from `TM_QUERY`. Models return:

`TM_DISCARD_ACTION`	Discard changes
`TM_EXIT_ACTION`	Return to screen without discarding changes

`TM_STATUS`
: Error indicator.

`TM_SV_SEL_COUNT`
: Set to 1 to get the size of the select set for the server view (either `SELECT` or `VIEW`).

`TM_SV_SEL_REQUEST`
: Request that gave rise to the current select cursor for the table view (either `SELECT` or `VIEW`).

`TM_USER_VALUE`
: Reserved for user use.

`TM_VALUE`, `TM_VALUE2`
: General purpose integer.

`TM_XA_TRANSACTION_BEGUN`
: For the Tuxedo middleware adapter, this argument sets the transaction model to start an XA transaction.

See Also `sm_tm_inquire`, `sm_tm_pcopy`, `sm_tm_pinquire`, `sm_tm_pset`

sm_tm_msg_count_error

Reports a transaction manager error

```
#include <tmusubs.h>
void sm_tm_msg_count_error(char *caller_id, int msg, int count);
```

`caller_id`
: A string used for identification; in Panther transactions models, this is set to the module name followed by the function name where the event was triggered.

`msg`
: Identifier for a predefined error message.

`count`
: Any integer value useful for display in the message string.

Description

`sm_tm_msg_count_error` reports an `ERROR` severity error to the transaction manager error processor (`sm_tm_error`). The error text includes the name of the function where the error occurred identified by `caller_id`, the message text string corresponding to `msg` (obtained by a call to `sm_msg_get`), and the value identified in `count`. A typical use for count would be to display an error return code from the function that triggered the error event.

If `msg` is `DM_TM_ALREADY` or 0, this function does nothing.

See Also

`sm_tm_error`, `sm_tm_msg_emsg`, `sm_tm_msg_error`

sm_tm_msg_emsg

Reports an error of message severity

```
#include <tmusubs.h>
void sm_tm_msg_emsg(char *caller_id, int msg);
```

`caller_id`
: A string used for identification; in Panther transactions models this is set to the module name followed by the function name where the event was triggered.

`msg`
: Identifies an error message.

Description

`sm_tm_msg_emsg` reports an `EMSG` severity error to the transaction manager error processor. The error text includes the name of the function where the error occurred identified by `caller_id` and the message text string corresponding to `msg`, obtained by a call to `sm_msg_get`.

If `msg` is `DM_TM_ALREADY` or 0, this function does nothing.

See Also

`sm_tm_error`, `sm_tm_msg_count_error`, `sm_tm_msg_error`

sm_tm_msg_error

Reports an error

```
#include <tmusubs.h>
void sm_tm_msg_error(char *caller_id, int msg);
```

`caller_id`
: A string used for identification; in Panther transactions models this is set to the module name followed by the function name where the event was triggered.

`msg`
: Identifies an error message.

Description

`sm_tm_msg_error` reports an `ERROR` severity error to the transaction manager error processor. The error text includes the name of the function where the error occurred identified by `caller_id` and the message text string corresponding to `msg`, obtained by a call to `sm_msg_get`.

If `msg` is `DM_TM_ALREADY` or 0, this function does nothing.

See Also

`sm_tm_error`, `sm_tm_msg_emsg`, `sm_tm_msg_count_error`

sm_tm_old_bi_context

Sets a backward compatibility flag

```
#include <tmusubs.h>
int sm_tm_old_bi_context(int flag);
```

`flag`

The setting for fetching before-image data:

1	Set backwards compatibility flag.
0	Use the current release process (default).
-1	The current setting.

Returns

- The old value of the flag.

Description

`sm_tm_old_bi_context` sets a new value of the `old_bi_context_flag` and returns the old value. The `old_bi_context_flag` is used to determine whether to use more powerful name parsing methods than had previously been used. `sm_tm_old_bi_context` also uses this flag to decide whether it should use the old or a new algorithm to get a before image context.

Calling this routine with a parameter of 1 causes the old algorithm to be used thereafter. Calling this routine with a parameter of 0 causes the new algorithm to be used thereafter; 0 is the default.

See Also

`sm_get_tv_bi_data`

sm_tm_pcopy

Gets a string attribute of the current transaction and stores it

```
#include <tmusubs.h>
int sm_tm_pcopy(int attribute, char *attr_value, int length);
```

`attribute`
: Specifies the string attribute of the current transaction to get with one of the constants shown in Table 5-20.

`attr_value`
: A string buffer where the specified attribute's value is copied.

`length`
: Specifies the maximum length of data to copy to `attr_value`, excluding the `NULL` string terminator. If `length` has a 0 or negative value, it is set to 255.

Environment C only

Returns

0 Success.

- `DM_TM_ERR_NO_TRANSACTION`: The current transaction is null.
- `DM_TM_ERR_ARGS`: `value` is a null pointer.
- `DM_TM_ERR_BAD_MEMBER`: `attribute` is invalid.
- `DM_TM_ERR_GENERAL`: The length of `attr_value` exceeds `length` or 255.

Description

`sm_tm_pcopy` is used to obtain the current value of a string attribute of the current transaction. This includes not only data in the current transaction structure itself, but also data that can be found indirectly, such as data relating to the current table view. This function stores the value to a user-defined buffer, and returns error information.

Table 5-20 lists the constants, defined in `tmusubs.h`, that specify the string attributes to get with this function.

Table 5-20 Transaction string attributes

Transaction Attribute	Description
`TM_BUFFER`	General purpose string.
`TM_COMMAND_PARM`	Text string passed to `sm_tm_command`.

Table 5-20 Transaction string attributes

Transaction Attribute	Description
`TM_MSG_TEXT`	Text of `sm_tm_command` exit message.
`TM_COMMAND_ROOT`	Identifies the root table view of the current command. This is either the root of the tree (for the entire transaction), or the root of the partial tree specified for the current command.
`TM_PARENT_NAME`	Name of parent table view of current table view.
`TM_PREVIOUS_HOOK`	Name of the previous event function. Used when writing an error handler.
`TM_ROOT_NAME`	Name of root table view of the transaction.
`TM_SAVE_CURSOR`	SAVE or VALIDATION cursor name.
`TM_SV_NAME`	Name of server view containing current table view.
`TM_SV_SELECT_CURSOR`	SELECT cursor name.
`TM_TRAN_NAME`	Name of the current transaction.
`TM_TRANS_MODEL_NAME`	Name of the transaction model.
`TM_TV_NAME`	Name of the current table view.
`TM_USER_BUFFER`	Buffer reserved for user use.

Data is only copied if no errors are encountered.

See Also `sm_tm_inquire`, `sm_tm_iset`, `sm_tm_pinquire`, `sm_tm_pset`

sm_tm_pinquire

Gets the value of a string attribute of the current transaction for immediate use

```
#include <tmusubs.h>
char *sm_tm_pinquire(int attribute);
```

`attribute`
: Specifies the string attribute of the current transaction to copy with one of the constants defined in `tmusubs.h` and shown in the Description section.

Returns

- Success: copy of the string value of `attribute`.
- Failure: empty string.

Description

`sm_tm_pinquire` gets the current value of a string attribute of the current transaction. This includes not only data in the structure itself, but also data that can be found indirectly, such as data relating to the current table view.

An empty string is returned if any of the following errors occurs: the current transaction is null, `attribute` is invalid, the value of `attribute` is a non-existent string, or the length of the value of `attribute` is greater than 255.

Because the objects pointed to by the pointers returned by `sm_tm_pinquire` usually have short duration, as they are stored in rotating buffers, use or copy them quickly

Supply one of the following constants to specify the desired transaction attribute:

`TM_BUFFER`
: General purpose string.

`TM_COMMAND_PARM`
: Text string passed to `sm_tm_command`.

`TM_COMMAND_ROOT`
: Text of `sm_tm_command` exit message.

`TM_MSG_TEXT`
: Identifies the root table view of the current command. This is either the root of the tree (for the entire transaction), or the root of the partial tree specified for the current command.

`TM_PARENT_NAME`
: Name of parent table view of current table view.

`TM_PREVIOUS_HOOK`
: Name of the previous event function. Used when writing an error handler.

`TM_ROOT_NAME`
: Name of root table view of the transaction.

`TM_SAVE_CURSOR`
: SAVE or VALIDATION cursor name.

`TM_SV_NAME`
: Name of server view containing current table view.

`TM_SV_SELECT_CURSOR`
: SELECT cursor name.

`TM_TRAN_NAME`
: Name of the current transaction.

`TM_TRANS_MODEL_NAME`
: Name of the transaction model.

`TM_TV_NAME`
: Name of the current table view.

`TM_USER_BUFFER`
: Buffer reserved for user use.

See Also `sm_tm_inquire`, `sm_tm_iset`, `sm_tm_pset`, `sm_tm_pcopy`

sm_tm_pop_model_event

Pops an event off the transaction event stack

```
#include <tmusubs.h>
int sm_tm_pop_model_event(void);
```

Returns The event popped off the event stack.

0: The stack is empty.

Description `sm_tm_pop_model_event` pops the next event in the transaction event stack. Events can be pushed onto the event stack by the transaction manager, a transaction model, or a user event function. The events generated by the transaction manager and those by the standard transaction models can be found in the include file `tmusubs.h`.

This function can be used by transaction models or by transaction event functions associated with a table view.

See Also `sm_tm_clear_model_events`, `sm_tm_push_model_event`

sm_tm_pset

Sets the value of a string transaction attribute

```
#include <tmusubs.h>
int sm_tm_pset(int attribute, char *value);
```

`attribute`
: Specifies the string attribute of the current transaction to change with one of the constants shown below.

`value`
: `attribute`'s new value.

Returns

0	Success.
-1	Invalid argument supplied for `attribute`.
-2	Unable to make the requested change.

Description

`sm_tm_pset` changes the value of a string attribute of the current transaction. This includes not only data in the current transaction structure itself, but also data that can be found indirectly, such as data relating to the current table view.

Table 5-21 describes the constants, defined in `tmusubs.h`, that specify the attributes to change with this function.

Table 5-21 Transaction string attributes for `sm_tm_pset`

Transaction Attribute	Description
`TM_BUFFER`	General purpose string.
`TM_MSG_TEXT`	Text of `sm_tm_command` exit message.
`TM_PROPOSE_MSG_TXT`	Used to conditionally set `TM_MSG_TEXT`.
`TM_SAVE_CURSOR`	SAVE or VALIDATION cursor name.
`TM_SV_SELECT_CURSOR`	SELECT cursor name.
`TM_USER_BUFFER`	Reserved for user use.

Example

```
void set_msg_text(msg);
    char *msg;
    {
        /*
         * Set the sm_tm_command exit message, possibly overriding
         *   any previously set message.
         */
         sm_tm_pset(TM_MSG_TEXT, msg);
    }
```

See Also `sm_tm_inquire`, `sm_tm_pinquire`, `sm_tm_pcopy`, `sm_tm_pset`

sm_tm_push_model_event

Pushes an event onto the transaction event stack

```
#include <tmusubs.h>
int sm_tm_push_model_event(int event);
```

`event`
    Any transaction event.

Returns

0	Success: `event` pushed on stack and stack was not full.
-1	`event` is 0.
•	The event value pushed off the stack because the stack was full.

Description

`sm_tm_push_model_event` pushes an event onto the transaction event stack. If `event` is 0, the stack is unchanged and a warning is logged. If the stack is full before the event is pushed, the event that is pushed off the stack is returned.

The transaction manager generates requests in response to commands. It calls this function to push each request onto the stack as an event, to commence event processing for the request. This function can also be used by transaction models or by transaction event functions associated with a table view. The events generated by the transaction manager and those generated by the standard transaction models are defined in `tmusubs.h`. For a description of these events, refer to Chapter 9, "Transaction Model Events," in this manual and Chapter 35, "Generating Transaction Manager Events," in *Application Development Guide*.

Example

```
/* The following example taken from the standard
       transaction model for JDB shows the processing for the
       TM_UPDATE request. */

case TM_UPDATE:
       /* Do nothing, except for updates */
    occ_type = sm_bi_compare();
    if (occ_type != BI_UPDATED)
    {
       break;
    }

       if (!reuse_cursor)
    {
       save_cursor_type = 0;
    }
    reuse_cursor = 0;

    sm_tm_push_model_event(TM_UPDATE_EXEC);
    sm_tm_push_model_event(TM_UPDATE_DECLARE);
    sm_tm_push_model_event(TM_GET_SAVE_CURSOR);
    break;
```

See Also `sm_tm_clear_model_events`, `sm_tm_pop_model_event`

sm_tmpnam

Creates a unique file name

```
char *sm_tmpnam(void)
```

Returns

- A pointer to the new file name.

Description

`sm_tmpnam` is an extension of the ANSI function `tmpnam()`; it returns a name that is unique among other file names.

In Windows, the `TMP` environment variable will be used to set the temporary file's directory. If `TMP` is not set but if the environment variable `TEMP` is set, it will be used. Otherwise, the `\` (root directory) on the current disk will be used.

In Linux, if the `TMPDIR` environment variable is set, it will be used as the trmporary file's directory. In UNIX and when `TMPDIR` is not set in Linux, `P_tmpdir` from the `stdio.h` file will be used. If `P_tmpdir` is not defined, `/tmp` will be used.

The default report service unload handler uses this function to generate a temporary file name on the client; it then moves a server-generated metafile to that file name. See "Handling Client Output" on page 9-26 in the Reports Manual for this code.

sm_tp_exec

Executes a middleware-related JPL command in JetNet and TUXEDO applications

```
int sm_tp_exec(char *command-stream);
```

`command-stream`
: A string that contains a middleware-related JPL command. Enclose the command string in quotation marks.

Environment JetNet, TUXEDO

Returns

≥0	An exception severity associated with execution of the command. Refer to the command for information about potential exceptions and their severity levels.
-1	The command failed due to an undefined error.
-2	The application executable is not capable of three-tier processing.

Description `sm_tp_exec` executes the specified middleware API-related JPL command. Table 5-22 shows the JPL commands that you can invoke from this function. For more information about a command, refer to its description in this manual.

Table 5-22 Middleware API Commands

`advertise`	`log`	`unadvertise`
`broadcast`	`notify`	`unload_data`
`client_exit`	`post`	`unsubscribe`
`client_init`	`service_call`	`wait`
`dequeue`	`service_cancel`	`xa_begin`
`enqueue`	`service_forward`	`xa_commit`
`jif_check`	`service_return`	`xa_end`
`jif_read`	`subscribe`	`xa_rollback`

Example

```
int severity;

severity = sm_tp_exec
```

```
("service_call \"WITHDRAWAL\" ({account_id, amount},
   {message, account_bal})");
```

sm_tp_free_arg_buf

Frees memory allocated by argument list generation functions

```
void sm_tp_free_arg_buf(void);
```

Environment JetNet, TUXEDO; C only

Description `sm_tp_free_arg_buf` frees memory that is allocated to build an argument list for any of the `sm_tp_gen_` functions. After the last call to a `sm_tp_gen_` function, call `sm_tp_free_arg_buf` to free this memory.

See Also `sm_tp_gen_insert`, `sm_tp_gen_sel_return`, `sm_tp_gen_sel_where`, `sm_tp_gen_val_link`, `sm_tp_gen_val_return`

sm_tp_gen_insert

Generates an argument list of fields for an INSERT operation

```
char *sm_tp_gen_insert(char *tv, int scope);
```

`tv`
: The name of the first table view to traverse. Supply `NULL` pointer or an empty string to use the screen's root table view.

`scope`
: Specifies which part of the table view tree to use:

  `TM_TV_AND_BELOW`
  : Build the argument list for the fields from `tv` and all table views below it on the tree.

  `TM_SV_ONLY`
  : Build the argument list for the table views on the server view only.

Environment JetNet, TUXEDO; C only

Returns

- A pointer to a string that contains the comma separated list of fields to be inserted for the current `INSERT` operation.
- `NULL` pointer if an error occurs.

Description `sm_tp_gen_insert` returns a list of fields for an `INSERT` operation that can be used as a service call's input argument. The fields are on the current screen and participate in the database `INSERT` operation.

The list is returned in a temporary buffer whose contents remain valid until the next call to a `sm_tp_gen_` function. So, you should use or save the return data immediately. When the last `sm_tp_gen_` function is called, free the memory allocated for the buffer with `sm_tp_free_arg_buf`.

See Also `sm_tp_free_arg_buf`

sm_tp_gen_sel_return

Generates a list of fields for the returned select set of a SELECT or VIEW operation

```
char *sm_tp_gen_sel_return(char *tv, int scope);
```

`tv`

The name of the first table view to traverse. Supply `NULL` pointer or an empty string to use the screen's root table view.

`scope`

Specifies which part of the table view tree to use:

`TM_TV_AND_BELOW`
Build the argument list for the fields from `tv` and all table views below it on the tree.

`TM_SV_ONLY`
Build the argument list for the table views on the server view only.

Environment JetNet, TUXEDO; C only

Returns

- A pointer to a string that contains a list of comma-separated fields for the returned select set of a `SELECT` or `VIEW` operation.
- `NULL` pointer: an error occurred.

Description `sm_tp_gen_sel_return` returns a list of fields that can be used as an output argument for a service request implementing a `SELECT`. or `VIEW` operation. The fields are on the current screen and are used for the returned select set of the database operation.

The list is returned in a temporary buffer whose contents remain valid until the next call to a `sm_tp_gen_` function. So, you should use or save the return data immediately. When the last `sm_tp_gen_` function is called, free the memory allocated for the buffer with `sm_tp_free_arg_buf`.

See Also `sm_tp_free_arg_buf`, `sm_tp_gen_sel_where`

sm_tp_gen_sel_where

Generates a list of fields for the WHERE clause of a SELECT or VIEW operation

```
char *sm_tp_gen_sel_where(char *tv, int scope);
```

`tv`
: The name of the first table view to traverse. Supply `NULL` pointer or an empty string to use the screen's root table view.

`scope`
: Specifies which part of the table view tree to use:

  `TM_TV_AND_BELOW`
  : Build the argument list for the fields from `tv` and all table views below it on the tree.

  `TM_SV_ONLY`
  : Build the argument list for the table views on the server view only.

Environment JetNet, TUXEDO; C only

Returns

- A pointer to a string that contains a list of comma-separated fields for the `WHERE` clause of a `SELECT` or `VIEW` operation.
- `NULL` pointer: an error occurred.

Description `sm_tp_gen_sel_where` returns a list of fields that can be used as an input argument for a service request implementing a `SELECT`. or `VIEW` operation. The fields are on the current screen and are used in the `WHERE` clause.

The list is returned in a temporary buffer whose contents remain valid until the next call to a `sm_tp_gen_` function. So, you should use or save the return data immediately. When the last `sm_tp_gen_` function is called, free the memory allocated for the buffer with `sm_tp_free_arg_buf`.

See Also `sm_tp_free_arg_buf`, `sm_tp_gen_sel_return`

sm_tp_gen_val_link

Generates a list of fields to be validated in a validation link operation

```
char *sm_tp_gen_val_link(char *tv);
```

`tv`
: The name of the first table view to traverse. Supply `NULL` pointer or an empty string to use the screen's root table view.

Environment JetNet, TUXEDO; C only

Returns

- A pointer to a string that contains a list of comma-separated fields to be validated in the current validation link operation.
- `NULL` pointer: an error occurred.

Description `sm_tp_gen_val_link` returns a list of fields that can be used as an input argument for a service request that implements validation link processing. The fields are on the current screen and require validation.

The list is returned in a temporary buffer whose contents remain valid until the next call to a `sm_tp_gen_` function. So, you should use or save the return data immediately. When the last `sm_tp_gen_` function is called, free the memory allocated for the buffer with `sm_tp_free_arg_buf`.

See Also `sm_tp_free_arg_buf`, `sm_tp_gen_val_return`

sm_tp_gen_val_return

Generates a list of fields for the returned select set of a validation link operation

```
char *sm_tp_gen_val_return(char *tv);
```

`tv`
: The name of the first table view to traverse. Supply `NULL` pointer or an empty string to use the screen's root table view.

Environment JetNet, TUXEDO; C only

Returns

- A pointer to a string that contains a list of comma-separated fields for the returned select set of a validation link operation.
- `NULL` pointer: an error occurred.

Description `sm_tp_gen_val_return` returns a list of fields that can be used as an output argument for a service request implementing validation link processing. The fields are on the current screen and are used for the returned select set of the operation.

The list is returned in a temporary buffer whose contents remain valid until the next call to a `sm_tp_gen_` function. So, you should use or save the return data immediately. When the last `sm_tp_gen_` function is called, free the memory allocated for the buffer with `sm_tp_free_arg_buf`.

See Also `sm_tp_free_arg_buf`, `sm_tp_gen_val_link`

sm_tp_get_svc_alias

Returns the service alias for a JetNet or TUXEDO server

```
char *sm_tp_get_svc_alias(char *server);
```

`server`
: A standard or debuggable server (`proserv` or `prodserv`).

Environment JetNet, TUXEDO

Returns

- The service alias currently assigned to the server.

Description `sm_tp_get_svc_alias` returns the service alias assigned to a standard or debuggable server (`proserv` or `prodserv`) in JetNet and TUXEDO applications.

If a server has been assigned a service alias, log entries in development mode for services running on that server will contain the alias name when the services are advertised or named in event handlers.

sm_tp_get_tux_callid

Returns the Tuxedo identifier for a service call

```
int sm_tp_get_tux_callid(char *callid);
```

`callid`

Tuxedo service call identifier, which is set in the `tp_return` property immediately after the service request is made.

Environment Tuxedo

Returns

- Tuxedo identifier of the specified service call.
- `NULL`: `callid` is invalid or there is no active service request.

Description `sm_tp_get_tux_callid` gets the Tuxedo identifier for the specified service call. Use this identifier in order to make direct calls to Tuxedo API functions.

Example

```
char *callid;
   int tux_id;

callid = sm_prop_get_str(sm_prop_id("@app"),\
         PR_TP_THIS_CALL);
   if ((callid != NULL) && (strlen(callid) > 0))
   {

   tux_id = sm_tp_get_tux_callid(callid);
      /*  Make call to TUXEDO API using tux_id... */
      ...
   }
```

sm_trace

Create event trace and dump files

```
int sm_trace(char *commands);
```

`commands`
Commands to execute.

Returns

- 0: no errors found in `commands`.
- `SM_EQUALS`: No equal sign following a `DUMPFILE`; `FRAMES` or `TRACEFILE` command.
- `SM_FORMAT`: a non-alphabetic character was found in a command.
- `SM_MALLOC`: a call to the malloc C function has failed.
- `SM_MISSARGS`: commands is the null pointer; the null string or just whitespace.
- `SM_NOFILE`: unable to open the file in a `DUMPFILE` or `TRACEFILE` command.
- `SM_NOT_LOADED`: `DUMP` command when `DUMPFILE` is not specified.
- `SM_NUMBER`: `DUMP` command when `FRAMES` is zero or the `FRAMES` command value is not one or more digits.
- `SM_QUOTE`: `DUMPFILE` or `TRACEFILE` value starts with a quote character but a matching terminating quote was not found.
- `SM_SYNTAX`: character after the closing quote in a `DUMPFILE` or `TRACEFILE` is not a `NUL` or whitespace.
- `SM_VERB_UNKNOWN`: `command` is not valid.

Description

`sm_trace` can create trace and dump files containing information about Panther events. The events that are reported in these files can be selected. The `commands` parameter is series of command tokens that control operations. Most tokens can be prefixed with `NO` to reverse their effect. The case of tokens is ignored but they will be capitalized in this manual.

`sm_trace` was first released in Panther 5.30.

Trace files are written as events occur. The following commands control trace files:

`TRACEFILE=`	File to write event information to. The file name should be quoted if it contains spaces. Use the null string "" to stop tracing and to close the current `TRACEFILE`.

`TRACE`	Used to control whether event information is to be written to `TRACEFILE`. Default is `TRACE`. You can use `NOTRACE` to disable tracing to reduce the size of the trace file when nothing of interest is happening.

Dump files are written when the `DUMP` command is executed. The number of events to store for reporting is controlled by the `FRAMES=` command. The following commands control dump files:

`DUMPFILE=`	File to write information to. The file name should be quoted if it contains spaces.
`DUMP`	Used to cause information is to be written to `DUMPFILE`.
`FRAMES=`	Specifies how many events are to be stored for writing by the `DUMP` command. 500 is the default.

The following commands control how files are opened:

`OVERWRITE`	Specifies that existing files should be overwritten. It is the default.
`APPEND`	Specifies that existing files should be appended to.

The following commands control which events are to be logged:

`CSTR`	Control String events.
`DBI`	Database events.
`FIELD`	Field entry, exit and validation events.
`FUNCTION`	Installed function call and return events.
`GRID`	Grid and grid row entry, exit and validation events.
`GROUP`	Group entry, exit and validation events.
`JAVA`	Java events.
`JPL`	JPL execution events.

`KEY`	Key being taken off the key queue events.
`LDB`	Local Data Block events.
`RW`	Report Writer events.
`SCREEN`	Screen entry, exit and expose events.
`TM`	Transaction Manager events.
`ALL`	All of the above events. This is the default.
`CORE`	All of the above events except for `DBI`, `RW` and `TM`.
`NONE`	The same as `NOALL`.

The following command causes parameters to be displayed:

`PARMS`	Causes parameters to be displayed for calls to functions, including JPL functions. The default is `NOPARMS`.

To enable tracing to begin at startup, the `SMTRACE` application variable can be included in a `SMVARS` or `SMSETUP` setup file, for example:

```
SMTRACE=NOJAVA FRAMES=100 TRACEFILE="c:\temp\forex.trc"
```

`SMINITJPL` in a setup file can also be used to call `sm_trace`, for example:

```
SMINITJPL=call sm_trace('PARMS TRACEFILE="c:\temp\forex.trc"')
```

When debugging a program, `sm_trace` can be called from the debugger to produce a dump file. For example, one can call:

```
sm_trace("DUMPFILE='dump1.dmp' DUMP")
```

to create a dump file named "dump1.dmp" with the most recent trace information.

sm_translatecoords

Translates screen coordinates to display coordinates

```
int sm_translatecoords(int column, int line, int *column_ptr,
    int *line_ptr);
```

`column, line`
: Zero-based coordinates relative to the current screen, where `0,0` specifies the screen's upper-left corner.

`column_ptr, line_ptr`
: On return, contain the pixel coordinates relative to the drawing area.

Environment Motif, Windows

Returns

0	Success.
-1	`line` or `column` is out of range

Description `sm_translatecoords` translates the Panther `line` and `column` relative to a screen, into pixel line and column relative to the upper left hand corner of the drawing area. `line` and `column` are zero based. This function in conjunction with `sm_drawingarea` is useful when placing objects such as bitmapped graphics or custom widgets on a Panther screen.

Example

```
#include <smdefs.h>
#include <smwin.h>
#include "drawbmp.h"
    /*
     * The following program shows how to display a bitmap file
     * on current Panther screen in Windows. The routine uses
     * several functions from sample code in Programming Windows
     * 3.1, pp. 610-616 by Charles Petzold (Microsoft Press,
     * 1992). All other functions are either standard C,
     * Panther API, or Windows API calls
     */

int prlfx_display_bmp_file(char *file_name, int ln, int col)
    {
      static BYTE huge *lpDib;
      HWND hwnd;
      HDC hdc;
      BYTE huge *lpDibBits;
      short cxDib, cyDib, pix_ln, pix_col;
  if (lpDib != NULL) {
        GlobalFreePtr(lpDib);
        lpDib = NULL;
      }
      lpDib = ReadDib(file_name); /* Petzold, pp. 613-614 */

  if (lpDib == NULL) {
          sm_message_box("Could not open DIB file", "ERROR",
                SM_MB_OK | SM_MB_ICONSTOP, 0);
      return RET_FATAL;
      }
      hwnd = sm_mw_drawingarea();
      hdc = GetDC(hwnd);

  if (hdc != NULL) {
        lpDibBits = GetDibBitsAddr(lpDib);/* Petzold,p. 612 */
        cxDib = GetDibWidth(lpDib);      /* Petzold, p. 612 */
        cyDib = GetDibHeight(lpDib);     /* Petzold, p. 612 */

    if (sm_translatecoords(col, ln, &pix_col, &pix_ln) < 0) {
          char buf[100];
          sprintf(buf,
            "prlfx_display_bmp_file: invalid line/column: %d/%d",
             ln, col);
          sm_message_box(buf, "ERROR",
                SM_MB_OK | SM_MB_ICONSTOP, 0);
          return RET_FATAL;
        }

    SetStretchBltMode(hdc, COLORONCOLOR);
        SetDIBitsToDevice(
             hdc, pix_col, pix_ln, cxDib, cyDib, 0, 0, 0, cyDib,
             (LPSTR) lpDibBits, (LPBITMAPINFO) lpDib,
             DIB_RGB_COLORS);
        }
      else {
        sm_message_box("Could not get handle to drawing area",
             "ERROR", SM_MB_OK | SM_MB_ICONSTOP, 0);
      }
      ReleaseDC(hwnd, hdc);
      return RET_SUCCESS;
    }
```

sm_tst_all_mdts

Finds the first modified occurrence on the current screen

```
int sm_tst_all_mdts(int *occurrence);
```

`occurrence`
: On output, the address of a variable that contains the number of the first modified occurrence.

Environment C only

Returns

≥1	The number of the first field on the current screen that contains a modified occurrence. In this case, the number of the first occurrence that has its `mdt` property set to `PV_YES` is returned in the variable addressed by `occurrence`.
0	No occurrence on the current screen has its `mdt` property set to `PV_YES`.

Description `sm_tst_all_mdts` tests the `mdt` property of all onscreen and offscreen occurrences of all fields on the current screen. If it finds an occurrence with its `mdt` property set to `PV_YES`, the function returns with the base field and occurrence number. Use this function to ascertain whether any occurrence has been modified on the screen, either from the keyboard or by the application program, since the screen was displayed or since the occurrence's `mdt` property was last cleared.

Note: `sm_tst_all_mdts` does not test for insertion or deletion of occurrences; it only tests the `mdt` property of existing occurrences.

Example

```
#include <smdefs.h>

   /* Clear mdt property for all fields on screen;
    * then write data to last field, and check
    * that its mdt property is the first one set.
    */

   int occurrence;
   int numflds;

   sm_cl_all_mdts();
   numflds = sm_prop_get_int(PR_CURSCREEN, PR_NUMFLDS);
   sm_putfield(numflds, "Hello");
   if (sm_tst_all_mdts(&occurrence) !=
         sm_prop_get_int(PR_CURSCREEN, PR_NUMFLDS))
```

```
ferr_reset(0,
"Something is rotten in the state of Denmark.");
```

sm_udtime

Formats a user-supplied date and time

```
char *sm_udtime(struct tm *dt_tm_data, char *format);
```

`dt_tm_data`
A pointer to the date and time data to format. `dt_tm_data` is a `tm` structure, defined in the standard C header file `time.h`.

`format`
Specifies the format to use with an expression that starts with `y` or `n`, followed by any combination of date/time tokens and literal text. `y` indicates a 12-hour clock; `n` or any other character indicates a 24-hour clock. This character is required even if the format does not include time tokens. Refer to Table 5-19 on page 5-467 for a list of the date/time tokens that you use to build a format expression.

Environment C only

Returns

- A pointer to a string that contains the user date/time in the specified format.
- Empty if `format` is invalid.

Description `sm_udtime` formats the date and time data in `dt_tm_data` according to the specified `format`.

This function uses a static buffer that it shares with other date and time formatting functions. The buffer is 256 bytes long. Panther does not check for overflow. Consequently, you should process the returned string or copy it to a local variable before making additional function calls.

Example

```
/* Put the date 135 days from now into the field "maturity"  */
   #include <smdefs.h>
   time_t tim;
   struct tm *matdate;
   char *ptr;

/* calculate local time in seconds */
   tim = time((time_t *)0) + 135L * 24 * 60 * 60;
   matdate = localtime(&tim);
   ptr = sm_udtime(matdate, " %0f");
   sm_n_putfield("maturity", ptr);
```

See Also `sm_sdtime`

sm_ungetkey

Pushes a translated key onto the input queue

```
#include <smkeys.h>
int sm_ungetkey(int key);
```

`key`
: The key to push onto the input stack.

Returns

- The value of `key`.
- −1: Insufficient memory.

Description

`sm_ungetkey` saves the translated key given by `key` so it can be retrieved by the next call to `sm_getkey`. Multiple calls are allowed. The key values are pushed onto a stack in last-in/first-out order.

When `sm_getkey` reads a key from the keyboard, it flushes the display first so the user sees a fully updated display before typing on. This is not the case for keys pushed back by `sm_ungetkey`.

Example

```
#include <smkeys.h>

    /* Force tab to next field */
    sm_ungetkey(TAB);
```

See Also

`sm_getkey`

sm_unload_screen

Unloads a screen from memory

```
void sm_unload_screen(char *screen_name);
```

`screen_name`
The name of the screen.

Description `sm_unload_screen` unloads a screen previously loaded into memory by `sm_load_screen` or `sm_svscreen`. This function simply makes a call to `sm_unsvscreen` for one screen. Unlike `sm_unsvscreen`, it can be called from JPL.

See Also `sm_load_screen`, `sm_svscreen`, `sm_unsvscreen`

sm_unsvscreen

Removes screens from the save list

```
void sm_unsvscreen(char **screen_list, int count);
```

`screen_list`
: The screens to remove from the save list.

`count`
: The number of screens to remove from the save list.

Environment C only

Description `sm_unsvscreen` removes screens from the list of screens that are saved in memory and frees the memory associated with them. You can call this function to remove screens from this list anywhere in your code, whether or not the screen is open. Note that if a screen is open, Panther frees its memory only when it closes.

See Also `sm_issv`, `sm_svscreen`

sm_upd_select

Updates the contents of an option menu or combo box

```
int sm_upd_select(int fldno);
int sm_n_upd_select(char *fldname);
```

`fldname, fldno`
The name or field number of the option menu or combo box to update.

Environment Motif, Windows

Returns

0	Success.
-1	Invalid widget type.
-2	Widget’s list contains constant data.

Description `sm_upd_select` updates the contents of an option menu or combo box with data from another screen. The widget must be defined to accept data from an external screen; otherwise, the function returns an error.

An option menu or combo box that gets its data from a screen can be initialized either on screen entry or each time the widget list displays, depending whether its `initialization` property is set to `PV_FILL_AT_POPUP` or `PV_FILL_AT_INIT`. Use `sm_upd_select` to force updates only if `initialization` is set to `PV_FILL_AT_INIT`.

Note: If fields on the external screen have initial data, LDB write-through is disabled for those fields.

sm_*validate

Forces validation for the specified object

```
int sm_validate(int obj_id);
int sm_n_validate(char *widget_name);
```

`obj_id`
: Object ID of the widget or screen to be validated, obtained from `sm_prop_id`

`widget_name`
: The name of the widget to validate. For fields, an occurrence number or element number can be included. For example, `sm_n_validate "field[[3]]")` causes the third element of the named field to be validated.

Returns

0	Success.
-1	One of the fields failed validation or one of the validation functions did not return 0.
-2	The specified object is invalid or not on the current screen.

Description

`sm_validate` causes the specified widget or screen to be validated. The specified widget must be on the current screen.

The widget must be one of the following types:

- A field that is not a card. When `sm_n_validate` is called, *`widget_name`* can specify an occurrence or an element to validate. If not specified, the first element is validated.
- A tab card. Each field, grid and tab card in the card is validated.
- A tab deck. Each tab card in the deck is validated.
- A screen. This is the same as calling `sm_s_val` except that the flag `K_USER` rather than `K_SVAL` is passed to the validation functions.
- A grid. The grid validation function is called (unless the grid is empty). The fields in the grid are not validated.

Validation stops when a field fails validation or when a validation function returns a non-zero value. See `sm_fval` for more about field validation and `sm_s_val` for more about screen validation.

Example

```
#include <smdefs.h>

/* Call a grid validation function and return if it fails.
   */

int
   validate (fieldnum, data, occurrence, bits)
   int fieldnum, occurrence, bits;
   char *data;
   {
      if (sm_n_validate("grid1")

   {
         /* Stop processing if grid validation function fails */
         return 1;
      }
      ...
   }
```

See Also `sm_fval`, `sm_n_gval`, `sm_s_val`

sm_*vinit

Initializes the video translation table

```
int sm_vinit(char *video_address);
int sm_n_vinit(char *video_file);
```

`video_address`
: The address of a memory-resident video file. Create this file with `vid2bin` and `bin2c` utilities, then compile it into the application.

`video_file`
: The name of a video file, set in the `SMVIDEO` variable that is specified in the setup file or in the environment.

Returns

- 0 Success.
- ● Non-zero value: failure.

Description

`sm_vinit` and `sm_n_vinit` initialize the video translation table. Panther uses one of these functions during program initialization, depending on whether the video file is memory-resident or resides on disk. These functions can also be called directly by an application program.

If `sm_vinit` fails, you can generate error messages through `sm_inimsg`. This function creates formatted output that you can display through other library functions like `sm_fqui_msg`.

Example

```
/* Install a memory-resident video file */

    extern char special_vid[];

    sm_vinit (special_vid);
```

sm_wcount

Obtains the number of currently open windows

```
int sm_wcount(void);
```

Returns

≥1	The number of windows open.
0	The base window is the only open screen.
-1	There is no current screen.

Description

`sm_wcount` returns the number of windows currently open. The number is equivalent to the number of windows in the window stack, excluding the base window.

Use this function with `sm_wselect` to activate another window from the window stack. For example, the following statement selects the screen beneath the current window:

```
sm_wselect(sm_wcount()-1);
```

See Also

`sm_wselect`

sm_wdeselect

Restores the previously active window

```
int sm_wdeselect(void);
```

Returns	0	Success.
	-1	No window to restore.

Description

`sm_wdeselect` restores a window to its original position in the window stack after it has been moved to the top by a call to `sm_wselect`. Successive calls to `sm_wdeselect` recursively restore windows selected by `sm_wselect`.

See Also

`sm_wcount`, `sm_wselect`

sm_web_get_cookie

Returns the value of the specified cookie

```
#include <smuweb.h>
char *sm_web_get_cookie(char *cookie_name);
```

`cookie_name`
The cookie whose value you want to retrieve.

Environment
Web

Returns
- The value of the specified cookie.
- A null pointer if `cookie_name` does not exist.

Description
`sm_web_get_cookie` returns the value of the specified cookie. The cookie must already exist in the user's browser program.

If the same cookie name is present more than once, only the first one is returned by this function. The entire cookie string is available as `@cgi_http_cookie`.

Example
```
// Get the browser's cookie values for user and visit_num
// and insert those values into the user and visit_num
// fields on the screen. Then, the visit_num cookie is reset
// to its new value.

proc entry

user = sm_web_get_cookie("user")
   visit_num = sm_web_get_cookie("visit_num")
   visit_num = visit_num + 1
   call sm_web_set_cookie("visit_num=:visit_num;\
      expires=Monday, 03-Jan-2030 00::00::00 GMT; \
      domain=.Panther.com; path=/samples")
```

See Also
`sm_web_set_cookie`

sm_web_invoke_url

Invokes a URL on the Web

```
#include <smuweb.h>
void sm_web_invoke_url(char *url);
```

`url`
: The URL to call and display in the browser.

Environment Web

Description `sm_web_invoke_url` displays the specified Web resource in the Web browser. When this function is called, Panther suspends the HTML generation for the screen and outputs the HTML to go directly to the specified resource.

Example

```
// Go to the Panther home page.
   proc go_home

   call sm_web_invoke_url("http\://www.Panther.com")
   return
```

sm_web_log_error

Write Web application errors to a log file

```
void sm_web_log_error(char *message);
```

`message`
: The text of the error message.

Environment Web

Description `sm_web_log_error` writes Web application errors to the file specified in the `ErrorFile` variable in the Web initialization file. This error text is *not* displayed to the user; the `sm_femsg` or `sm_message_box` functions display error messages to application users.

Example

```
// The following JPL error handler displays a message to
// the user and then writes a message to the error log file.

proc error_def
   msg emsg "Error:  File not found"
   call sm_web_log_error("Unable to find file.")
   return
```

sm_web_save_global

Creates a context global variable

```
#include <smuweb.h>
int sm_web_save_global(char *variable_name);
```

`variable_name`
Name of the JPL global variable to be designated as a context global.

Environment Web

Returns

0	Success
-1	The supplied variable is not an existing JPL global.

Description `sm_web_save_global` designates a JPL global variable as a context global. Each context global is private to a single user of a Web application server and is automatically cached until it reverts to a transient global or the application exits. Before calling this function, create the global variable with the `global` command.

A context global can save user-specific information such as ID, preferences, or start time. The value of the variable is cached before Panther generates the HTML for the screen; the value is restored from the cache file when the screen is submitted back to the server.

If the `global` command executes a second time, it overwrites the global's previous value. If you execute the `global` command in the unnamed JPL procedure or during screen entry, also test whether the screen is being opened for a `GET` event, because the screen is then reopened on a `POST` event. You can test this by using the `K_WEBPOST` flag or the CGI variable `@cgi_request_method`.

For more information about using JPL global variables in a Web application, refer to Chapter 7, "JPL Globals in Web Applications," in *Web Development Guide*.

Example

```
// Call on application startup.
   proc setup()

// Create global variables if they do not already exist.
   if (sm_web_save_global("pref_lang") < 0)
   {
      global pref_lang(15), pref_textonly(1), pref_maxrows(1)
      call sm_web_save_global("pref_lang")
```

```
    call sm_web_save_global("pref_textonly")
    call sm_web_save_global("pref_maxrows")

// Initialize the preferences to the default values.
    pref_lang = def_lang
    pref_textonly = def_textonly
    pref_maxrows = def_maxrows
}
```

See Also `sm_web_unsave_global`, `sm_web_unsave_all_globals`

sm_web_set_cookie

Sets HTML cookies on a client

```
#include <smuweb.h>
void sm_web_set_cookie(char *cookie_string);
```

`cookie_string`

Specifies the cookie's name and properties in the following format:

`cookie-name``=`*`value`*` [ ; expires=`*`date`*` ] [ ; path=`*`path-string`*` ] [ ; domain=`*`domain-name`*` ] [ ; secure ]`

`name``=`*`value`*

Specifies a unique character string to identify the cookie and assigns the cookie a value. For example:

```
Visits=1
```

`expires=`*`date`*

Specifies an expiration date for the cookie in Greenwich mean time (GMT), where *`date`* has this format:

```
Wdy, DD-Mon-YYYY HH:mm:ss GMT
```

After this date, the cookie is no longer stored or given out. This parameter must be specified in order to store the cookie value on the browser for multiple sessions; otherwise, the cookie value expires when the browser session ends.

For example, the following cookie expires on December 31, 1999 at 11:45 PM.

```
Friday, 31-Dec-1999 23:45:05 GMT
```

`path=`*`path`*

Specifies the path of the URL to use in matching the cookie values. If you specify the path value as `/`, the cookie is sent for every request to your HTTP server. If you specify the path value as `/`*`dir`*, the cookie is sent only if the URL path contains `/`*`dir`*. For example, `path` might be set as follows:

```
path=/vid
```

Given this path, the cookie is included when the following URL is sent to the HTTP server:

`http:/www.Panther.com/cgi-bin/vid/main.scr`

`domain=`*`domain-name`*
: Specifies the domain of the URL to use when matching the cookie values. If there is a tail match, the path value is checked to determine whether to send the cookie value.

  The value specified must have at least two periods in it. For example:

  `domain=.Panther.com`

`secure`
: Specifies to send the cookie only if the HTTP request is transmitted to a secure server.

Environment Web

Description `sm_web_set_cookie` adds the specified string to a list of cookies. When HTML is generated for the screen, each cookie is sent as a `Set-cookie:` header. After HTML generation, the list of cookies is removed.

Cookies are pieces of information that can be stored on the browser side of the connection and later retrieved. In order to use them, the browser must accept cookie specifications.

Any cookie specified with this function is included in the HTML header for the screen. If accepted by the browser, the cookie is stored on the browser. Afterward, if the browser asks for a resource from the HTTP server that originally sent the cookie, the cookie value is sent along with the resource request.

Example

```
// Get the browser's cookie values for user and visit_num
   // and insert those values into the user and visit_num
   // fields on the screen, then reset the visit_num cookie
   // to its new value.

proc entry

user = sm_web_get_cookie("user")
   visit_num = sm_web_get_cookie("visit_num")
   visit_num = visit_num + 1
   call sm_web_set_cookie("visit_num=:visit_num;\
      expires=Monday, 03-Jan-2030 00::00::00 GMT; \
      domain=.Panther.com; path=/samples")
```

See Also `sm_web_get_cookie`

sm_web_set_onevent

Install a C Web event hook function

```
int sm_web_set_onevent(char *handler);
```

`handler`
: The name of the Web event handler. If `handler` is the null pointer or the null string, the Web event handler is uninstalled. Otherwise, it must be the name of a C function installed in the prototyped function list.

Returns

0	The Web event hook function was installed or uninstalled.
-1	`handler` is not in the list of installed prototyped C functions.

Description

`handler` is called after processing has completed for each request. The prototype of this function is:

```
void handler(char *command, char *path_info, char *arg,
             int elapsed_time);
```

This function is passed the following parameters:

`command`
: A string containing the web command that was processed. It will be either `"GET"` or `"POST"`.

`path_info`
: This is the value of `PATH_INFO` in the CGI request.

`arg`
: The query string for GET and the form name for POST.

`elapsed_time`
: This is the elapsed time for the `Web` command in milliseconds.

sm_web_unsave_all_globals

Redesignates all context global variables as transient globals

```
#include <smuweb.h>
int sm_web_unsave_all_globals(void);
```

Environment Web

Returns

0	Success.
-1	No context global variables exist.

Description `sm_web_unsave_all_globals` redesignates all context global variables as transient globals, which are destroyed when Panther completes the screen's processing and generates HTML for the screen. Context global variables are created through `sm_web_save_global`. For more information about using JPL global variables in a Web application, refer to Chapter 7, "JPL Globals in Web Applications," in *Web Development Guide*.

Example

```
// Remove all context globals from application
proc undo_var()

call sm_web_unsave_all_globals()
```

See Also `sm_web_save_global`, `sm_web_unsave_global`

sm_web_unsave_global

Redesignates a context global variable as a transient global

```
#include <smuweb.h>
int sm_web_unsave_global(char *variable_name);
```

`variable_name`
  The name of the JPL global variable to remove from the save list.

Environment Web

Returns

0	Success.
-1	The specified variable is not on the save list, or the save list is empty.

Description `sm_web_unsave_global` redesignates the specified context global variable as a transient global, which is destroyed when Panther completes the screen's processing and generates HTML for the screen. Context global variables are created through `sm_web_save_global`. For more information about using JPL global variables in a Web application, refer to Chapter 7, "JPL Globals in Web Applications," in *Web Development Guide*.

Example

```
// Remove context globals from application
    proc undo_var()

    call sm_web_unsave_global("pref_lang")
    call sm_web_unsave_global("pref_textonly")
    call sm_web_unsave_global("pref_maxrows")
```

See Also `sm_web_save_global`, `sm_web_unsave_all_globals`

sm_*widget

Gets a handle to a widget

```
#include <smmwuser.h>
HWND sm_mw_widget(int widgetnumber);
HWND sm_mwn_widget(char *widgetname);
HWND sm_mwe_widget(char *widgetname, int element);

#include <smxmuser.h>
Widget sm_xm_widget(int widgetnumber);
Widget sm_xmn_widget(char *widgetname);
Widget sm_xme_widget(char *widgetname, int element);
```

`widgetname, widgetnumber`
: Specifies the widget whose handle you want to get. (See the Returns section for unavailable widget types.)

`element`
: If the widget is an array, specifies element whose handle you want to get.

Environment C only

Returns

- Success: For Windows, an HWND handle; for Motif, a Widget ID.
- -1 The specified widget is a graph, grid, grid member, box, line or scale widget.
- Null pointer: the widget does not exist.

Description `sm_widget` gets a handle to the specified widget or widget element—in the case of Windows applications, a HWND handle; under Motif, a Widget ID. You can pass this handle to Windows and Motif functions when you want the window manager to act directly on a Panther widget.

For more information about corresponding Motif and Panther widget types, refer to “Widget Hierarchy” on page 4-12 in *Configuration Guide*.

Note: For scale widgets, list box widgets, and multiline text widgets in Motif applications, `sm_xm_widget` and its variants return the widget ID of the scroll bar. Use `XtParent` to obtain the ID of the scale, list box or multiline text widget. For list boxes in Windows applications, `sm_mw_widget` and its variants return a handle to the list box itself. SDK function calls such as GetScrollPos use the list box’s handle and a flag that identifies the desired scroll bar.

sm_win_shrink

Trims the current screen

```
int sm_win_shrink(void);
```

Environment
: Motif, Windows

Returns
: `PI_ERR_NONE`: Success

Description
: `sm_win_shrink` trims all space on a screen to the right of the rightmost widget and below the bottom widget. It does not change the number of Panther lines and columns. It is primarily useful after repositioning fields. Call `sm_adjust_area` to restore a screen to its original size.

sm_*window

Opens a window at a given position

```
int sm_d_window(char *address, int start_line, int start_column);
int sm_l_window(int lib_desc, char *name, int start_line,
    int start_column);
int sm_r_window(char *name, int start_line, int start_column);
```

`address`
: The address of the screen in memory

`lib_desc`
: Specifies the library in which the window is stored, where `lib_desc` is an integer library descriptor returned by `sm_l_open`. You must call `sm_l_open` before you read any screens from a library.

`name`
: The name of the window.

`start_line, start_column`
: Specifies the window's top left corner, where `start_line` and `start_column` are zero-based offsets from the physical display's top left corner. Thus, setting `start_line` to 1 starts the window at the screen's second line. If the window does not fit on the display at the specified location, Panther adjusts it as needed.

  A negative value for `start_line` specifies to clear the current screen before displaying the window. The screen's contents are discarded and cannot be restored.

Environment `sm_d_window` is C only

Returns

0	Success.
-1	Screen file's format is incorrect.
-2	Screen cannot be found.
-3	Insufficient memory available to display the screen; the current screen remains displayed.
-4	Read error occurred after the current screen was cleared and `start_line` is -1. Consequently, Panther cannot restore the screen.
-5	System ran out of memory after the current screen was cleared and `start_line` is -1. Consequently, Panther cannot restore the screen.

-6 Library is corrupted.
-7 The window is larger than the physical display and there are fields that overhang the display.

Description

Use `sm_d_window`, `sm_l_window`, or `sm_r_window` to display a screen as a stacked window at the specified line and column.:

- On GUI platforms such as Windows, the window is positioned relative to the GUI display. For example, in Windows, the screen is positioned in the middle of the display; on Motif, it is positioned relative to the base window's status line.
- In character mode, the window is positioned relative to the cursor position on the invoking screen, offset by one line to avoid hiding the line's current display.

The area of the display that surrounds the window remains visible. However, only the opened window is active, and only its fields are accessible to input and library functions. To change the active window, use `sm_wselect`.

To display a form use `sm_r_form` or one of its variants. Use `sm_close_window` to close the window.

Search Path

When you use `sm_r_window`, Panther looks for the named screen in the following places in this order:

1. The application's memory-resident list; if found, `sm_d_window` is called to display the screen.
2. All open libraries; if found, `sm_l_window` is called to display the screen.

If the search fails and the supplied file name has no extension, Panther appends the `SMFEXTENSION`-specified extension to the file name and repeats the search. If all searches fail, `sm_r_window` displays an error message and returns.

Memory-resident Screens

You can save processing time by using `sm_d_window` to display screens that are memory-resident. Use `bin2c` to convert screens from disk files to program data structures that you can compile into your application.

A memory-resident screen never changes at runtime and therefore can be made sharable on systems let you share read-only data. `sm_r_window` can also display memory-resident screens if they are properly installed with `sm_formlist`. Memory-resident screens are especially useful in applications with a limited number of screens, or in environments with a slow disk.

Screens Stored in Libraries You can also save processing time with `sm_l_window` to display screens from a library. A library is a single file that stores screens, JPL modules, and menus. You can assemble a library from individual screen files with `formlib`. Libraries let you distribute a large number of screens with an application, and can improve efficiency by reducing the number of search paths.

Example

```
/* Bring up a window from a library. */
    int ld;

   if ((ld = sm_l_open("myforms")) < 0)
      sm_cancel();
   ...
   sm_l_window(ld, "popup", 5, 22);
   ...
   sm_l_close(ld);
```

See Also `sm_*at_cur`, `sm_close_window`, `sm_*form`, `sm_jwindow`

sm_winsize

Lets users interactively move and resize a window

```
int sm_winsize(void);
```

Returns

0	Success.
-1	Failure.

Description

`sm_winsize` invokes the viewport status line and lets the user move, resize and change the offset of the current screen and any sibling windows. XMIT restores the previous status line. To resize the viewport programmatically, set the applicable viewport properties for the screen.

sm_wrotate

Rotates the display of sibling windows

```
int sm_wrotate(int step);
```

`step`

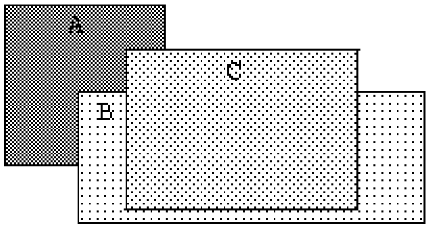
A positive or negative integer that specifies the number of times to rotate the windows. A positive value makes the topmost sibling window the last sibling window for each instance of `step`. A negative value makes the last sibling window first window. A value of 0 specifies to perform no rotations.

Returns

≥1	The number of sibling windows, less one, on top of the window stack.
0	Failure: There are no sibling windows.

Description

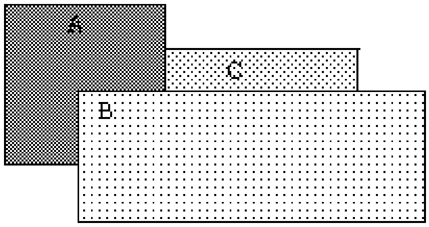
`sm_wrotate` rotates the sequence of sibling windows according to the value of `step`. For example, given the following sequence of sibling windows A, B, and C:



this following function call:

```
sib_windows = sm_wrotate(1);
```

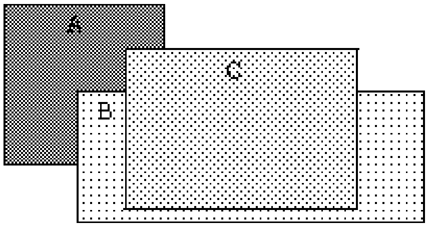
rotates the top sibling window C to the bottom of the sibling stack and leaves screen B on top.



Conversely, this function call supplies a value of `-1`:

```
sib_windows = sm_wrotate(-1);
```

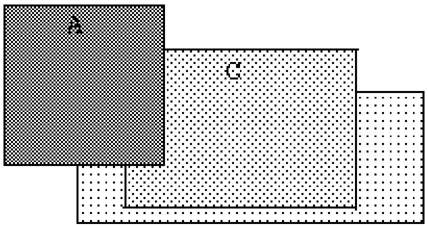
This rotates the bottom sibling window C to the top:



`sm_wrotate` can take any value, positive or negative, as the `step` value. If the value of `step` is greater than one, Panther rotates the windows that many times. For example, given the previous window order, this call tells Panther to perform two rotations:

```
sib_windows = sm_wrotate(2);
```

This moves the top two windows to the back—first C, then B. This leaves window A as the topmost window:



See Also `sm_setsibling`, `sm_wselect`

sm_*wselect

Activates a window

```
int sm_wselect(int window_number);
int sm_n_wselect(char *window_name);
```

`window_number`
: Specifies the window to activate, where `window_number` is its zero-based offset in the window stack. Windows are numbered sequentially from the bottom of the stack, where the bottom-most screen, or base window, is 0. Calling `sm_wselect` changes the number of the specified window and all windows previously above it.

`window_name`
: The window's screen name.

Returns

≥0	The number of the window that was made active—either the value of `window_number`, or the maximum if `window_number` is out of range.
-1	Failure: The window was not found or the window was not open.

Description

`sm_wselect` lets you change the active window in a multi-window display. This function is typically used in routines that update information in windows that might be inactive.

Only one window—the one at the top of the window stack—can be active at a time, and thereby accessible to library functions and user input. These functions activate a window by bringing it to the top of the window stack and restores the cursor to its last position in it. If the window is hidden by an overlying window, Panther brings it to the forefront of the display.

You can specify a window by its offset into the window stack with `sm_wselect`, or by its screen name with `sm_n_wselect`. `sm_wselect` involves more work inasmuch as you must keep track of the inactive window's position on the stack. However, `sm_wselect` can find windows displayed with `sm_d_window` or related functions, which do not record the screen name.

In character mode, `sm_wselect` selects sibling windows as a group. If any one of a set of sibling windows is activated by this function, then all of the siblings are brought to the top of the window stack. The selected window becomes the active window at the top of this set. Otherwise, the sequence within the set of siblings remains the same.

`sm_wselect` and `sm_n_wselect` can be used in the following ways:

- Select a hidden screen, update it with `sm_putfield`, then deselect it with `sm_wdeselect`. Panther updates the visible portion of the hidden screen with the new data. Because of delayed write, Panther updates the screen only when keyboard input is sought.
- Select a hidden screen and open the keyboard. In this case, the selected screen becomes visible, and can hide part or all of the previously active screen. This lets you implement multi-page forms, or switch among several tiled windows. You can let the user select among windows by defining them as siblings.

See Also `sm_wcount`, `sm_wdeselect`

sm_*ww_length

Gets the number of characters in a wordwrapped multiline text widget

```
int sm_ww_length(int field_number);
int sm_n_ww_length(char *field_name);
```

`field_number, field_name`
: Specifies the field whose length is required. Word wrapped text is allowed only in multiline text widgets whose `word_wrap` property is set to `PV_YES`.

Returns

≥0	The number of bytes in the specified field, excluding the null terminator.
-1	Failure.

Description

`sm_ww_length` returns the number of bytes in a word wrap field—that is, a multiline text widget whose Word Wrap property is set to Yes. You can call this function to get the offset into the end of word wrap field data, then use that offset to append data to the field with `sm_ww_write`. You can also use it to determine how large a buffer you need to allocate for reading word wrap field data with `sm_ww_read`.

Example

```
/* this JPL procedure reads text from a filestream and
     * reads each line into a word wrapped field. It uses
     * sm_ww_write to reformat the file text so that it
     * wraps within the field.
     */

proc wrapFileTextToMulti
    {
      vars str, last_char, wwErr, err, fileStream

      call sm_fio_error_set(0)

  /* get file stream sent from previous dialog */
      receive DATA fileStream
      err = 0
      while (err == 0)
      {
        str = sm_fio_gets(fileStream, 255)
        /* check for error condition like EOF */
        if (str != "")
        {
          last_char = sm_n_ww_length("comments")
       /* if writing to empty array */
          if (last_char = 0)
          {
            wwErr = sm_n_ww_write("comments", str, last_char)
          }

       /* otherwise add space after last char before write*/
          else
          {
            wwErr = sm_n_ww_write("comments", " ", last_char)
            wwErr = sm_n_ww_write("comments", str, last_char+1)
          }
        }
        else
        {
          err = sm_fio_error()
        }
      }
      call sm_fio_close(fileStream)
      return
    }
```

See Also `sm_ww_read`, `sm_ww_write`

sm_*ww_read

Copies the contents of a wordwrapped text widget into a text buffer

```
int sm_ww_read(int field_number, char *buffer, int nbytes,
    int offset);
int sm_n_ww_read(char *field_name, char *buffer, int nbytes,
    int offset);
```

`field_name, field_number`
: Specifies the field whose contents you want to read. Word wrapped text is allowed only in multiline text widgets whose `word_wrap` property is set to `PV_YES`.

`buffer`
: A pointer to the buffer into which the field's contents are to be read. To determine the size required by this buffer, call `sm_ww_length` to get the length of the word wrapped text and add 1.

`nbytes`
: The size of `buffer` in bytes.

`offset`
: The offset into the word wrap field at which to start reading. Supply a value of 0 to start reading from the beginning of the field.

Environment C only

Returns

≥0	The number of bytes read into buffer, excluding the null terminator.
-1	Failure.

Description `sm_ww_read` copies word wrapped text from a multiline text widget into `buffer`, starting at offset. A null terminator is supplied in buffer after the copied data. Use `sm_ww_length` to determine the size required for `buffer` and add 1.

See Also `sm_ww_length`, `sm_ww_write`

sm_*ww_write

Writes text into a word wrapped text widget

```
int sm_ww_write(int field_number, char *buffer, int offset);
int sm_n_ww_write(char *field_name, char *buffer, int offset);
```

`field_name, field_number`
: Specifies the field to receive the contents of `buffer`. The field must be a multiline text widget whose `word_wrap` property is set to `PV_YES`.

`buffer`
: A pointer to a null-terminated buffer that contains the text to write.

`offset`
: The offset into the word wrap field at which to start writing. Supply a value of 0 to start writing at the beginning of the field. If supplied value is greater than the field's total length, Panther recalculates the value of `offset` to the field's length + 1; the contents of `buffer` are thereby appended to the end of the field.

Returns

≥0	The number of bytes written to the field.
-1	Failure.

Description

`sm_ww_write` copies `buffer` text into the specified word wrap field—that is, a multiline text widget whose Word Wrap property is set to Yes. `sm_ww_write` wraps at the end of words and leaves a space at the end of each line. If a word is equal to or longer than the length of the field, `sm_ww_write` breaks the word one character before the end of the field, appends a space, and wraps the rest of the word on the next line.

Overflow and Underflow

If you try to copy data that is too large for the field to hold, `sm_ww_write` truncates the excess text. If the field's original contents exceeds the amount of text in `buffer`, the leftover text remains in the field. To avoid this, first clear the field with `sm_clear_array` or one of its variants before calling `sm_ww_write`.

Example

```
/* this procedure reads text from a filestream and
 * reads each line into a word wrapped field. It uses
 * sm_ww_write to reformat the file text so that it
 * wraps within the field.
 */
```

```
proc wrapFileTextToMulti
{
  vars str, last_char, wwErr, err, fileStream

  call sm_fio_error_set(0)

  /* get file stream sent from previous dialog */
  receive DATA fileStream
  err = 0

  while (err == 0)
  {
    str = sm_fio_gets(fileStream, 255)
    /* check for error condition like EOF */
    if (str != "")
    {
      last_char = sm_n_ww_length("comments")

      /* if writing to empty array */
      if (last_char = 0)
      {
        wwErr = sm_n_ww_write("comments", str, last_char)
      }

      /* otherwise add space after last char before write*/
      else
      {
        wwErr = sm_n_ww_write("comments", " ", last_char)
        wwErr = sm_n_ww_write("comments", str, last_char + 1)
      }
    }
    else
    {
      err = sm_fio_error()
    }
  }
  call sm_fio_close(fileStream)
  return
}
```

See Also `sm_clear_array`, `sm_ww_length`, `sm_ww_read`

sm_xlate_table

Installs or deinstalls an 8-bit character translation table

```
char *sm_xlate_table(int which, char *new);
```

`which`
: Determines whether the table is for keyboard input or screen output through arguments of `XLATE_INPUT` or `XLATE_OUTPUT`.

`new`
: The name of the new translation table, where `new` can hold at least 256 bytes. Be sure to allocate permanent memory to hold the table data.

Environment C only

Returns

- Pointer to the previous table.
- `NULL`: No previous table found.

Description `sm_xlate_table` installs the translation table pointed to by `new`. To deinstall and deactivate translation, supply a value of `NULL` for `new`.

Example

```
   /*******************************************************/
   /* The following example translates, on keyboard input, */
   /* all vowels to the letter 'a'.                        */
   /*******************************************************/

static char working_buf[256];

int
   install_xlate_table ()
   {
      int i;

      for (i = 0x00; i <= 0xff; i++)
         working_buf[i] = i;

      working_buf[0x65] = 0x61;  /* change 'e' to 'a' */
      working_buf[0x69] = 0x61;  /* change 'i' to 'a' */
      working_buf[0x6f] = 0x61;  /* change 'o' to 'a' */
      working_buf[0x75] = 0x61;  /* change 'u' to 'a' */

      sm_xlate_table (XLATE_INPUT, working_buf);
      return (0);
   }
```

sm_xm_get_base_window

Gets a Widget ID to the base window

```
#include <smxmuser.h>
Widget sm_xm_get_base_window(void);
```

Environment: Motif

Returns:

- The base window's Widget ID.
- Failure: 0

Description: `sm_xm_get_base_window` gets a Widget ID to the base window that you can pass to the Motif window manager.

See Also: `sm_drawingarea`

sm_xm_get_display

Gets a pointer to the current display

```
#include <smxmuser.h>
Display *sm_xm_get_display(void);
```

Environment Motif

Returns

- A pointer to the current display.
- Failure: 0

Description `sm_xm_get_display` gets a pointer to the current display that you can pass to the Motif window manager.

sm_*xml_export

Generates XML for annotated widgets

```
char *sm_xml_export();
char *sm_n_xml_export(char *gsd);
char *sm_obj_xml_export(int *objid);
```

`gsd`
: An expression indicating the screen or LDB to use for XML generation. For example, `@screen_num(-1)` specifies the next to top screen on the form stack, and `@ldb("customer_xml.scr")` specifies the `customer_xml.scr` screen in the LDB.

`objid`
: An object id indicating the screen or LDB to use for XML generation.

Returns

- Success: A character string containing the generated XML
- Failure: Null pointer

Description

The `sm_*xml_export` functions generate XML for the specified screen. You can call `sm_xml_export` to generate XML for the current screen.

In order to be included in the XML, widgets must have the `xml_tag` property specified. In addition, the screen must have a value in the `xml_tag` property or both the `xml_prefix` and `xml_postfix` properties. For more information on using these functions, see Chapter 22, “Using XML Data,” in the *Application Development Guide*.

Do not call `sm_ffree` to free the returned value. This value will be freed on the next call to one of these functions.

sm_*xml_export_file

Generates XML for annotated widgets to a file

```
int sm_xml_export_file (char *filename);
int sm_n_xml_export_file (char *filename, char *gsd);
int sm_obj_xml_export_file (char *filename, int *objid);
```

`filename`
: The name of the file to contain the generated XML.

`gsd`
: An expression indicating the screen or LDB to use for XML generation. For example, `@screen_num(-1)` specifies the next to top screen on the form stack, and `@ldb("customer_xml.scr")` specifies the `customer_xml.scr` screen in the LDB.

`objid`
: An object id indicating the screen or LDB to use for XML generation.

Returns

- 0 Success.
- Failure: One of the `PR_E_` error messages:
  - `PR_E_OBJID`: The specified object cannot be found.

Description

The `sm_*xml_export_file` functions generate XML for the specified screen and write it to the specified file. You can call `sm_xml_export_file` to generate XML for the current screen.

In order to be included in the XML, widgets must have the `xml_tag` property specified. In addition, the screen must have a value in the `xml_tag` property or both the `xml_prefix` and `xml_postfix` properties.

For more information on using these functions, see Chapter 22, “Using XML Data,” in the *Application Development Guide*.

sm_*xml_import

Import data from XML in a character string

```
int sm_xml_import(char *xmlbuf);
int sm_n_xml_import(char *gsd, char *xmlbuf);
int sm_obj_xml_import(int *objid, char *xmlbuf);
```

`xmlbuf`
: A buffer containing the XML to import.

`gsd`
: An expression indicating the screen or LDB that will receive data from the XML import. For example, `@screen_num(-1)` specifies the next to top screen on the form stack, and `@ldb("customer_xml.scr")` specifies the `customer_xml.scr` screen in the LDB.

`objid`
: An object id indicating the screen or LDB receiving the XML import.

Returns

- 0 Success.
- Failure: One of the `PR_E_` error messages:
  - `PR_E_OBJID`: The specified object cannot be found.

Description

`sm_xml_import` updates objects in the screen from data in the imported XML. In order for the XML import to work, the tags associated with widgets and their corresponding containers must match the tags in the XML file. For more information on using these functions, see Chapter 22, "Using XML Data," in the *Application Development Guide*.

sm_*xml_import_file

Import data from an XML file

```
int sm_xml_import_file (char *filename);
int sm_n_xml_import_file (char *filename, char *gsd);
int sm_obj_xml_import_file (char *filename, int *objid);
```

`filename`
: The name of the XML file.

`gsd`
: An expression indicating the screen or LDB receiving the XML import. For example, `@screen_num(-1)` specifies the next to top screen on the form stack, and `@ldb("customer_xml.scr")` specifies the `customer_xml.scr` screen in the LDB.

`objid`
: An object id indicating the screen or LDB receiving the XML import.

Returns

- 0 Success.
- Failure: One of the `PR_E_` error messages:
  - `PR_E_OBJID`: The specified object cannot be found.

Description

`sm_xml_import_file` reads the specified XML file and updates the specified objects. In order for the XML import to work, the tags associated with widgets and their corresponding containers must match the tags in the XML file. For more information on using these functions, see Chapter 22, “Using XML Data,” in the *Application Development Guide*.

CHAPTER

6 Java Library Function Interfaces

This chapter contains a listing of the library functions in each of Panther's Java library function interfaces.

CFunctionsInterface

Panther general library function interface

```
public interface CFunctionsInterface
```

Methods

```
int sm_allget(int a1);
int sm_append_bundle_data(String a1, int a2, String a3);
int sm_append_bundle_done(String a1);
int sm_append_bundle_item(String a1);
void sm_backtab();
void sm_bel();
int sm_bkrect(int a1, int a2, int a3, int a4, int a5);
int sm_c_com_obj_create(String a1);
void sm_c_off();
void sm_c_on();
void sm_c_vis(int a1);
int sm_calc(int a1, int a2, String a3);
void sm_cancel(int a1);
int sm_ckdigit(int a1, String a2, int a3, int a4, int a5);
void sm_cl_all_mdts();
int sm_clear_array(int a1);
int sm_close_screen();
int sm_close_window();
int sm_com_attach(String a1);
String sm_com_call(String a1);
String sm_com_call_method(String a1);
int sm_com_delete_id(int a1);
```

```
String sm_com_get_prop(int a1, String a2);
String sm_com_get_property(int a1, String a2);
int sm_com_obj_create(String a1);
int sm_com_obj_destroy(int a1, int a2);
void sm_com_onerror(String a1);
int sm_com_receive_args(String a1);
int sm_com_result();
String sm_com_result_msg();
int sm_com_return_args(String a1);
int sm_com_set_handler(int a1, String a2, String a3);
int sm_com_set_prop(int a1, String a2, String a3);
int sm_com_set_property(int a1, String a2, String a3);
int sm_copyarray(int a1, int a2);
int sm_create_bundle(String a1);
void sm_d_msg_line(String a1, int a2);
int sm_d_msg_read(String a1, int a2, int a3);
int sm_dd_able(int a1);
int sm_delay_cursor(int a1);
int sm_disp_off();
double sm_djplcall(String a1);
String sm_fi_path(String a1);
int sm_file_copy(String a1, String a2, String a3);
int sm_file_exists(String a1);
int sm_file_move(String a1, String a2, String a3);
int sm_file_remove(String a1);
int sm_filetypes(String a1, String a2);
int sm_fio_a2f(String a1, String a2);
int sm_fio_close(int a1);

int sm_fio_editor(String a1);

int sm_fio_error();

int sm_fio_error_set(int a1);

int sm_fio_f2a(String a1, String a2);

int sm_fio_getc(int a1);

String sm_fio_gets(int a1, int a2);

int sm_fio_open(String a1, String a2);

int sm_fio_putc(int a1, int a2);

int sm_fio_puts(String a1, int a2);

int sm_fio_rewind(int a1);

void sm_flush();

int sm_free_bundle(String a1);

String sm_get_bundle_data(String a1, int a2, int a3);

int sm_get_bundle_item_count(String a1);

int sm_get_bundle_occur_count(String a1, int a2);

String sm_get_next_bundle_name(String a1);

int sm_getenv(String a1);

int sm_getkey();

int sm_h_ldb_fld_get(int a1, int a2, int a3);

int sm_h_ldb_fld_store(int a1, int a2);

int sm_h_ldb_n_fld_get(int a1, int a2, String a3);

int sm_h_ldb_n_fld_store(int a1, String a2);

int sm_hlp_by_name(String a1);

int sm_home();

int sm_i_amt_format(String a1, int a2, String a3);

double sm_i_dblval(String a1, int a2);
```

```
int sm_i_dlength(String a1, int a2);
String sm_i_fptr(String a1, int a2);
int sm_i_intval(String a1, int a2);
int sm_i_is_no(String a1, int a2);
int sm_i_is_yes(String a1, int a2);
int sm_i_itofield(String a1, int a2, int a3);
int sm_i_ldb_h_putfield(String a1, int a2, int a3, String a4);
int sm_i_ldb_putfield(String a1, int a2, String a3, String a4);
int sm_i_putfield(String a1, int a2, String a3);
String sm_i_strip_amt_ptr(String a1, int a2, String a3);
int sm_input(int a1);
int sm_inquire(int a1);
String sm_inst_script(int a1);
int sm_is_bundle(String a1);
int sm_iset(int a1, int a2);
int sm_isselected(String a1, int a2);
int sm_issv(String a1);
int sm_jclose();
int sm_jfilebox(String a1, String a2, String a3, String a4, int a5);
int sm_jform(String a1);
int sm_jplcall(String a1);
int sm_jplpublic(String a1);
int sm_jplunload(String a1);
int sm_jwindow(String a1);
int sm_key_integer(String a1);
int sm_keyfilter(int a1);
int sm_keyhit(int a1);
```

```
String sm_keylabel(int a1);

int sm_keyoption(int a1, int a2, int a3);

int sm_l_at_cur(int a1, String a2);

int sm_l_close(int a1);

int sm_l_form(int a1, String a2);

int sm_l_open(String a1);

int sm_l_open_syslib(String a1);

int sm_l_window(int a1, String a2, int a3, int a4);

void sm_last();

int sm_launch(String a1);

int sm_ldb_get_active();

int sm_ldb_get_inactive();

int sm_ldb_get_next_active(int a1);

int sm_ldb_get_next_inactive(int a1);

int sm_ldb_h_putfield(int a1, int a2, String a3);

int sm_ldb_h_state_get(int a1, int a2);

int sm_ldb_h_state_set(int a1, int a2, int a3);

int sm_ldb_h_unload(int a1);

int sm_ldb_handle(String a1);

int sm_ldb_is_loaded(String a1);

int sm_ldb_load(String a1);

String sm_ldb_name(int a1);

int sm_ldb_next_handle(int a1);

int sm_ldb_pop();

int sm_ldb_push();

int sm_ldb_putfield(int a1, String a2, String a3);

int sm_ldb_state_get(String a1, int a2);
int sm_ldb_state_set(String a1, int a2, int a3);

int sm_ldb_unload(String a1);

int sm_list_objects_count(int a1);

void sm_list_objects_end(int a1);

int sm_list_objects_next(int a1);

int sm_list_objects_start(int a1);

int sm_load_screen(String a1);

int sm_log(String a1);

int sm_lstore();

void sm_m_flush();

int sm_menu_bar_error();

int sm_menu_change(int a1, String a2, String a3, int a4, int a5,
    String a6);

int sm_menu_create(int a1, String a2, String a3);

int sm_menu_delete(int a1, String a2, String a3);

int sm_menu_get_int(int a1, String a2, String a3, int a4);

String sm_menu_get_str(int a1, String a2, String a3, int a4);

int sm_menu_install(int a1, int a2, String a3, String a4);

int sm_menu_remove(int a1);

int sm_message_box(String a1, String a2, int a3, String a4);

int sm_mnitem_change_i_any(String a1, String a2, int a3, int a4,
    int a5);

int sm_mnitem_change_i_app(String a1, String a2, int a3, int a4,
    int a5);

int sm_mnitem_change_i_field(String a1, String a2, int a3, int a4,
    int a5);

int sm_mnitem_change_i_screen(String a1, String a2, int a3, int a4,
    int a5);

int sm_mnitem_change_s_any(String a1, String a2, int a3, int a4,
    String a5);
```

```
int sm_mnitem_change_s_app(String a1, String a2, int a3, int a4,
    String a5);

int sm_mnitem_change_s_field(String a1, String a2, int a3, int a4,
    String a5);

int sm_mnitem_change_s_screen(String a1, String a2, int a3, int a4,
    String a5);

int sm_mnitem_create(int a1, String a2, String a3, int a4, int a5,
    String a6);

int sm_mnitem_delete(int a1, String a2, String a3, int a4);

int sm_mnitem_get_int(int a1, String a2, String a3, int a4, int a5);

String sm_mnitem_get_str(int a1, String a2, String a3, int a4,
    int a5);

int sm_mnscript_load(int a1, String a2);

int sm_mnscript_unload(int a1, String a2);

int sm_ms_inquire(int a1);

void sm_msg(int a1, int a2, String a3);

int sm_msg_del(int a1);

String sm_msg_get(int a1);

String sm_msgfind(int a1);

int sm_n_amt_format(String a1, String a2);

int sm_n_clear_array(String a1);

double sm_n_dblval(String a1);

int sm_n_dlength(String a1);

int sm_n_dtofield(String a1, double a2, String a3);

String sm_n_fptr(String a1);

int sm_n_intval(String a1);

int sm_n_is_no(String a1);

int sm_n_is_yes(String a1);

int sm_n_itofield(String a1, int a2);

int sm_n_keyinit(String a1);
int sm_n_ldb_fld_store(String a1, int a2);

int sm_n_ldb_h_fldno(String a1, int a2);

int sm_n_ldb_h_putfield(String a1, int a2, String a3);

int sm_n_ldb_n_fld_get(int a1, String a2, String a3);

int sm_n_ldb_n_fld_store(String a1, String a2);

int sm_n_ldb_putfield(String a1, String a2, String a3);

int sm_n_length(String a1);

int sm_n_max_occur(String a1);

int sm_n_mnitem_change_i_any(String a1, String a2, String a3,
    int a4, int a5);

int sm_n_mnitem_change_i_app(String a1, String a2, String a3,
    int a4, int a5);

int sm_n_mnitem_change_i_field(String a1, String a2, String a3,
    int a4, int a5);

int sm_n_mnitem_change_i_screen(String a1, String a2, String a3,
    int a4, int a5);

int sm_n_mnitem_change_s_any(String a1, String a2, String a3,
    int a4, String a5);

int sm_n_mnitem_change_s_app(String a1, String a2, String a3,
    int a4, String a5);

int sm_n_mnitem_change_s_field(String a1, String a2, String a3,
    int a4, String a5);

int sm_n_mnitem_change_s_screen(String a1, String a2, String a3,
    int a4, String a5);

int sm_n_mnitem_delete(int a1, String a2, String a3, String a4);

int sm_n_mnitem_get_int(int a1, String a2, String a3, String a4,
    int a5);

String sm_n_mnitem_get_str(int a1, String a2, String a3, String a4,
    int a5);

int sm_n_msg_read(String a1, int a2, int a3, String a4);

int sm_n_num_occurs(String a1);

int sm_n_putfield(String a1, String a2);
```

```
String sm_n_strip_amt_ptr(String a1, String a2);
int sm_n_validate(String a1);
int sm_n_wselect(String a1);
int sm_n_ww_length(String a1);
int sm_n_ww_write(String a1, String a2, int a3);
int sm_n_xml_export_file(String a1, String a2);
int sm_n_xml_import(String a1, String a2);
int sm_n_xml_import_file(String a1, String a2);
String sm_name(int a1);
int sm_next_sync(int a1);
void sm_nl();
int sm_o_ldb_h_putfield(int a1, int a2, int a3, String a4);
int sm_o_ldb_putfield(int a1, int a2, String a3, String a4);
int sm_o_off_gofield(int a1, int a2, int a3);
String sm_obj_call(String a1);
int sm_obj_copy(String a1, String a2);
int sm_obj_copy_id(int a1, int a2);
int sm_obj_create(String a1);
int sm_obj_delete(String a1);
int sm_obj_delete_id(int a1);
String sm_obj_get_property(int a1, String a2);
String sm_obj_onerror(String a1);
int sm_obj_set_property(int a1, String a2, String a3);
int sm_obj_sort(int a1, int a2);
int sm_obj_sort_auto(int a1);
int sm_obj_xml_export_file(String a1, int a2);
int sm_obj_xml_import(int a1, String a2);
int sm_obj_xml_import_file(int a1, String a2);

int sm_option(int a1, int a2);

int sm_optmnu_id();

int sm_popup_at_cur();

int sm_prop_error();

double sm_prop_get_dbl(int a1, int a2);

int sm_prop_get_int(int a1, int a2);

double sm_prop_get_m_dbl(int a1, int a2, int a3);

int sm_prop_get_m_int(int a1, int a2, int a3);

String sm_prop_get_m_str(int a1, int a2, int a3);

String sm_prop_get_str(int a1, int a2);

double sm_prop_get_x_dbl(int a1, int a2, int a3);

int sm_prop_get_x_int(int a1, int a2, int a3);

String sm_prop_get_x_str(int a1, int a2, int a3);

int sm_prop_id(String a1);

int sm_prop_id_app();

int sm_prop_id_element(int a1, String a2);

int sm_prop_id_element_num(int a1, int a2);

int sm_prop_id_screen(String a1, int a2);

int sm_prop_id_screen_num(int a1);

int sm_prop_id_widget(int a1, String a2);

int sm_prop_id_widget_num(int a1, int a2);

int sm_prop_name_to_id(String a1);

int sm_prop_set_int(int a1, int a2, int a3);

int sm_prop_set_m_int(int a1, int a2, int a3, int a4);

int sm_prop_set_m_str(int a1, int a2, int a3, String a4);

int sm_prop_set_str(int a1, int a2, String a3);
int sm_prop_set_x_int(int a1, int a2, int a3, int a4);

int sm_prop_set_x_str(int a1, int a2, int a3, String a4);

String sm_pset(int a1, String a2);

int sm_r_at_cur(String a1);

int sm_r_form(String a1);

int sm_r_window(String a1, int a2, int a3);

int sm_raise_exception(int a1, String a2);

int sm_receive(String a1);

int sm_receive_args(String a1);

void sm_rescreen();

int sm_resize(int a1, int a2);

int sm_return_args(String a1);

int sm_s_val();

String sm_save_screen(int a1, String a2);

String sm_sb_gettext(int a1);

String sm_sdtime(String a1);

int sm_send(String a1);

void sm_set_help();

void sm_setbkstat(String a1, int a2);

void sm_setsibling();

void sm_setstatus(int a1);

int sm_sh_off();

int sm_shell(String a1, int a2);

void sm_shrink_to_fit();

String sm_sjplcall(String a1);

int sm_slib_error();

int sm_slib_install(String a1, int a2, int a3);
int sm_slib_load(String a1);

String sm_soption(int a1, String a2);

void sm_tab();

String sm_tmpnam();

int sm_ungetkey(int a1);

int sm_unload_screen(String a1);

int sm_wcount();

int sm_wdeselect();

int sm_winsize();

int sm_wrotate(int a1);

int sm_wselect(int a1);

int sm_ww_length(int a1);

int sm_ww_write(int a1, String a2, int a3);

int sm_xml_export(String a1);

int sm_xml_import(String a1);

int sm_xml_export_file(String a1);

int sm_xml_import_file(String a1);
```

Environment Java only

Description To get an object of type `CFunctionsInterface`, call the `getCFunctions` method. The `getCFunctions` method is supported by all the Java objects that represent Panther objects. An object of type `CFunctionsInterface` implements methods that correspond to the core Panther library functions.

ComFunctionsInterface

Panther library function interface for MTS applications

```
public interface ComFunctionsInterface
```

Methods

```
int log (String text, int code);
int raise_exception (int code);
int receive_args (String text);
int return_args (String text);
int sm_mts_CreateInstance (String text);
int sm_mts_CreateProperty (String group, String prop);
int sm_mts_CreatePropertyGroup (String group);
int sm_mts_DisableCommit ();
int sm_mts_EnableCommit ();
String sm_mts_GetPropertyValue (String group, String prop);
int sm_mts_IsCallerInRole (String role);
int sm_mts_IsInTransaction ();
int sm_mts_IsSecurityEnabled ();
int sm_mts_PutPropertyValue (String group, String prop, String
    value);
int sm_mts_SetAbort ();
int sm_mts_SetComplete ();
```

Environment Java only for COM/MTS

Description Objects that implement this interface provide access to functions that are of use in service components running under COM/MTS. Java methods that implement a service component's public methods are passed an object of type

Example ComFunction sInterface as a parameter.

Additional COM functions, such as `sm_obj_call` and `sm_com_result`, are implemented as part of the `CFunctionsInterface`.

DMFunctionsInterface

Panther general library function interface

```
public interface DMFunctionsInterface
```

Methods

```
int dm_convert_empty(int a1);
String dm_cursor_connection(String a1);
int dm_cursor_consistent(String a1);
String dm_cursor_engine(String a1);
int dm_dbms(String a1);
int dm_dbms_noexp(String a1);
int dm_get_connection_option(String a1, String a2);
int dm_get_driver_option(String a1, String a2);
int dm_is_connection(String a1);
int dm_is_cursor(String a1);
int dm_is_engine(String a1);
int dm_set_con_pool_size(int a1);
int dm_set_connection_option(String a1, String a2, int a3);
int dm_set_driver_option(String a1, String a2, int a3);
int dm_set_max_fetches(int a1);
int dm_set_max_rows_per_fetch(int a1);
```

Environment Java only

Description To get an object that implements `DMFunctionsInterface`, call the `getDMFunctions` method. The `getDMFunctions` method is supported by all the Java objects that represent Panther objects. The methods implemented by an object of type `DMFunctionsInterface` correspond to the Panther database interface library functions.

RWFunctionsInterface

Panther library function interface for reports

```
public interface RWFunctionsInterface
```

Methods

```
String sm_rw_error_message ();

int sm_rw_play_metafile (String metatfileName);

int sm_rw_runreport (String reportName);
```

Environment Java only

Description Objects that support this interface provide access to functions that are of use in implementing reports. Java methods that implement the public methods of reports are passed an object of type `RWFunctionsInterface` as a parameter.

TMFunctionsInterface

Panther transaction manager function interface

```
public interface TMFunctionsInterface
```

Methods

```
int dm_disable_styles();

int dm_enable_styles();

int dm_exec_sql(int a1, String a2);

int dm_free_sql_info(int a1);

int dm_gen_change_execute_using(String a1, String a2, String a3,
    int a4, int a5, int a6);

int dm_gen_change_select_from(String a1, String a2, String a3,
    int a4);

int dm_gen_change_select_group_by(String a1, String a2, int a3);

int dm_gen_change_select_having(String a1, String a2, int a3);

int dm_gen_change_select_list(String a1, String a2, String a3,
    int a4);

int dm_gen_change_select_order_by(String a1, String a2, int a3,
    int a4);

int dm_gen_change_select_suffix(String a1, String a2);

int dm_gen_change_select_where(String a1, String a2, int a3);

String dm_gen_get_tv_alias(String a1);

int dm_gen_sql_info(int a1, String a2);

int dm_set_tm_clear_fast(int a1);

void dm_val_relative();

int sm_bi_compare();

int sm_bi_copy();

int sm_bi_initialize();

int sm_tm_clear(int a1);
```

```
void sm_tm_clear_model_events();
int sm_tm_clear_no_select(int a1);
int sm_tm_command(String a1);
int sm_tm_command_emsgset(String a1, int a2);
int sm_tm_command_errset(String a1, int a2);
int sm_tm_continuation_validity(int a1);
int sm_tm_dbi_checker(int a1);
int sm_tm_dbms(String a1);
void sm_tm_error(String a1, String a2, String a3, int a4);
int sm_tm_errorlog(int a1, int a2, String a3);
int sm_tm_event(String a1);
String sm_tm_event_name(int a1);
int sm_tm_failure_message(int a1, String a2, String a3);
int sm_tm_handling(int a1);
int sm_tm_inquire(int a1);
int sm_tm_iset(int a1, int a2);
void sm_tm_msg_count_error(String a1, int a2, int a3);
void sm_tm_msg_emsg(String a1, int a2);
void sm_tm_msg_error(String a1, int a2);
int sm_tm_old_bi_context(int a1);
String sm_tm_pinquire(int a1);
int sm_tm_pop_model_event();
int sm_tm_pset(int a1, String a2);
int sm_tm_push_model_event(int a1);
int sm_tm_set_max_bind_name_len(int a1);
int sm_tm_set_save_backward(int a1);
int sm_tm_synchronization(int a1);
int sm_tm_update_pkey();
```

Environment Java only

Description To get an object of type `TMFunctionsInterface`, call the method `getTMFunctions`. The methods implemented by objects of this type correspond to the Prolifics transaction manager library functions. The `getTMFunctions` method is supported by all the Java objects that represent Panther objects.

TPFunctionsInterface

Panther library function interface for service components in JetNet and Oracle Tuxedo

```
public interface TPFunctionsInterface
```

Methods

```
int sm_tp_exec(String a1);

WidgetInterface getTpRequest();

WidgetInterface getTpRequest(String callid);
```

Environment

Java only for JetNet and Oracle Tuxedo

Description

Code in a client screen in a Oracle Tuxedo or JetNet application can get a handle to an object that implements `TPFunctionsInterface` by calling the `getTPFunctions` method. Methods of service components that implement services in a Oracle Tuxedo or JetNet application receive an object of type `TPFunctionsInterface` as a parameter.

The method `getTPRequest` returns a handle to an object that represents a service request. These objects implement `WidgetInterface`. Interactions with such an object will generally only be for the purpose of querying its property values.

The method `sm_tp_exec` corresponds to the Panther library function of the same name.

WSFunctionsInterface

Library function interface for Enterprise JavaBeans operating in WebSphere

```
public interface WSFunctionsInterface
```

Methods

```
PantherSessionBean get_bean();

int log (String message);

void raise_exception (String message);

int receive_args (String args);

int return_args (String args);
```

Environment

Java only for Enterprise JavaBeans on WebSphere

Description

Objects that support this interface provide access to functions that are of use in implementing Enterprise JavaBeans deployed on WebSphere Application Server. Java methods that implement the public methods of Enterprise JavaBeans are passed an object of type `WSFunctionsInterface` as a parameter.

CHAPTER

7 Java Object Interfaces

This chapter contains descriptions of Panther's Java object interfaces arranged alphabetically. Panther objects are referenced in Java code by means of Java objects of types that are defined by these interfaces.

These interfaces define the methods supported by such objects and would be used to perform operations on those objects.

Information about each interface is organized into the following sections:

- Methods supported by objects that implement the interface.
- Description of the interface.

ApplicationInterface

Interface definition for the application as a whole

```
public interface ApplicationInterface extends WidgetInterface
```

Methods

getScreen

```
ScreenInterface getScreen();
ScreenInterface getScreen(int level);
ScreenInterface getScreen(String name);
ScreenInterface getScreen(String name, int instance);
```

getWidget

```
WidgetInterface getWidget(int id);
```

Environment Java only

Description One of the methods of `WidgetInterface`, and therefore common to all widgets, is `getApplication`. This returns an object that represents the application as a whole. The methods that get and set properties can be used on this object to program application-scope properties.

The `getScreen` method returns an object corresponding to a screen. It has four variants. When called with no parameters, it returns an object corresponding to the current screen (the top of the window stack). When called with one integer parameter, the integer specifies the zero-based offset in the window stack, as used by the Panther library function `sm_wselect`. When called with one string parameter, the string specifies the screen's name. To get an object corresponding to a specific instance of a screen (you can have more than one copy of a screen open at a time), call `getScreen` with both the name and an integer that corresponds to the instance of the screen you wish to specify.

The `getWidget` method returns an object corresponding to a widget given an object id for the widget. If the widget is a screen, field, group, etc., the object returned will be cast to the appropriate type. In other words, the object returned will not merely implement `WidgetInterface`, but will implement `FieldInterface`, `ScreenInterface`, etc., as appropriate.

FieldInterface

Interface definition for fields

```
public interface FieldInterface extends WidgetInterface
```

Methods

```
amt_format
       int amt_format(String value);
       int amt_format(int item, String value);

clear_array
       int clear_array();

dblval
       double dblval();
       double dblval(int item);

dtofield
       int dtofield(double value, String format);
       int dtofield(int item, double value, String format);

fval
       int fval();
       int fval(int item);

getfield
       String getfield();
       String getfield(int item);

getScreen
       ScreenInterface getScreen();

gofield
       int gofield();
       int gofield(int item);

intval
       int intval();
       int intval(int item);

ioccur
       int ioccur(int count);
       int ioccur(int item, int count);
```

```
is_no
        boolean is_no();
        boolean is_no(int item);

is_null
        boolean is_null();
        boolean is_null(int item);

is_yes
        boolean is_yes();
        boolean is_yes(int item);

itofield
        int itofield(int value);
        int itofield(int item, int value);

off_gofield
        int off_gofield(int offset);
        int off_gofield(int item, int offset);

putfield
        int putfield(String text);
        int putfield(int item, String text);

ww_read
        String ww_read();
        String ww_read(int offset);
```

Environment Java only

Description `FieldInterface` defines the methods for objects representing fields. In addition, objects of type `FieldInterface` support all the methods in `WidgetInterface`.

Java objects representing text fields, push buttons, toggle buttons, check boxes, radio buttons, dynamic labels, tab cards, option menus, combo boxes and scales are of type `FieldInterface`.

The method `getScreen` will return an object corresponding to the screen on which the field in question is found.

The other methods correspond in functionality to Panther library functions that have the same name, only with an `sm_` prefix. Those library functions take parameters to indicate which field the function should operate on. Since these are methods of objects that correspond to fields, the field in question is always implicitly indicated.

GridInterface

Interface definition for grids

```
public interface GridInterface extends WidgetInterface
```

Methods

```
getColumn
       FieldInterface getColumn(int n);

num_columns
       int numColumns();
```

Environment

Java only

Description

`GridInterface` defines the methods for objects representing grid widgets. In addition, objects of type `GridInterface` support all the methods in `WidgetInterface`.

The method `numColumns` returns the number of columns in the grid; the method `getColumn` returns an object corresponding to a given field in the grid, referenced by column number.

GroupInterface

Interface definition for groups

```
public interface GroupInterface extends WidgetInterface
```

Methods

```
deselect
       void deselect(int item);

getMember
       FieldInterface getMember(int n);

isSelected
       boolean isSelected(int item);

numMembers
       int numMembers();

select
       void select(int item);
```

Environment Java only

Description `GroupInterface` defines the methods for objects representing groups. In addition, objects of type `GroupInterface` support all the methods in `WidgetInterface`.

The method `numMembers` returns the number of members in the group; the method `getMember` returns an object corresponding to a given member, referenced by number.

ScreenInterface

Interface definition for screens

```
public interface ScreenInterface extends WidgetInterface
```

Methods

```
getField
      FieldInterface getField(String name);
      FieldInterface getField(int fieldnum);

getWidget
      WidgetInterface getWidget(String name);
```

Environment

Java only

Description

`ScreenInterface` defines the methods for objects representing screens. In addition, objects of type `ScreenInterface` supports all the methods in `WidgetInterface`.

The method `getField` returns an object corresponding to a particular field on the screen. The methods of this object can then be used perform operations on that field. Field objects can be obtained by field name or by field number. If you refer to an object by name and the object is not a field, the `getField` method will return null.

To get handles to objects that are not fields, use the `getWidget` method. This will return a generic widget handle that can then be cast to a handle of a given type (grid or group).

WidgetInterface

Interface definition for widgets

```
public interface WidgetInterface
```

Methods

```
getApplication
       ApplicationInterface getApplication();

get_dbl
       double get_dbl(int prop);
       double get_dbl(int item, int prop);

get_int
       int get_int(int prop);
       int get_int(int item, int prop);

getServer
       ServerInterface getServer();

get_str
       String get_str(int prop);
       String get_str(int item, int prop);

set_dbl
       int set_dbl(int prop, double value);
       int set_dbl(int item, int prop, double value);

set_int
       int set_int(int prop, int value);
       int set_int(int item, int prop, int value);

set_str
       int set_str(int prop, String value);
       int set_str(int item, int prop, String value);

Library Function Interfaces
       CFunctionsInterface  getCFunctions();
       ComFunctionsInterface getcomFunctions();
       DMFunctionsInterface getDMFunctions();
       RWFunctionsInterface getRWFunctions();
       TMFunctionsInterface getTMFunctions();
       TPFunctionsInterface getTPFunctions();
       WSFunctionsInterface getWSFunctions();
```

Environment `Java only`

Description `WidgetInterface` defines the methods that are common to all objects that correspond to Panther objects.

The `get_int`, `get_str`, and `get_dbl` methods are used to get property values. These methods come in two variants, using either one parameter or two. The version with two parameters is for widgets in arrays. The first parameter is the occurrence number. Use the `PR_` values to identify the property requested.

The `set_int`, `set_str`, and `set_dbl` methods are used to set property values. These methods come in two variants, one with two parameters and one with three. The version with three parameters is used when referencing a particular occurrence in an array. The final parameter is the new value for the property.

The `get*Functions` methods are used to get handles to special objects that exist to support the Panther library functions. For a listing of the methods in each library function interface, refer to Chapter 6, “Java Library Function Interfaces.”

CHAPTER

8 Transaction Manager Commands

This chapter describes the `sm_tm_command` function and the transaction commands (listed alphabetically) that can be called using this function.

Each reference page includes the following information:

- Syntax—Lists the command and its parameters.
- Description—Gives an explanation of the command.
- Sequence—Lists other transaction manager commands that might be needed before or after this command.
- Events—Lists the transaction request events and slice events that can be generated with a command. This information is useful when writing a transaction event function to change the processing in a request or when modifying the transaction model. For information on writing transaction event functions, refer to Chapter 32, “Writing Transaction Event Functions,” in the *Application Development Guide*.

Some requests refer to the following transaction attributes:

Table 8-1 Transaction Attributes

Attribute	Description
`TM_FULL`	Indicator of whether it is a full (`1`) or partial (`0`) command.
`TM_OCC`	Occurrence number being processed.
`TM_OCC_COUNT`	The number of occurrences in the table view.
`TM_STATUS`	Error indicator.
`TM_VALUE`	General purpose integer.

To test the value of transaction attributes, use the library functions `sm_tm_inquire`, `sm_tm_pinquire`, or `sm_tm_pcopy`. To set transaction attributes, use the library functions `sm_tm_iset` and `sm_tm_pset`

sm_tm_command

Executes a transaction command

```
#include <tmusubs.h>

int sm_tm_command (cmd_string);
```

Arguments *`cmd_string`*

Contains one of the following transaction commands and its associated parameters. The parameters can include a table view name and/or command scope. Refer to the specific command for details and command syntax.

CHANGE	FORCE_CLOSE
CLEAR	NEW
CLOSE	REFRESH
CONTINUE	RELEASE
CONTINUE_BOTTOM	SAVE
CONTINUE_DOWN	SELECT
CONTINUE_TOP	START
CONTINUE_UP	VIEW
COPY	WALK_DELETE
COPY_FOR_UPDATE	WALK_INSERT
COPY_FOR_VIEW	WALK_SELECT
FETCH	WALK_UPDATE
FINISH	

Returns

- `STATUS` of the current transaction.

Description

`sm_tm_command` executes the specified transaction manager command.

When specifying a command, the table view name is case sensitive; however, the command name and the optional parameters following the table view name are not case sensitive.

By definition, a command is in progress from the moment `sm_tm_command` is called until the moment it returns. As it processes most commands, `sm_tm_command` invokes transaction event functions and transaction models. These, in turn, should not invoke transaction manager commands, because the transaction manager cannot process its commands recursively. This implies that they should not close the active screen (which triggers a `FINISH` command), or cause any other screen to be displayed that contains table views (which triggers a `CHANGE` command).

Transaction Modes

After recognizing a transaction command, the transaction manager either sets the transaction mode or checks the transaction mode to see if the specified command is available with the current mode. If the command is not supported in the current mode, or if the command is not recognized, then the transaction manager displays an error message that the mode does not permit the specified command. It also sets the value of `TM_STATUS` to -1, which causes `sm_tm_command` to return a value of -1. For more information on command availability in transaction modes, refer to “Setting the Transaction Mode” on page 34-7 in *Application Development Guide*.

Tree Traversal

The transaction tree is the group of linked table views that are part of the current transaction manager transaction. After a command is issued, the transaction manager traverses the transaction tree, issuing the request and slice events defined for that command for each table view and performing the processing defined for each event in transaction event functions and transaction models. The most common order is referred to as table/server view order. A server view is defined as:

- A single table view having no server links to other table views.
- A group of table views connected by server links.

Tree traversal in table/server view order begins at the root table view or at the specified table view. The traversal covers all table views within the server view, and then moves on to the next server view. The Parent and Child properties for each link help determine the traversal order. The tree traversal reaches a parent table view before its child, but there can be intervening table views (in the same and different server views).

Restriction A transaction manager transaction must be in progress in order to call commands. Transactions are created with the `START` command which is called automatically on screen entry. However, the Panther events that occur on screen entry call the unnamed JPL procedure before calling the `START` command. Therefore, transaction manager commands cannot be invoked in the unnamed procedure.

Example

```
int sm_tm_command ("SELECT titles BELOW_TV");
```

Errors Errors in the transaction manager set `TM_STATUS` to `-1`.

In addition, there are return values for transaction models or transaction event functions that set the value of `TM_STATUS`. Table 8-2 lists the return codes, the events that get generated for each return code, and the processing that occurs for the event.

Table 8-2 Return values for transaction event functions and transaction models

Return Code	**Event**	**Processing**
`TM_OK`	None	None.
`TM_PROCEED`	None	Invoke transaction model.
`TM_FAILURE`	`TM_NOTE_FAILURE`	`Call sm_tm_failure_message.`
`TM_UNSUPPORTED`	`TM_NOTE_UNSUPPORTED`	`Call sm_tm_failure_message.`
`TM_CHECK`	`TM_TEST_ERROR`	`Call sm_tm_dbi_checker.`
`TM_CHECK_ONE_ROW`	`TM_TEST_ONE_ROW`	`Call sm_tm_dbi_checker.`
`TM_CHECK_SOME_ROWS`	`TM_TEST_SOME_ROWS`	`Call sm_tm_dbi_checker.`

Events Once you select a transaction command, the transaction manager generates the transaction events defined for that command. These events are defined to perform the processing needed for the command. The major events for each command are called requests. Some requests are further subdivided into more events, called slices. As the transaction manager traverses the tree, it looks for the processing for each event first in a transaction manager event function, then in an engine-specific transaction model, and then in the common transaction model.

The transaction manager has an event stack, onto which the transaction events are pushed. As the events are processed, they are popped from the stack. For more information on the event stack, refer to Chapter 35, “Generating Transaction Manager Events,” in *Application Development Guide*.

Table 8-3 lists all the transaction manager events. A description of the general processing performed by each request or slice is part of the documentation for a command in which it is used. To see the processing done for a particular database engine, refer to the transaction model for that engine. For a summary list of the commands, requests, and slices, refer to Chapter 9, "Transaction Model Events."

Table 8-3 Events available in the transaction manager

Event	Command*
`TM_CLEAR`	`CLEAR`
`TM_CLEAR_SEL_COUNT_FLAG`	`SELECT`, `VIEW`
`TM_CLOSE`	`CLOSE`
`TM_CONTINUE_BOTTOM`	`CONTINUE_BOTTOM`
`TM_CONTINUE_DOWN`	`CONTINUE_DOWN`
`TM_CONTINUE_TOP`	`CONTINUE_TOP`
`TM_CONTINUE_UP`	`CONTINUE_UP`
`TM_COPY`	`COPY`
`TM_COPY_FOR_UPDATE`	`COPY_FOR_UPDATE`
`TM_COPY_FOR_VIEW`	`COPY_FOR_VIEW`
`TM_DELETE`	`SAVE`
`TM_DELETE_DECLARE`	`SAVE`
`TM_DELETE_EXEC`	`SAVE`
`TM_DISCARD`	`CLOSE`
`TM_FETCH`	`FETCH`
`TM_FINISH`	`FINISH`
`TM_GET_SAVE_CURSOR`	`SAVE`
`TM_GET_SEL_CURSOR`	`SELECT`
`TM_GIVE_UP_SAVE_CURSOR`	`SAVE`

Table 8-3 Events available in the transaction manager ***(Continued)***

Event	Command*
TM_GIVE_UP_SEL_CURSOR	SELECT
TM_INSERT	SAVE
TM_INSERT_DECLARE	SAVE
TM_INSERT_EXEC	SAVE
TM_NEW	NEW
TM_NOTE_FAILURE	Part of error processing
TM_NOTE_UNSUPPORTED	Part of error processing
TM_POST_CLEAR	CLEAR
TM_POST_CLOSE	CLOSE
TM_POST_COPY	COPY
TM_POST_COPY_FOR_UPDATE	COPY_FOR_UPDATE
TM_POST_COPY_FOR_VIEW	COPY_FOR_VIEW
TM_POST_NEW	NEW
TM_POST_RELEASE	RELEASE
TM_POST_SAVE	SAVE
TM_POST_SAVE1	SAVE
TM_POST_SAVE2	SAVE
TM_POST_SELECT	SELECT
TM_POST_VAL_LINK	Part of validation link processing
TM_POST_VIEW	VIEW
TM_PRE_CLEAR	CLEAR
TM_PRE_CLOSE	CLOSE
TM_PRE_COPY	COPY

Table 8-3 Events available in the transaction manager *(Continued)*

Event	Command*
TM_PRE_COPY_FOR_UPDATE	COPY_FOR_UPDATE
TM_PRE_COPY_FOR_VIEW	COPY_FOR_VIEW
TM_PRE_NEW	NEW
TM_PRE_RELEASE	RELEASE
TM_PRE_SAVE	SAVE
TM_PRE_SELECT	SELECT
TM_PRE_VAL_LINK	Part of validation link processing
TM_PRE_VIEW	VIEW
TM_PREPARE_CONTINUE	SELECT, VIEW
TM_QUERY	CLOSE
TM_RELEASE	RELEASE
TM_SAVE	SAVE
TM_SAVE_BEGIN	SAVE
TM_SAVE_COMMIT	SAVE
TM_SAVE_ROLLBACK	SAVE
TM_SAVE_SET_MODE	SAVE
TM_SET_SEL_COUNT_FLAG	SELECT, VIEW
TM_SEL_BUILD_PERFORM	SELECT
TM_SEL_CHECK	FETCH
TM_SEL_COUNT_CHECK	SELECT, VIEW
TM_SEL_GEN	SELECT
TM_SELECT	SELECT
TM_START	START

Table 8-3 Events available in the transaction manager *(Continued)*

Event	Command*
`TM_TEST_ERROR`	Part of error processing
`TM_TEST_ONE_ROW`	Part of error processing
`TM_TEST_SOME_ROWS`	Part of error processing
`TM_UPDATE`	`SAVE`
`TM_UPDATE_DECLARE`	`SAVE`
`TM_UPDATE_EXEC`	`SAVE`
`TM_VAL_BUILD_PERFORM`	Part of validation link processing
`TM_VAL_CHECK`	Part of validation link processing
`TM_VAL_GEN`	Part of validation link processing
`TM_VAL_LINK`	Part of validation link processing
`TM_VIEW`	`VIEW`
`TM_WALK_DELETE`	`WALK_DELETE`
`TM_WALK_INSERT`	`WALK_INSERT`
`TM_WALK_SELECT`	`WALK_SELECT`
`TM_WALK_UPDATE`	`WALK_UPDATE`

* *Indicates under which command the event is documented.*

CHANGE

Switches to another transaction

```
int sm_tm_command ("CHANGE transactionName");
```

Arguments

`transactionName`
The name of a valid transaction manager transaction.

Description

`CHANGE` switches to another transaction, making it the current transaction. To use this command, you must specify the transaction name. If the transaction does not exist, the previous transaction remains active. In cases where you move between two screens, the command is automatically issued as part of Panther's screen processing.

To get the current transaction name, call `sm_tm_pinquire`(`TM_TRAN_NAME`).

To specify a new transaction, use the `START` command. Any transaction begun with an explicit call to the `START` command must also be closed by an explicit call to the `FINISH` command.

Events

There are no request events generated by the `CHANGE` command.

Example

Refer to the `START` command.

CLEAR

Clears data in widgets

```
int sm_tm_command ("CLEAR [ tableViewName [ tableViewScope ] ]");
```

Arguments

`tableViewName`

The name of a table view in the current transaction. This parameter is case sensitive.

If *`tableViewName`* is specified, the command is applied according to the *`tableViewScope`* parameter. Since the entire table view tree might not be included, this is known as a partial command, and `sm_tm_command` sets `TM_FULL` to 0.

If *`tableViewName`* is not specified, the command is applied for each table/server view, starting with the root table view. This is known as a full command, and `sm_tm_command` sets `TM_FULL` to 1.

`tableViewScope`

One of the following parameters, which must be preceded by a table view name. `TV_AND_BELOW` which applies the command to the specified table view and all table views below it on the tree. If no parameter is specified, the transaction manager acts as though `TV_AND_BELOW` was supplied. `BELOW_TV` which applies the command to the table views below the specified table view. `TV_ONLY` which applies the command to the specified table view only. `SV_ONLY` which applies the command only to the table views of the specified server view.

Description

`CLEAR` clears the data displayed on the screen for any widget belonging to a valid table view. `CLEAR` has two major uses:

- Clears onscreen data so that you can enter selection criteria for a subsequent `VIEW` or `SELECT`.
- Clears onscreen data so that `SAVE` processing deletes the database rows represented.

In order to delete rows from the database, the table view must be updatable. If the table view is non-updatable, the data is cleared from the screen, but SQL `DELETE` statements are not issued.

The `CLEAR` command does not change the transaction mode.

Push buttons and menu selections for the `CLEAR` command can choose to set the class property to `clear_button`. By default, `clear_button` is active in all transaction modes.

Sequence

To delete rows, `CLEAR` must be followed by the `SAVE` command.

To perform a query-by-example, execute `CLEAR` before entering a value for the `SELECT` or `VIEW` commands.

Events

The following request events can be generated by the `CLEAR` command to ascertain whether the changes from the previous command have been saved and, if desired, discard those changes:

Table 8-4 Request events for CLEAR

Request	Traversal	Typical Processing
`TM_PRE_CLOSE`	Described under `CLOSE`	
`TM_CLOSE`	Described under `CLOSE`	
`TM_QUERY`	Described under `CLOSE`	
`TM_DISCARD`	Described under `CLOSE`	
`TM_POST_CLOSE`	Described under `CLOSE`	
`TM_PRE_CLEAR`	By table/server view from the specified table view	Do nothing
`TM_CLEAR`	By table/server view from the specified table view	Do nothing (`sm_tm_clear` is called for the table view by the transaction manager after this request)
`TM_POST_CLEAR`	By table/server view from the specified table view	Do nothing

CLOSE

Closes the current database transaction, allowing you to discard or save your changes

```
int sm_tm_command ("CLOSE [ tableViewName [ tableViewScope ] ]");
```

Arguments

tableViewName

The name of a table view in the current transaction. This parameter is case sensitive.

If *tableViewName* is specified, the command is applied according to the *tableViewScope* parameter. Since the entire table view tree might not be included, this is known as a partial command, and sm_tm_command sets TM_FULL to 0.

If *table View Name* is not specified, the command is applied for each table/server view, starting with the root table view. This is known as a full command, and sm_tm_command sets TM_FULL to 1.

table View Scope

One of the following parameters, which must be preceded by a table view name. TV_AND_BELOW which applies the command to the specified table view and all table views below it on the tree. If no parameter is specified, the transaction manager acts as though TV_AND_BELOW was supplied. BELOW_TV which applies the command to the table views below the specified table view. TV_ONLY which applies the command to the specified table view only. SV_ONLY which applies the command only to the table views of the specified server view.

Description

If changes are made in table views on which CLOSE operates after a SELECT, NEW, COPY, or COPY_FOR_UPDATE command, CLOSE displays a dialog box which allows users to discard any changes entered. If the user chooses OK, changes are discarded; choosing Cancel, returns the user to the current screen so changes can be saved. In Web applications, this command is the same as the FORCE_CLOSE command.

CLOSE sets the transaction mode to initial unless a table view is specified. In the default styles file (style.sty), the style assigned to initial mode clears any protections on the widgets.

Push buttons and menu selections for the CLOSE command can choose to set the class property to close_button. By default, close_button is inactive in initial mode but active in all other modes.

For some database engines, such as SYBASE CT-Lib, the CLOSE command does not release the database locks when a SELECT command is not followed by a SAVE command. In this case, follow the CLOSE with the RELEASE command which gives up the locks on the database.

Sequence The CLOSE command is useful after SELECT, NEW, COPY, or COPY_FOR_UPDATE in order to discard your changes.

Events

Table 8-5 Request events for CLOSE (if there are screen changes)

Request	Traversal	Typical Processing
TM_PRE_CLOSE	By table/server view from the specified table view	CLOSE or SAVE processing is beginning. (Processing identical for TM_PRE_SAVE)
TM_CLOSE	By table/server view from the specified table view. Traversal ends if TM_VALUE is set to TM_DISCARD_ACTION or TM_EXIT_ACTION.	Appropriate responses are those listed for TM_QUERY below, but typical processing is to do nothing.
TM_QUERY	By table/server view from the specified table view, but restricted to table views, if any, in which there has been a change that would entail a SAVE command. Traversal ends if TM_VALUE is set to TM_DISCARD_ACTION or TM_EXIT_ACTION	A message is chosen according to the value of TM_FULL. If 1, the displayed message is for the complete transaction tree. If 0, the message is for a portion of the tree. sm_message_box, which displays the message, gives a choice of OK and Cancel. TM_DISCARD_ACTION and TM_EXIT_ACTION are the corresponding values passed back to TM_VALUE.
TM_DISCARD	By table/server view from the specified table view	Set a discard flag, consulted by TM_POST_SAVE1
TM_POST_CLOSE	By table/server view from the specified table view	Generates slice events: TM_POST_CLOSE, TM_POST_SAVE1, TM_POST_SAVE2 (described under SAVE, but no save cursor exists.

The TM_CLOSE and TM_QUERY requests have four possible return values: TM_NO_ACTION, TM_DISCARD_ACTION, TM_SAVE_ACTION, and TM_EXIT_ACTION. The distributed transaction models use two of these return values:

- TM_DISCARD_ACTION discards the changes to the data.
- TM_EXIT_ACTION returns the user to the screen in order to choose the SAVE command or make additional changes.

If TM_SAVE_ACTION is used as a return value, all the requests associated with the SAVE command (except TM_PRE_SAVE and TM_POST_SAVE) are completed, but this processing is not used in the distributed transaction models.

Table 8-6 Slice event processing for CLOSE

Slices	Typical Processing
TM_POST_CLOSE	Processing is identical to that of TM_POST_SAVE described under SAVE.
TM_POST_SAVE1	Described under SAVE, but no save cursor exists.
TM_POST_SAVE2	Described under SAVE.

CONTINUE

Fetches the next set of information from the database

```
int sm_tm_command ("CONTINUE [ tableViewName [ tableViewScope ]
   ]");
```

Arguments

`tableViewName`

The name of a server view in the current transaction. This parameter is case sensitive.

If *`tableViewName`* is specified, the command is applied according to the *`tableViewScope`* parameter. (Since the entire table view tree might not be included, this is known as a partial command, and `sm_tm_command` sets `TM_FULL` to 0.) The specified table view must either be a server view or be the server view to which the desired table view belongs.

If *`tableViewName`* is not specified, the command is applied for each table/server view, starting with the root table view. This is known as a full command, and `sm_tm_command` sets `TM_FULL` to 1.

`tableViewScope`

One of the following parameters, which must be preceded by a table view name.`TV_AND_BELOW` which applies the command to the specified table view and all table views below it on the tree. If no parameter is specified, the transaction manager acts as though `TV_AND_BELOW` was supplied. `BELOW_TV` which applies the command to the table views below the specified table view. `TV_ONLY` which applies the command to the specified table view only. `SV_ONLY` which applies the command only to the table views of the specified server view.

Description

`CONTINUE` (not available in Web applications or three-tier processing) fetches the next set of information from the database. If there are no additional rows, this command has no effect.

`CONTINUE` does not set the transaction mode but requires view or update mode. A partial `CONTINUE` command is also permitted in new mode.

Push buttons and menu selections for the CONTINUE command can choose to set the class property to continue_button. By default, continue_button is active in view and update modes.

If your screen has multiple table views, the transaction manager issues a DBMS CONTINUE for the specified table view and any table views linked to it via server links. This displays the next set of rows for that server view. Then SELECT or VIEW processing is done for any additional child table views.

If your screen has multiple table views and you want to fetch data for only one table view, use FETCH instead of CONTINUE.

Warning about Continuation Files

If the setting of the Fetch Directions property, as discussed in the CONTINUE_DOWN command, permits the CONTINUE_DOWN command to be executed, the data displayed by this command for the specified server view can come from a continuation file. The warnings for CONTINUE_DOWN then apply.

Sequence

Use CONTINUE after SELECT or VIEW which generate a database query and display the first set of query results.

Events

The following requests can be generated by the CONTINUE command to ascertain whether the changes from the previous command have been saved and, if desired, to discard those changes:

- TM_PRE_CLOSE (described under CLOSE)
- TM_CLOSE (described under CLOSE)
- TM_QUERY (described under CLOSE)
- TM_DISCARD (described under CLOSE)
- TM_POST_CLOSE (described under CLOSE)

The following requests can also be generated:

- TM_FETCH (described under FETCH)
- TM_PRE_SELECT (described under SELECT)
- TM_SELECT (described under SELECT)
- TM_POST_SELECT (described under SELECT)
- TM_PRE_VIEW (described under VIEW)

- `TM_VIEW` (described under `VIEW`)
- `TM_POST_VIEW` (described under `VIEW`)

If `TM_VIEW` or `TM_SELECT` for a parent table view returns no data, `TM_CLEAR` requests are generated for all subordinate table views, but not for table views at the same level of the tree. `TM_CLEAR` requests are described under `CLEAR`.

CONTINUE_BOTTOM

Fetches the last set of rows from the file

```
int sm_tm_command ("CONTINUE_BOTTOM [ tableViewName [
    tableViewScope ] ]");
```

Arguments

tableViewName

The name of a table view in the current transaction. This parameter is case sensitive. If *tableViewName* is specified, the command is applied according to the *tableViewScope* parameter. Since the entire table view tree might not be included, this is known as a partial command, and `sm_tm_command` sets `TM_FULL` to 0.

If *tableViewName* is not specified, the command is applied for each table/server view, starting with the root table view. This is known as a full command, and `sm_tm_command` sets `TM_FULL` to 1.

tableViewScope

One of the following parameters, which must be preceded by a table view name.

- `TV_AND_BELOW` which applies the command to the specified table view and all table views below it on the tree. If no parameter is specified, the transaction manager acts as though `TV_AND_BELOW` was supplied.
- `BELOW_TV` which applies the command to the table views below the specified table view.
- `TV_ONLY` which applies the command to the specified table view only.
- `SV_ONLY` which applies the command only to the table views of the specified server view.

Description

`CONTINUE_BOTTOM` (not available in Web applications or three-tier processing) fetches the last set of rows from the file. The availability of this command is de pendent on the setting of the `fetch_directions` property for the server view or screen. If the `fetch_directions` property is set to `PV_CONT_ALWAYS` (Up/ Down-all modes), this command is available in update or view mode. If the `fetch_directions` property is set to `PV_CONT_VIEW_ONLY` (Up/Down-view mode), this command is available only in view mode. Otherwise, an error is generated. For more information on setting the Fetch Directions property, refer to "Scrolling Through the Select Set" on page 36-5 in *Application Development Guide*.

If your screen has multiple table views, the transaction manager issues a `DBMS CONTINUE_BOTTOM` for the specified table view and any table views linked to it via server links. This displays the last set of rows for that server view. Then `SELECT` or `VIEW` processing is done for any additional child table views.

The data displayed with this command is from a continuation file; it is not refetched from the database. Therefore, any updates made to the data in this server view are not displayed immediately to other users. In order to display those updates, the data must be refetched from the database with a `VIEW` or `SELECT` command.

The advantage of using the continuation file is that it prevents having shared locks on data. However, if the `fetch_directions` property is set to `PV_CONT_ALWAYS` (Up/Down-all modes), you are responsible for implementing the necessary locking scheme for concurrent users. For more information on using the Version Column property to implement optimistic locking, refer to "Implementing Optimistic Locking" on page 33-39 in *Application Development Guide*.

If you want to use the database engine's facilities for non-sequential scrolling, you need to add processing for the request events to the engine-specific transaction model.

Push buttons and menu selections for the `CONTINUE_BOTTOM` command can choose to set the class property to `continue_button` which activates the option only in view and update modes or to `continue_view_button` which activates the option only in view mode.

Sequence

Use `CONTINUE_BOTTOM` after `SELECT` or `VIEW` which generate a database query and display the first set of query results or after any other `CONTINUE` command.

Events

Table 8-7 Request event for CONTINUE_BOTTOM

Request	Traversal	Typical Processing
`TM_CONTINUE_BOTTOM`	By table views in the specified server view	Can generate slices: `TM_CONTINUE_BOTTOM` and `TM_SEL_CHECK`. Refer to Table 7.

Table 8-8 Slice event processing for CONTINUE_BOTTOM

Slices	Typical Processing
TM_CONTINUE_BOTTOM	A select cursor must have been set up for the server view encompassing the current table view or nothing more is done. Calls sm_tm_continuation_availability to check if the command is available. If not, an error is issued. On entry, TM_OCC_COUNT specifies the maximum number of occurrences to be fetched. If TM_OCC_COUNT is zero on entry, it means that there is no explicit limit being imposed. The TM_OCC member on entry specifies the first occurrence to be fetched into. TM_OCC_COUNT is then zeroed. (At the end of this request, TM_SEL_CHECK sets it to contain the number of rows fetched.) The data is fetched. TM_SEL_CHECK is pushed onto the event stack to report the number of rows fetched.
TM_SEL_CHECK	If there was an error in earlier processing, give up the select cursor. Otherwise, report the number of rows fetched to TM_OCC_COUNT.

The following requests can be generated by the CONTINUE_BOTTOM command to ascertain if the changes from the previous command have been saved and, if desired, to discard those changes:

- TM_PRE_CLOSE (described under CLOSE)
- TM_CLOSE (described under CLOSE)
- TM_QUERY (described under CLOSE)
- TM_DISCARD (described under CLOSE)
- TM_POST_CLOSE (described under CLOSE)

The following requests can also be generated for any child table views:

- TM_PRE_SELECT (described under SELECT)
- TM_SELECT (described under SELECT)
- TM_POST_SELECT (described under SELECT)
- TM_PRE_VIEW (described under VIEW)

- `TM_VIEW` (described under `VIEW`)
- `TM_POST_VIEW` (described under `VIEW`)

If `TM_VIEW` or `TM_SELECT` for a parent table view returns no data, `TM_CLEAR` requests are generated for all subordinate table views, but not for table views at the same level of the tree. `TM_CLEAR` requests are described under `CLEAR`.

CONTINUE_DOWN

Fetches the next set of rows from the file

```
int sm_tm_command ("CONTINUE_DOWN [ tableViewName [ tableViewScope
   ] ]");
```

Arguments

tableViewName

The name of a table view in the current transaction. This parameter is case sensitive.

If *tableViewName* is specified, the command is applied according to the *tableViewScope* parameter. Since the entire table view tree might not be included, this is known as a partial command, and `sm_tm_command` sets `TM_FULL` to 0.

If *tableViewName* is not specified, the command is applied for each table/server view, starting with the root table view. This is known as a full command, and `sm_tm_command` sets `TM_FULL` to 1.

tableViewScope

One of the following parameters, which must be preceded by a table view name. `TV_AND_BELOW` which applies the command to the specified table view and all table views below it on the tree. If no parameter is specified, the transaction manager acts as though `TV_AND_BELOW` was supplied. `BELOW_TV` which applies the command to the table views below the specified table view. `TV_ONLY` which applies the command to the specified table view only. `SV_ONLY` which applies the command only to the table views of the specified server view.

Description

`CONTINUE_DOWN` (not available in Web applications or three-tier processing) fetches the next set of rows from the file. Note that even though the commands `CONTINUE_DOWN` and `CONTINUE` both display the next set of data, `CONTINUE_DOWN` generates different a request than `CONTINUE`.

The availability of `CONTINUE_DOWN` is dependent on the setting of the `fetch_directions` property for the server view or screen. If the `fetch_directions` property is set to `PV_CONT_ALWAYS` (Up/Down-all modes), this command is available in update or view mode. If the `fetch_directions` property is set to

PV_CONT_VIEW_ONLY (Up/Down-view mode), this command is available only in view mode. Otherwise, an error is generated. For more information on setting the Fetch Directions property, refer to “Scrolling Through the Select Set” on page 36-5 in *Application Development Guide*.

If your screen has multiple table views, the transaction manager issues a DBMS CONTINUE_DOWN for the specified table view and any table views linked to it via server links. This displays the next set of rows for that server view. Then SELECT or VIEW processing is done for any additional child table views.

The data displayed with this command is from a continuation file; it is not refetched from the database. Therefore, any updates made to the data in this server view either by you, or by another user, are not displayed. In order to display those updates, you must again fetch the data from the database with a VIEW or SELECT command.

The advantage of using Panther’s continuation file is that it prevents having shared locks on data. However, if the fetch_directions property is set to PV_CONT_ALWAYS (Up/Down-all modes), you are responsible for implementing the necessary locking scheme for concurrent users. For more information on using the Version Column property to implement optimistic locking, refer to “Implementing Optimistic Locking” on page 33-39 in *Application Development Guide*.

If you want to use the database engine's facilities for non-sequential scrolling, you need to add processing for the request events to the engine-specific transaction model.

Push buttons and menu selections for the CONTINUE_DOWN command can choose to set the class property to continue_button which activates the option only in view and update modes or to continue_view_button which activates the option only in view mode.

Sequence

Use CONTINUE_DOWN after SELECT or VIEW which generate a database query and display the first set of query results or after any other CONTINUE command.

Events

Table 8-9 Request event for CONTINUE_DOWN

Request	Traversal	Typical Processing
TM_CONTINUE_DOWN	The table views in the specified server view	See below

Table 8-10 Request and slice event processing for CONTINUE_DOWN

Slices	Typical Processing
`TM_CONTINUE_DOWN`	A select cursor must have been set up for the server view encompassing the current table view or nothing more is done. Calls `sm_tm_continuation_availability` to check if the command is available. If not, an error is issued. On entry, `TM_OCC_COUNT` specifies the maximum number of occurrences to be fetched. If `TM_OCC_COUNT` is zero on entry, it means that there is no explicit limit being imposed. The `TM_OCC` member on entry specifies the first occurrence to be fetched into. `TM_OCC_COUNT` is then zeroed. (At the end of this request, `TM_SEL_CHECK` sets it to contain the number of rows fetched.) The data is fetched. `TM_SEL_CHECK` is pushed onto the event stack to report the number of rows fetched.
`TM_SEL_CHECK`	If there was an error in earlier processing, give up the select cursor. Otherwise, report the number of rows fetched to `TM_OCC_COUNT`.

The following requests can be generated by the `CONTINUE_DOWN` command to ascertain if the changes from the previous command have been saved and, if desired, to discard those changes:

- `TM_PRE_CLOSE` (described under `CLOSE`)
- `TM_CLOSE` (described under `CLOSE`)
- `TM_QUERY` (described under `CLOSE`)
- `TM_DISCARD` (described under `CLOSE`)
- `TM_POST_CLOSE` (described under `CLOSE`)

The following requests can also be generated for any child table views:

- `TM_PRE_SELECT` (described under `SELECT`)
- `TM_SELECT` (described under `SELECT`)
- `TM_POST_SELECT` (described under `SELECT`)
- `TM_PRE_VIEW` (described under `VIEW`)

- TM_VIEW (described under VIEW)
- TM_POST_VIEW (described under VIEW)

If TM_VIEW or TM_SELECT for a parent table view returns no data, TM_CLEAR requests are generated for all subordinate table views, but not for table views at the same level of the tree. TM_CLEAR requests are described under CLEAR.

CONTINUE_TOP

Fetches the first set of rows from the file

```
int sm_tm_command ("CONTINUE_TOP [tableViewName [tableViewScope]
   ]");
t
```

Arguments

tableViewName

The name of a table view in the current transaction. This parameter is case sensitive. If *tableViewName* is specified, the command is applied according to the *tableViewScope* parameter. Since the entire table view tree might not be included, this is known as a partial command, and `sm_tm_command` sets `TM_FULL` to 0.

If *tableViewName* is not specified, the command is applied for each table/server view, starting with the root table view. This is known as a full command, and `sm_tm_command` sets `TM_FULL` to 1.

tableViewScope

One of the following parameters, which must be preceded by a table view name.

- `TV_AND_BELOW` which applies the command to the specified table view and all table views below it on the tree. If no parameter is specified, the transaction manager acts as though `TV_AND_BELOW` was supplied.
- `BELOW_TV` which applies the command to the table views below the specified table view.
- `TV_ONLY` which applies the command to the specified table view only.
- `SV_ONLY` which applies the command only to the table views of the specified server view.

Description

`CONTINUE_TOP` (not available in Web applications or three-tier processing) fetches the first set of rows from the file. The availability of this command is dependent on the setting of the `fetch_directions` property for the server view or screen. If the `fetch_directions` property is set to `PV_CONT_ALWAYS` (Up/Down-all modes), this command is available in update or view mode. If the `fetch_directions` is set to `PV_CONT_VIEW_ONLY` (Up/Down-view mode), this command is available only in view mode. Otherwise, an error is generated. For more information on setting the Fetch Directions property, refer to “Scrolling Through the Select Set” on page 36-5 in *Application Development Guide*.

If your screen has multiple table views, the transaction manager issues a `DBMS CONTINUE_TOP` for the specified table view and any table views linked to it via server links. This displays the first set of rows for that server view. Then `SELECT` or `VIEW` processing is done for any additional child table views.

The data displayed with this command is from a continuation file: it is not refetched from the database. Therefore, any updates made to the data in this server view either by you, or by another user, are not displayed. In order to display those updates, you must again fetch the data from the database with a `VIEW` or `SELECT` command.

The advantage of using Panther’s continuation file is that it prevents having shared locks on data. However, if the `fetch_directions` property is set to `PV_CONT_ALWAYS` (Up/Down-all modes), you are responsible for implementing the necessary locking scheme for concurrent users. For more information on using the Version Column property to implement optimistic locking, refer to “Implementing Optimistic Locking” on page 33-39 in *Application Development Guide*.

If you want to use the database engine's facilities for non-sequential scrolling, you need to add processing for the request events to the engine-specific transaction model.

Push buttons and menu selections for the `CONTINUE_TOP` command can choose to set the class property to `continue_button` which activates the option only in view and update modes or to `continue_view_button` which activates the option only in view mode.

Sequence

Use `CONTINUE_TOP` after `SELECT` or `VIEW` which generate a database query and display the first set of query results or after any other `CONTINUE` command.

Events

Table 8-11 Request event for the CONTINUE_TOP

Request	Traversal	Typical Processing
`TM_CONTINUE_TOP`	The table views in the specified server view	See below

Table 8-12 Request and slice event processing for CONTINUE_TOP

Slices	Typical Processing
TM_CONTINUE_TOP	A select cursor must have been set up for the server view encompassing the current table view or nothing more is done. Calls sm_tm_continuation_availability to check if the command is available. If not, an error is issued. On entry, TM_OCC_COUNT specifies the maximum number of occurrences to be fetched. If TM_OCC_COUNT is zero on entry, it means that there is no explicit limit being imposed. The TM_OCC member on entry specifies the first occurrence to be fetched into. TM_OCC_COUNT is then zeroed. (At the end of this request, TM_SEL_CHECK sets it to contain the number of rows fetched.) The data is fetched. TM_SEL_CHECK is pushed onto the event stack to report the number of rows fetched.
TM_SEL_CHECK	If there was an error in earlier processing, give up the select cursor. Otherwise, report the number of rows fetched to TM_OCC_COUNT.

The following requests can be generated by the CONTINUE_TOP command to ascertain if the changes from the previous command have been saved and, if desired, to discard those changes:

- TM_PRE_CLOSE (described under CLOSE)
- TM_CLOSE (described under CLOSE)
- TM_QUERY (described under CLOSE)
- TM_DISCARD (described under CLOSE)
- TM_POST_CLOSE (described under CLOSE)

The following requests can also be generated for any child table views:

- TM_PRE_SELECT (described under SELECT)
- TM_SELECT (described under SELECT)
- TM_POST_SELECT (described under SELECT)
- TM_PRE_VIEW (described under VIEW)
- TM_VIEW (described under VIEW)

- TM_POST_VIEW (described under VIEW)

If TM_VIEW or TM_SELECT for a parent table view returns no data, TM_CLEAR requests are generated for all subordinate table views, but not for table views at the same level of the tree. TM_CLEAR requests are described under CLEAR.

CONTINUE_UP

Fetches the previous set of rows from the file

```
int sm_tm_command ("CONTINUE_UP [ tableViewName [ tableViewScope ]
   ]");
```

Arguments

`tableViewName`

The name of a table view in the current transaction. This parameter is case sensitive.

If *`tableViewName`* is specified, the command is applied according to the *`tableViewScope`* parameter. Since the entire table view tree might not be included, this is known as a partial command, and `sm_tm_command` sets `TM_FULL` to 0.

If *`tableViewName`* is not specified, the command is applied for each table/server view, starting with the root table view. This is known as a full command, and `sm_tm_command` sets `TM_FULL` to 1.

`tableViewScope`

One of the following parameters, which must be preceded by a table view name.

- `TV_AND_BELOW` which applies the command to the specified table view and all table views below it on the tree. If no parameter is specified, the transaction manager acts as though `TV_AND_BELOW` was supplied.
- `BELOW_TV` which applies the command to the table views below the specified table view.
- `TV_ONLY` which applies the command to the specified table view only.
- `SV_ONLY` which applies the command only to the table views of the specified server view.

Description

`CONTINUE_UP` (not available in Web applications or three-tier processing) fetches the previous set of rows from the file. The availability of this command is dependent on the setting of the `fetch_directions` property for the server view or screen. If the `fetch_directions` property is set to `PV_CONT_ALWAYS` (Up/ Down-all modes), this command is available in update or view mode. If the `fetch_directions` property is set to `PV_CONT_VIEW_ONLY` (Up/Down-view mode), this command is available only in view mode. Otherwise, an error is generated. For more information on setting the Fetch Directions property, refer to “Scrolling Through the Select Set” on page 36-5 in *Application Development Guide*.

If your screen has multiple table views, the transaction manager issues a `DBMS CONTINUE_UP` for the specified table view and any table views linked to it via server links. This displays the previous set of rows for that server view. Then `SELECT` or `VIEW` processing is done for any additional child table views.

You should be aware that the data displayed with this command is from a continuation file. It is not re-fetched from the database. Therefore, any updates made to the data in this server view either by you, or by another user, are not displayed. In order to display those updates, you must again fetch the data from the database with a `VIEW` or `SELECT` command.

The advantage of using Panther’s continuation file is that it prevents having shared locks on data. However, if the `fetch_directions` property is set to `PV_CONT_ALWAYS` (Up/Down-all modes), you are responsible for implementing the necessary locking scheme for concurrent users. For more information on using the Version Column property to implement optimistic locking, refer to “Implementing Optimistic Locking” on page 33-39 in *Application Development Guide*.

If you want to use the database engine's facilities for non-sequential scrolling, you need to add processing for the request events to the engine-specific transaction model.

Push buttons and menu selections for the `CONTINUE_UP` command can choose to set the class property to `continue_button` which activates the option only in view and update modes or to `continue_view_button` which activates the option only in view mode.

Sequence

Use `CONTINUE_UP` after `SELECT` or `VIEW` which generate a database query and display the first set of query results or after any other `CONTINUE` command.

Events

Table 8-13 Request event for CONTINUE_UP

Request	Traversal	Typical Processing
`TM_CONTINUE_UP`	The table views in the specified server view	See below

Table 8-14 Request and slice event processing for CONTINUE_UP

Slices	Typical Processing
TM_CONTINUE_UP	A select cursor must have been set up for the server view encompassing the current table view or nothing more is done. Calls sm_tm_continuation_availability to check if the command is available. If not, an error is issued. On entry, TM_OCC_COUNT specifies the maximum number of occurrences to be fetched. If TM_OCC_COUNT is zero on entry, it means that there is no explicit limit being imposed. The TM_OCC member on entry specifies the first occurrence to be fetched into. TM_OCC_COUNT is then zeroed. (At the end of this request, TM_SEL_CHECK sets it to contain the number of rows fetched.) The data is fetched. TM_SEL_CHECK is pushed onto the event stack to report the number of rows fetched.
TM_SEL_CHECK	If there was an error in earlier processing, give up the select cursor. Otherwise, report the number of rows fetched to TM_OCC_COUNT.

The following requests can be generated by the CONTINUE_UP command to ascertain if the changes from the previous command have been saved and, if desired, to discard those changes:

- TM_PRE_CLOSE (described under CLOSE)
- TM_CLOSE (described under CLOSE)
- TM_QUERY (described under CLOSE)
- TM_DISCARD (described under CLOSE)
- TM_POST_CLOSE (described under CLOSE)

The following requests can also be generated for any child table views:

- TM_PRE_SELECT (described under SELECT)
- TM_SELECT (described under SELECT)
- TM_POST_SELECT (described under SELECT)
- TM_PRE_VIEW (described under VIEW)

- `TM_VIEW` (described under `VIEW`)
- `TM_POST_VIEW` (described under `VIEW`)

If `TM_VIEW` or `TM_SELECT` for a parent table view returns no data, `TM_CLEAR` requests are generated for all subordinate table views, but not for table views at the same level of the tree. `TM_CLEAR` requests are described under `CLEAR`.

COPY

Duplicates the data on the screen so it can be edited

```
int sm_tm_command ("COPY");
```

Description `COPY` copies the data on the screen for use in the next insertion.

After you select `COPY`, the following steps occur:

1. If you have made changes in the table views on which this command operates in a previous `NEW`, `COPY`, `COPY_FOR_UPDATE`, or `SELECT`, you are prompted to discard your changes. If you choose `OK`, changes are discarded; however, the data remains visible and is treated as though you had just typed it in after a `NEW` command. If you choose Cancel, you return to the screen so you can save your changes.
2. The data currently displayed on the screen is copied.
3. The transaction mode is set to new. By default, this mode clears all the protection bits in updatable table views to reflect that data entry is available in those widgets.
4. Edit the data as much as you wish. Select `SAVE` to insert the data into the database. If you select `SAVE` without changing any data, the transaction manager generates an `INSERT` statement for the duplicate data. Depending on the engine, this could result in a duplicate entry or in an engine error.

Push buttons and menu selections for the `COPY` command can choose to set the class property to `copy_button`. By default, `copy_button` is active in all transaction modes.

Sequence `COPY` is available after you enter new data using `NEW` and `SAVE`. It is also available after `SELECT` or `VIEW` which display data on the screen. Select `SAVE` after you finish your edits.

Events The following requests can be generated by the `COPY` command to ascertain whether the changes from the previous command have been saved and, if desired, to discard those changes:

- `TM_PRE_CLOSE` (described under `CLOSE`)
- `TM_CLOSE` (described under `CLOSE`)
- `TM_QUERY` (described under `CLOSE`)

- TM_POST_CLOSE (described under CLOSE)

Since no TM_DISCARD request is made for the COPY command, the discard flag used in TM_POST_SAVE1 is not set.

Table 8-15 Request events for COPY

Request	Traversal	Typical Processing
TM_PRE_COPY	By table/server view from the specified table view	Do nothing
TM_COPY	By table/server view from the specified table view	Do nothing (sm_bi_initialize is called for the table view by the transaction manager after this request)
TM_POST_COPY	By table/server view from the specified table view	Do nothing

See Also NEW

COPY_FOR_UPDATE

Changes the transaction manager to update mode

```
int sm_tm_command ("COPY_FOR_UPDATE");
```

Description `COPY_FOR_UPDATE` changes the current mode to update. This allows the data currently displayed on the screen to be modified, as though it had been fetched from the database. After you select `COPY_FOR_UPDATE`, the transaction manager initializes before image processing.

If you edit the data and select `SAVE`, the transaction manager generates statements as if the data now on the screen had come from a `SELECT` command. If corresponding data is not in the database, the results might not be what you expect.

Push buttons and menu selections for the `COPY_FOR_UPDATE` command can choose to set the class property to `continue_button` since, by default, `continue_button` is active in view or update modes.

Sequence `COPY_FOR_UPDATE` is available from any mode. Select `SAVE` after you finish your edits.

Events The following requests can be generated by the `COPY_FOR_UPDATE` command to ascertain whether the changes from the previous command have been saved and, if desired, discard those changes:

- `TM_PRE_CLOSE` (described under `CLOSE`)
- `TM_CLOSE` (described under `CLOSE`)
- `TM_QUERY` (described under `CLOSE`)
- `TM_DISCARD` (described under `CLOSE`)
- `TM_POST_CLOSE` (described under `CLOSE`)

Table 8-16 Request events for COPY_FOR_UPDATE

Request	Traversal	Typical Processing
`TM_PRE_COPY_FOR_UPDATE`	By table/server view from the specified table view	Do nothing

Table 8-16 Request events for COPY_FOR_UPDATE *(Continued)*

Request	Traversal	Typical Processing
`TM_COPY_FOR_UPDATE`	By table/server view from the specified table view	Do nothing (`sm_bi_initialize` and `sm_bi_copy` are called for the table view by the transaction manager after this request)
`TM_POST_COPY_FOR_UPDATE`	By table/server view from the specified table view	Do nothing

COPY_FOR_VIEW

Changes the transaction manager to view mode

```
int sm_tm_command ("COPY_FOR_VIEW");
```

Description `COPY_FOR_VIEW` makes view the current mode. After you select `COPY_FOR_VIEW`, the transaction manager disables before image processing. Changes to the data currently on the screen no longer generate updates to the data base with a `SAVE` command.

Push buttons and menu selections for the `COPY_FOR_VIEW` command can choose to set the class property to `continue_button`. By default, `continue_button` is active in view or update modes.

Sequence `COPY_FOR_VIEW` is available after any command.

Events The following requests can be generated by the `COPY_FO+R_VIEW` command to as certain whether the changes from the previous command have been saved and, if desired, discard those changes:

- `TM_PRE_CLOSE` (described under `CLOSE`)
- `TM_CLOSE` (described under `CLOSE`)
- `TM_QUERY` (described under `CLOSE`)
- `TM_DISCARD` (described under `CLOSE`)
- `TM_POST_CLOSE` (described under `CLOSE`)

Table 8-17 Request events for COPY_FOR_VIEW

Request	Traversal	Typical Processing
`TM_PRE_COPY_FOR_VIEW`	By table/server view from the specified table view	Do nothing
`TM_COPY_FOR_VIEW`	By table/server view from the specified table view	Do nothing (`sm_bi_suppress` is called for the table view by the transaction manager after this request)
`TM_POST_COPY_FOR_VIEW`	By table/server view from the specified table view	Do nothing

FETCH

Fetches the next set of data from the database

```
int sm_tm_command ("FETCH [ tableViewName [ { FETCH_SIMPLE |
   FETCH_SPECIAL } ] ]");
```

Arguments

`tableViewName`

The name of a table view in the current transaction. The table view must either be a server view or be the server view to which the desired table view belongs. This parameter is case sensitive. If *`tableViewName`* is not specified, the command is applied to the root table view.

`FETCH_SIMPLE`

Start the fetch with the first occurrence. The number of rows fetched depends on the size of the arrays. This is the default parameter if none is specified, or if no table view name is specified.

`FETCH_SPECIAL`

Allows you to override the occurrence number and the size of the array. To use the `FETCH_SPECIAL` parameter, you must set the value of `TM_OCC` and `TM_OCC_COUNT` with `sm_tm_iset` before calling this command. When `FETCH_SPECIAL` is specified, `TM_OCC` is consulted for the start position and `TM_OCC_COUNT` is consulted for the count.

Description

`FETCH` fetches the next set of rows for the specified table view.

If your screen has multiple table views and you want to fetch data for all of them at the same time, use `CONTINUE` instead of `FETCH`, since fetch is performed only for the specified table view.

Push buttons and menu selections for the `FETCH` command can choose to set the class property to `continue_button`. By default, `continue_button` is active in view and update modes.

Sequence

The `FETCH` command is available after `SELECT` or `VIEW`, both of which generate a database query and display the first set of query results.

Events

Table 8-18 Request events for FETCH

Request	Traversal	Typical Processing
`TM_FETCH`	No tree traversal, since performed only for the specified table view	Slices: `TM_FETCH`, `TM_SEL_CHECK`

Table 8-19 Slice event processing for FETCH

Slices	Typical Processing
`TM_FETCH`	A select cursor must have been set up for the server view encompassing the current table view or nothing more is done. On entry, `TM_OCC_COUNT` specifies the maximum number of occurrences to be fetched. If `TM_OCC_COUNT` is zero on entry, it means that there is no explicit limit being imposed. The `TM_OCC` member on entry specifies the first occurrence to be fetched into. `TM_OCC_COUNT` is then zeroed. (At the end of this event, `TM_SEL_CHECK` sets it to contain the number of rows fetched.) The data is fetched. `TM_SEL_CHECK` is pushed onto the event stack to report the number of rows fetched.
`TM_SEL_CHECK`	If there was an error in earlier processing, give up the select cursor. Otherwise, report the number of rows fetched to `TM_OCC_COUNT`. Give up the select cursor if there are no more rows unless a continuation file is in use.

FINISH

Closes the current transaction manager transaction

```
int sm_tm_command ("FINISH");
```

Description

`FINISH` contains the screen exit processing needed by the transaction manager and is called automatically on screen exit. Therefore, if you use only the default transaction manager transaction on your screen, you do not need to explicitly call this command.

As part of its processing, `FINISH` closes the current transaction, which has been set with the `START` or `CHANGE` commands. In cases where you initiate a transaction by calling the `START` command, you must also call the `FINISH` command to close that transaction before closing the transaction's screen. Note that you might need to call the `CHANGE` command to make the transaction active before closing it with the `FINISH` command.

The `FINISH` command is called after the named screen exit function and after the default screen function. After `FINISH`, the transaction manager data structures for what had been the current transaction no longer exist.

Events

Table 8-20 Request events for FINISH

Request	Traversal	Typical Processing
`TM_FINISH`	By table/server view from the root table view. Done both for event functions and the transaction model.	Slice: `TM_FINISH`

Table 8-21 Slice event processing for FINISH

Slices	Typical Processing
`TM_FINISH`	Give up the save cursor (if it is in use) and the select cursor for the server view encompassing the current table view (if it is in use). For engines where giving up a cursor involves closing the cursor, the return value is `TM_CHECK`. Give up data areas allocated to this transaction, but not areas that are allocated for the transaction model, since there can be other transactions that are still active.

Example

The following procedure closes two additional transactions and then changes back to the main transaction which is assumed to have been active when the procedure is invoked and which is closed on screen exit.

```
proc close_tran
vars main_tran
    main_tran = sm_tm_pinquire(TM_TRAN_NAME)
    call sm_tm_command("CHANGE tran1")
    call sm_tm_command("FINISH")
    call sm_tm_command("CHANGE tran2")
    call sm_tm_command("FINISH")
    call sm_tm_command("CHANGE :main_tran")
return 0
```

Also, refer to the `START` command.

FORCE_CLOSE

Unconditionally discards the changes to the screen

```
int sm_tm_command ("FORCE_CLOSE [ tableViewName [ tableViewScope ]
   ]");
```

Arguments

`tableViewName`

The name of a table view in the current transaction. This parameter is case sensitive.

If *`tableViewName`* is specified, the command is applied according to the *`tableViewScope`* parameter. Since the entire table view tree might not be included, this is known as a partial command, and `sm_tm_command` sets `TM_FULL` to `0`.

If *`tableViewName`* is not specified, the command is applied for each table/server view, starting with the root table view. This is known as a full command, and `sm_tm_command` sets `TM_FULL` to 1.

`tableViewScope`

One of the following parameters, which must be preceded by a table view name.

- `TV_AND_BELOW` which applies the command to the specified table view and all table views below it on the tree. If no parameter is specified, the transaction manager acts as though `TV_AND_BELOW` was supplied.
- `BELOW_TV` which applies the command to the table views below the specified table view.
- `TV_ONLY` which applies the command to the specified table view only.
- `SV_ONLY` which applies the command only to the table views of the specified server view.

Description

`FORCE_CLOSE` unconditionally discards the changes to the screen without a query message. If a table view is not specified, it sets the transaction mode to initial.

Push buttons and menu selections for the FORCE_CLOSE command can choose to set the class property to close_button. By default, close_button is active in all but initial mode.

Sequence

The FORCE_CLOSE command is useful after SELECT, NEW or COPY in order to discard changes.

Events

The following requests can be generated by the FORCE_CLOSE command to discard changes that have been made to the screen.

Table 8-22 Request events for FORCE_CLOSE

Request	Traversal	Typical Processing
TM_PRE_CLOSE	By table/server view from the specified table view	SAVE/CLOSE processing is beginning. Identical processing is performed for TM_PRE_SAVE.
TM_DISCARD	By table/server view from the specified table view	Sets a discard flag, consulted by TM_POST_SAVE1.
TM_POST_CLOSE	By table/server view from the specified table view	Generates event slices: TM_POST_CLOSE, TM_POST_SAVE1, TM_POST_SAVE2 For some engines, the processing in TM_POST_SAVE1 can suggest a change to initial mode at the end of this request.

Table 8-23 Table 22.Slice event processing for the FORCE_CLOSE command.

Table 8-24 Slice event processing for FORCE_CLOSE

Slices	**Typical Processing**
TM_POST_CLOSE	Processing is identical to that of TM_POST_SAVE described under SAVE, but no save cursor exists.
TM_POST_SAVE1	Described under SAVE, but no save cursor exists.
TM_POST_SAVE2	Described under SAVE.

NEW

Prepares screen for data entry

```
int sm_tm_command ("NEW [ tableViewName [ tableViewScope ] ]");
```

Arguments

`tableViewName`

The name of a table view in the current transaction. A table view can only be specified if the mode has already been set to new. This parameter is case sensitive.

If *`tableViewName`* is specified, the command is applied according to the *`tableViewScope`* parameter. Since the entire table view tree might not be included, this is known as a partial command, and `sm_tm_command` sets `TM_FULL` to `0`.

If *`tableViewName`* is not specified, the command is applied for each table/server view, starting with the root table view. This is known as a full command, and `sm_tm_command` sets `TM_FULL` to 1.

`tableViewScope`

One of the following parameters, which must be preceded by a table view name.

- `TV_AND_BELOW` which applies the command to the specified table view and all table views below it on the tree. If no parameter is specified, the transaction manager acts as though `TV_AND_BELOW` was supplied.
- `BELOW_TV` which applies the command to the table views below the specified table view.
- `TV_ONLY` which applies the command to the specified table view only.
- `SV_ONLY` which applies the command only to the table views of the specified server view.

Description

`NEW` clears each field and prepares it for data entry. To insert data successfully, all the fields in a table view that are participating in the `SQL INSERT` statement need to have the same number of occurrences.

After you select `NEW`, the following steps occur:

1. If you have made changes in the table views on which this command operates in a previous NEW, COPY or SELECT, you are prompted to discard your changes. If you choose OK, changes are discarded and fields in the specified table views are cleared. If you choose Cancel, you return to the screen so you can save your changes. You must then select NEW again.
2. The fields are cleared of all previous values.
3. The transaction mode is set to new. By default, this mode clears all the protection bits in updatable table views to reflect that data entry is available in those widgets.
4. Before image processing for the screen is enabled. Any changes made to the screen following this step can then be processed using SAVE.

Push buttons and menu selections for the NEW command can choose to set the class property to new_button. By default, new_button is active in initial and view modes.

Sequence

To save the additions, select SAVE as the next transaction command. To discard the additions, select CLOSE or FORCE_CLOSE.

If you are entering a series of rows, COPY copies the data on a screen so it can then be edited, without having to enter the data again.

Events

The following requests can be generated by the NEW command to ascertain whether the changes from the previous command have been saved and, if desired, discard those changes:

- TM_PRE_CLOSE (described under CLOSE)
- TM_CLOSE (described under CLOSE)
- TM_QUERY (described under CLOSE)
- TM_DISCARD (described under CLOSE)
- TM_POST_CLOSE (described under CLOSE)

Table 8-25 Request events for NEW

Request	Traversal	Typical Processing
TM_PRE_NEW	By table/server view from the specified table view	Do nothing

Table 8-25 Request events for NEW *(Continued)*

Request	Traversal	Typical Processing
`TM_NEW`	By table/server view from the specified table view	Do nothing (`sm_bi_initialize` and `sm_bi_copy` are called for the table view by the transaction manager after this request)
`TM_POST_NEW`	By table/server view from the specified table view	Do nothing

REFRESH

Refreshes the screen in order to update the style and class settings

```
int sm_tm_command ("REFRESH");
```

Description: `REFRESH` reapplies the styles and classes for the current mode.

Events: There are no request events generated by the `REFRESH` command.

RELEASE

Release the cursor used to fetch data in the transaction manager

```
int sm_tm_command ("RELEASE [ tableViewName [ tableViewScope ] ]");
```

Description

`RELEASE` releases the database cursor used to fetch data in the transaction manager. In two-tier applications, if a continuation file is in use, that file will no longer be available. `RELEASE` is only used in special cases; generally, cursor management is part of the `SELECT`, `VIEW` and `CONTINUE` command.

For some database engines, such as SYBASE CT-Lib, the `CLOSE` command does not release the database locks when a `SELECT` command is not followed by a `SAVE` command. In this case, follow the `CLOSE` with the `RELEASE` command which gives up the locks on the database.

Sequence

`RELEASE` has no effect unless it is called after `SELECT` or `VIEW`.

Events

Table 8-26 Request events for RELEASE

Request	Traversal	Typical Processing
`TM_PRE_RELEASE`	By table/server view from the specified table view	Do nothing
`TM_RELEASE`	By table/server view from the specified table view	Slice: `TM_GIVE_UP_SEL_CURSOR`
`TM_POST_RELEASE`	By table/server view from the specified table view	Do nothing

Table 8-27 Slice event processing for RELEASE

Slices	Typical Processing
`TM_GIVE_UP_SEL_CURSOR`	Give up the select cursor. If the cursor name is that of an existing cursor in the database interface, close that cursor (in which case the return value will be `TM_CHECK`).

SAVE

Saves the changes made on the screen to the database

```
int sm_tm_command ("SAVE [ tableViewName [ tableViewScope ] ]");
```

Arguments

`tableViewName`

The name of a table view in the current transaction. This parameter is case sensitive.

If *`tableViewName`* is specified, the command is applied according to the *`tableViewScope`* parameter. Since the entire table view tree might not be included, this is known as a partial command, and `sm_tm_command` sets `TM_FULL` to 0. Refer to the description for more information about partial commands.

If *`tableViewName`* is not specified, the command is applied for each table/server view, starting with the root table view. This is known as a full command, and `sm_tm_command` sets `TM_FULL` to 1.

`tableViewScope`

One of the following parameters, which must be preceded by a table view name.

- `TV_AND_BELOW` which applies the command to the specified table view and all table views below it on the tree. If no parameter is specified, the transaction manager acts as though `TV_AND_BELOW` was supplied.
- `BELOW_TV` which applies the command to the table views below the specified table view.
- `TV_ONLY` which applies the command to the specified table view only.
- `SV_ONLY` which applies the command only to the table views of the specified server view.

Description

`SAVE` compares the current screen to the before image data and generates the necessary statements needed to update the database.

After you select `SAVE`, the following steps occur:

1. The transaction manager checks to see that the mode is not initial or view.

2. For engines requiring it, the transaction model starts a database transaction.
3. The transaction model calls the SQL generator to execute the necessary statements according to the changes that were entered on the screen. Statements can only be generated for updatable table views.

   Note: Some database engines discard the select set when a commit or rollback is performed. For those engines, the distributed transaction models give up the select cursor after a commit or rollback.

4. If an error is encountered, the database transaction is rolled back. If no errors are reported and the `SAVE` command has been specified as a full command, the transaction model commits the database transaction.

Push buttons and menu selections for the `SAVE` command can choose to set the class property to `save_button`. By default, `save_button` is active in new and update modes.

Primary Key Changes

The transaction manager is aware of any primary key changes. If the primary key is updated, the common transaction model deletes the row containing the old value of the primary key and inserts a row contains the new value of the primary key. If the primary key is cleared, the common transaction model deletes the row corresponding to the cleared key fields.

Partial Commands

When a table view is specified for a command, the transaction manager considers it to be a partial command since the command may not apply to the entire tree. In the standard transaction models, the processing for partial `SAVE` commands does not commit the database transaction. Therefore, you must perform an explicit `DBMS COMMIT`. Otherwise, the changes could be rolled back if a later rollback is performed or if the database engine automatically performs a rollback when the connection is closed.

Events

The following request events can be generated by the `SAVE` command:

- `TM_PRE_SAVE`
- `TM_SAVE`
- `TM_DELETE`
- `TM_UPDATE`
- `TM_INSERT`
- `TM_SAVE_ENDING`

- TM_POST_SAVE

Table 8-28 Request events for SAVE

Request	Traversal	Typical Processing
TM_PRE_SAVE	By table/server view from the specified table view	Note that SAVE/CLOSE processing is beginning. (Processing is identical for TM_PRE_CLOSE.) close_or_save_started flag is set to 1, discard flag is set to 0, and reuse_cursor flag is set to 0.
TM_SAVE	By table/server view from the specified table view	Do nothing
TM_DELETE	Modify table views in the order specified in the link properties, with one request in each table view for each row in that table view that has changed	Slices: TM_DELETE, TM_GET_SAVE_CURSOR, TM_DELETE_DECLARE, TM_DELETE_EXEC
TM_UPDATE	Modify table views in the order specified in the link properties, with one request in each table view for each row in that table view that has changed	Slices: TM_UPDATE, TM_GET_SAVE_CURSOR, TM_UPDATE_DECLARE, TM_UPDATE_EXEC
TM_INSERT	Modify table views in the order specified in the link properties, with one request in each table view for each row in that table view that has changed	Slices: TM_INSERT, TM_GET_SAVE_CURSOR, TM_INSERT_DECLARE, TM_INSERT_EXEC
TM_SAVE_ENDING	By table/server view from the specified table view	Call sm_tm_handling to invoke an appropriate function.
TM_POST_SAVE	By table/server view from the specified table view	Slices: TM_POST_SAVE, TM_POST_SAVE1, TM_GIVE_UP_SAVE_CURSOR, TM_POST_SAVE2

Table 8-29 Slice event processing for SAVE

Slices	Typical Processing
`TM_DELETE`	Find out what type of change was made to the current occurrence by checking the return code from `sm_bi_compare`. If the return code is `BI_KEY_CHANGED`, `BI_KEY_NULLED`, or `BI_DELETED`, push the `TM_GET_SAVE_CURSOR`, `TM_DELETE_DECLARE` and `TM_DELETE_EXEC` events onto the stack.
`TM_GET_SAVE_CURSOR`	If a name does not exist for the save cursor, generate it. For some engines, push the `TM_SAVE_BEGIN` event onto the stack.
`TM_SAVE_BEGIN`	For some engines, start a database transaction with `DBMS BEGIN` or a savepoint with `DBMS SAVE`.
`TM_DELETE_DECLARE`	For some engines, give up any select cursor relating to this table view. Call `dm_exec_sql` to declare the save cursor for this deletion, unless the occurrence is part of an array and previously generated SQL is being reused.
`TM_DELETE_EXEC`	Call `dm_exec_sql` to execute the save cursor for this deletion. The return value is `TM_CHECK_ONE_ROW` which tests that only one row was deleted.
`TM_UPDATE`	Find out what type of change was made to the current occurrence by checking the return code from `sm_bi_compare`. If the return code is `BI_UPDATED`, push the `TM_GET_SAVE_CURSOR`, `TM_UPDATE_DECLARE` and `TM_UPDATE_EXEC` events onto the stack.
`TM_UPDATE_DECLARE`	For some engines, give up any select cursor relating to this table view. Call `dm_exec_sql` to declare the save cursor for this update, unless the occurrence is part of an array and previously generated SQL is being reused.
`TM_UPDATE_EXEC`	Call `dm_exec_sql` to execute the save cursor for this update. The return value is `TM_CHECK_ONE_ROW` which tests that only one row was updated.
`TM_INSERT`	Find out what type of change was made to the current occurrence by checking the return code from `sm_bi_compare`. If the return code is `BI_KEY_CHANGED` or `BI_INSERTED`, push the `TM_GET_SAVE_CURSOR`, `TM_INSERT_DECLARE` and `TM_INSERT_EXEC` events on the stack.
`TM_INSERT_DECLARE`	For some engines, give up any select cursor relating to this table view. Call `dm_exec_sql` to declare the cursor for this insertion, unless the occurrence is part of an array and previously generated SQL is being reused.

Table 8-29 Slice event processing for SAVE *(Continued)*

TM_INSERT_EXEC	Call dm_exec_sql to execute the save cursor for this insertion. The return value is TM_CHECK_ONE_ROW which tests that only one row was inserted.
TM_POST_SAVE	If this is the first TM_POST_SAVE event since the last TM_PRE_CLOSE or TM_PRE_SAVE, push the TM_POST_SAVE1 event on the stack. Otherwise, if TM_DISCARD or TM_POST_SAVE1 set the saving worked flag, push the TM_POST_SAVE2 event on the stack.
TM_POST_SAVE1	The existence of a save cursor indicates that SQL statements were executed so the saving worked flag is set to 1. If there is a save cursor and if TM_STATUS is equal to 0 (indicating that the statements executed successfully) and if TM_FULL is equal to 1 (indicating a full SAVE command), the TM_SAVE_COMMIT event is pushed onto the stack. If there is a save cursor and if TM_STATUS is non-zero (indicating that the statements failed), the TM_SAVE_ROLLBACK is pushed onto the stack. For rollbacks, the saving worked flag is reset to 0 since no changes were actually made.
TM_SAVE_ROLLBACK	Push the TM_SAVE_SET_MODE event onto the stack. Do a DBMS ROLLBACK.
TM_SAVE_COMMIT	For full SAVE commands (when TM_FULL is set to 1), do a DBMS COMMIT. If it fails, push the TM_SAVE_ROLLBACK event onto the stack. If it succeeds, push the TM_SAVE_SET_MODE event onto the stack. If the saving worked flag is set to 1, push the TM_GIVE_UP_SAVE_CURSOR and TM_POST_SAVE2 events on the stack and set the discard flag.
TM_SAVE_SET_MODE	Some engine-specific models set TM_VALUE to TM_INITIAL_MODE to indicate that processing should resume in initial mode. (These engines discard the select set when commits and roll backs are performed.)
TM_GIVE_UP_SAVE_CURSOR	Give up the save cursor.
TM_POST_SAVE2	Call sm_bi_initialize. Set TM_OCC to 1 and TM_OCC_COUNT to -1; then, call sm_bi_copy.

SELECT

Fetches data from the database to be updated

```
int sm_tm_command ("SELECT [ tableViewName [ tableViewScope ] ]");
```

Arguments

tableViewName

The name of a server view in the current transaction. This parameter is case sensitive.

If *tableViewName* is specified, the command is applied according to the *tableViewScope* parameter. (Since the entire table view tree might not be included, this is known as a partial command, and `sm_tm_command` sets `TM_FULL` to `0`.) The specified table view must either be a server view or be the server view to which the desired table view belongs.

If *tableViewName* is not specified, the command is applied for each table/server view, starting with the root table view. This is known as a full command, and `sm_tm_command` sets `TM_FULL` to `1`.

tableViewScope

One of the following parameters, which must be preceded by a table view name.

- `TV_AND_BELOW` which applies the command to the specified table view and all table views below it on the tree. If no parameter is specified, the transaction manager acts as though `TV_AND_BELOW` was supplied.
- `BELOW_TV` which applies the command to the table views below the specified table view.
- `TV_ONLY` which applies the command to the specified table view only.
- `SV_ONLY` which applies the command only to the table views of the specified server view.

Description

`SELECT` fetches data from the database so it can be modified. In order to success fully update data or insert new data, all the fields in a server view which are included in the select list need to have the same number of occurrences.

After you choose `SELECT`, the following steps occur:

1. If you have made changes in the table views on which this command operates in a previous NEW, COPY, COPY_FOR_UPDATE, or SELECT, you are prompted to discard your changes. If you choose OK, changes are discarded and fields in the specified table views are cleared. If you choose Cancel, you return to the screen so you can save your changes.

2. The transaction mode is set to update unless a table view is specified and the mode is not initial mode. By default, update mode protects the primary key fields from data entry and sets the display attributes differently for key and non-key fields.

3. If the Count Select property is set to Yes, the transaction manager issues a SELECT statement using COUNT(*) to find the number of rows in the select set. If this number exceeds the amount set in the Count Threshold property, a message box offers the user the choice of discontinuing data selection.

4. The screen displays the first set of data for all linked table views. When you choose SELECT, the standard transaction models have the SQL generator execute a SELECT statement for the database table named in the root table view and any table views connected to it via a server link. Then, recursively, SELECT statements are issued for the child table views having sequential links, and any table views connected to those child table views by server links.

5. The before image, or snapshot, of the screen is taken for the screen`s updatable table views. An updatable table view must have its primary key fields on screen. Any changes made to the screen following this step can then be processed using a SAVE command.

Push buttons and menu selections for the SELECT command can choose to set the class property to view_button. By default, view_button is active in initial or view modes.

Using QBE

If you want to select a specific record or group of records, set the widget's use_in_where property to PV_YES and the type of operator (where_operator) to be used in the WHERE clause. Then, in the transaction manager, choose CLEAR to clear the fields, enter a value in your query field, and then choose SELECT. The screen displays the specified information.

Using the Count Select Property

If the server view's Count Select and Count Warning properties are set to Yes, the application will warn users about large select sets; however, the SELECT statement is performed twice, and the tables must remain locked for the result to be the same for both statements.

Sequence

To save the changes or additions made to the selected data, choose `SAVE` as the next transaction command.

To display the next row of information, choose `CONTINUE` as the next transaction command. If you have updated the data on the screen, you are prompted to discard your changes. If you choose `OK`, changes are discarded. If you choose `Cancel`, you return to the screen so you can save your changes.

To discard any changes you have made to the screen, choose `CLOSE` or `FORCE_CLOSE`. For some database engines, such as SYBASE CT-Lib, the `CLOSE` command does not release the database locks when a `SELECT` command is not followed by a `SAVE` command. In this case, follow the `CLOSE` with the `RELEASE` command which gives up the locks on the database.

Events

The following request events can be generated by the `SELECT` command to ascertain whether the changes from the previous command have been saved and, if desired, discard those changes:

- `TM_PRE_CLOSE` (described under `CLOSE`)
- `TM_CLOSE` (described under `CLOSE`)
- `TM_QUERY` (described under `CLOSE`)
- `TM_DISCARD` (described under `CLOSE`)
- `TM_POST_CLOSE` (described under `CLOSE`)

The `SELECT` command generates `TM_CLEAR` requests if `TM_SELECT` for a parent table view returns no data. In that case, `TM_CLEAR` is generated for all subordinate table views, but not for table views at the same level of the tree. `TM_CLEAR` requests are described under `CLEAR`.

Table 8-30 Request events for SELECT

Request	Traversal	Typical Processing
`TM_PRE_SELECT`	By table/server view from the specified table view	Do nothing

Table 8-30 Request events for SELECT *(Continued)*

Request	Traversal	Typical Processing
`TM_SELECT`	By table/server view from the specified table view	Slices: `TM_SELECT`, `TM_GET_SEL_CURSOR`, `TM_PREPARE_CONTINUE`, `TM_SET_SEL_COUNT_FLAG`, `TM_SEL_GEN`, `TM_SEL_BUILD_PERFORM`, `TM_SEL_COUNT_CHECK`, `TM_CLEAR_SEL_COUNT_FLAG`, `TM_SEL_CHECK` (`sm_bi_initialize` is called for the table view by the transaction manager after this request. If rows were fetched, `sm_bi_copy` is also called.)
`TM_POST_SELECT`	By table/server view from the specified table view	Do nothing

Table 8-31 Table 29. the SELECT command.

Table 8-32 Slice event processing for SELECT

Slices	Typical Processing
`TM_SELECT`	`TM_OCC_COUNT` is zeroed. At the end of processing for this request, it contains the number of rows fetched (set, if at all, by `TM_SEL_CHECK`). If the table view is the first one in the current server view: -Push the `TM_GET_SEL_CURSOR` (only if there is no select cursor already) and the `TM_PREPARE_CONTINUE` events on the stack. -If `use_select_count` is set to 1, push `TM_SET_SEL_COUNT_FLAG`, `TM_SEL_GEN`, `TM_SEL_BUILD_PERFORM`, `TM_SEL_COUNT_CHECK`, and `TM_CLEAR_SEL_COUNT_FLAG` events on the stack. -Push the `TM_SEL_GEN`, `TM_SEL_BUILD_PERFORM`, and `TM_SEL_CHECK` events on the stack. If the table view is not the first one in the server view, nothing more is done for this request, and the number of rows fetched for this request is correctly reported as zero.

Table 8-32 Slice event processing for SELECT

Slices	Typical Processing
TM_GET_SEL_CURSOR	If a name does not exist for the Panther select cursor, generate it. (Depending on the engine, a Panther cursor may or may not correspond to a database cursor.)
TM_PREPARE_CONTINUE	If the select cursor does not already exist, a dummy DECLARE CURSOR command is issued. If sm_tm_continuation_validity reports that continuation file commands (like CONTINUE_TOP) are valid, DBMS STORE FILE is issued. If the function reports that those commands are invalid, DBMS STORE is issued.
TM_SET_SEL_COUNT_FLAG	If count_select is set to Yes, set TM_SV_SEL_COUNT to 1.
TM_SEL_GEN	Generate data structures with dm_gen_sql_info that are used in the TM_SEL_BUILD_PERFORM slice to build the SQL statements. If TM_SV_SEL_COUNT is 1, modify the structure to use count(*) and alias the result into the server view's count_result property.
TM_SEL_BUILD_PERFORM	Build, and then, if there was no error in building, perform SELECT (and other DBMS commands) with dm_exec_sql. Free the select information.
TM_SEL_COUNT_CHECK	If count_result > count_threshold, check count_warning to see if a Y/N message box should be displayed. If count_result is 0, push the TM_GIVE_UP_SEL_CURSOR event and call sm_tm_clear.
TM_CLEAR_SEL_COUNT_FLAG	Set TM_SV_SEL_COUNT to 0.
TM_SEL_CHECK	If there was an error in earlier processing, give up the select cursor. Otherwise, report the number of rows fetched to TM_OCC_COUNT. Give up the select cursor if there are no more rows un less a continuation file is in use. (On engines where this means that the cursor is closed, the return code is TM_CHECK. Otherwise, the return code is TM_OK.)

If TM_SELECT for a parent table view returns no data, TM_CLEAR requests are generated for all subordinate table views, but not for table views at the same level of the tree. TM_CLEAR requests are described under CLEAR.

START

Initializes a new transaction tree

```
int sm_tm_command ("START transaction-name [ tableViewName ]");
```

Arguments

`transaction-name`
: The name of a transaction to be used for this screen.

`tableViewName`
: The name of a table view in the current transaction. This parameter is case sensitive. If *`tableViewName`* is not specified, the command is applied for each table/server view, starting with the root table view. This is known as a full command, and `sm_tm_command` sets `TM_FULL` to 1.

Description

`START` initiates a transaction manager transaction and makes it the current transaction. The mode is set to initial. Since this command is called automatically on screen entry whenever a root table view can be determined, you only call this command explicitly when you are using more than one transaction manager transaction on the same screen.

Any transaction begun with an explicit call to the `START` command must also be closed by an explicit call to the `FINISH` command. If necessary, use the `CHANGE` command to make the transaction active before closing it with the `FINISH` command.

A transaction manager transaction must be in progress in order to call other transaction manager commands.

Note: In screen entry events, the unnamed JPL procedure is called before the `START` command. Therefore, transaction manager commands cannot be invoked in the unnamed procedure. After the `START` command is called, Panther then calls the default screen function and the named screen entry function.

Sequence

Once a transaction has been initiated with the `START` command, you can make it the current transaction using the `CHANGE` command.

Events

Table 8-33 Request events for START

Request	Traversal	Typical Processing
`TM_START`	By table/server view from the specified table view. Done both for event functions and the transaction model.	Do nothing

Example

The following example illustrates the use of the `START`, `CHANGE`, and `FINISH` commands in order to execute transaction manager commands on an unlinked table view.

In this example, `pricecats` is the unlinked table view. The procedure `start_new_tran` first finds the name of the current transaction, starts a new transaction for the `pricecats` table view, and then changes to that transaction in order to execute a `VIEW` command. The procedure `change_to_main` changes back to the original transaction in order to execute transaction manager commands on those table views. The procedure `change_to_new_tran` changes to the new transaction in order to execute transaction manager commands on the `pricecats` table view. The procedure exit is set as the value of the screen's `exit_function` property.

```
# JPL Procedures:
   vars main_tran(31)
   proc start_new_tran
   main_tran=sm_tm_pinquire(TM_TRAN_NAME)
   call sm_tm_command("START price_tran pricecats")
   call sm_tm_command("CHANGE price_tran")
   call sm_tm_command("VIEW")
   return

proc change_to_main
   call sm_tm_command("CHANGE :main_tran")
   return

proc change_to_new_tran
   call sm_tm_command("CHANGE price_tran")
   return

# Screen exit function property invokes following procedure.

proc exit(screen, flags)
   if (flags & K_EXIT)
   {
   call sm_tm_command("CHANGE price_tran")
      call sm_tm_command("FINISH")
      call sm_tm_command("CHANGE :main_tran")
      call sm_tm_command("FINISH")
   }
   return
```

VIEW

Fetches data from the database for display purposes

```
int sm_tm_command ("VIEW [ tableViewName [ tableViewScope ] ]");
```

Arguments

tableViewName

The name of a server view in the current transaction. This parameter is case sensitive.

If *tableViewName* is specified, the command is applied according to the *tableViewScope* parameter. (Since the entire table view tree might not be included, this is known as a partial command, and sm_tm_command sets TM_FULL to 0.) The specified table view must either be a server view or be the server view to which the desired table view belongs.

If *tableViewName* is not specified, the command is applied for each table/server view, starting with the root table view. This is known as a full command, and sm_tm_command sets TM_FULL to 1.

tableViewScope

One of the following parameters, which must be preceded by a table view name.

- TV_AND_BELOW which applies the command to the specified table view and all table views below it on the tree. If no parameter is specified, the transaction manager acts as though TV_AND_BELOW was supplied.
- BELOW_TV which applies the command to the table views below the specified table view.
- TV_ONLY which applies the command to the specified table view only.
- SV_ONLY which applies the command only to the table views of the specified server view.

Description

VIEW fetches data from the database for display purposes only.

When VIEW is selected the following steps occur:

1. If you have made changes in the table views on which this command operates in a previous NEW, COPY, COPY_FOR_UPDATE, or SELECT, you are prompted to discard your changes. If you choose OK, changes are discarded and fields in the specified table views are cleared. If you choose Cancel, you return to the screen so you can save your changes.

2. The transaction mode is set to view unless a table view is specified. By default, view mode protects all fields from data entry.

3. If the Count Select property is set to Yes, the transaction manager issues a SELECT statement using COUNT(*) to find the number of rows in the select set. If this number exceeds the amount set in the Count Threshold property, a message box offers the user the choice of discontinuing data selection.

4. The screen displays the first set of data for all linked table views. When you choose VIEW, the common transaction model has the SQL generator execute a SELECT statement for the database table named in the root table view and any table views connected to it via a server link. Then, recursively, SELECT statements are issued for the child table views having sequential links, and any table views connected to those child table views by server links. If the query does not return any rows for the first server view, no data are displayed for the remaining server views. (A query which successfully returns rows sets TM_OCC_COUNT as part of the TM_SEL_CHECK slice. When TM_OCC_COUNT is greater than 0, the query is generated for the next server view.)

Push buttons and menu selections for the VIEW command can choose to set the class property to view_button. By default, view_button is active in initial or view modes.

Using QBE

If you want to select a specific record or group of records, you need to set the use_in_where property to PV_YES and set the type of operator (where_operator property) to be used in the WHERE clause. Then, in the transaction manager, choose CLEAR to clear the fields, enter a value in your query field, and then choose VIEW. The screen displays the specified information.

Using the Count Select Property

If the server view's Count Select and Count Warning properties are set to Yes, the application will warn users about large select sets; however, the SELECT statement is performed twice, and the tables must remain locked for the result to be the same for both statements.

Sequence

To display additional data in two-tier processing, choose any CONTINUE command.

Events The following request events can be generated by the VIEW command to ascertain whether the changes from the previous command have been saved and, if desired, discard those changes:

- TM_PRE_CLOSE (described under CLOSE)
- TM_CLOSE (described under CLOSE)
- TM_QUERY (described under CLOSE)
- TM_DISCARD (described under CLOSE)
- TM_POST_CLOSE (described under CLOSE)

The VIEW command generates TM_CLEAR requests if TM_VIEW for a parent table view returns no data. In that case, TM_CLEAR is generated for all subordinate table views, but not for table views at the same level of the tree. TM_CLEAR requests are described under CLEAR.

Table 8-34 Transaction manager command.

Table 8-35 Request events for VIEW

Request	Traversal	Typical Processing
TM_PRE_VIEW	By table/server view from the specified table view	Do nothing
TM_VIEW	By table/server view from the specified table view	Slices: TM_VIEW, TM_GET_SEL_CURSOR, TM_PREPARE_CONTINUE, TM_SET_SEL_COUNT_FLAG, TM_SEL_GEN, TM_SEL_BUILD_PERFORM, TM_SEL_COUNT_CHECK, TM_CLEAR_SEL_COUNT_FLAG, TM_SEL_CHECK (sm_bi_suppress is called for the table view by the transaction manager after this request.)
TM_POST_VIEW	By table/server view from the specified table view	Do nothing

Table 8-36 Slice event processing for VIEW

Slices	Typical Processing
TM_VIEW	TM_OCC_COUNT is zeroed. At the end of processing for this request, it contains the number of rows fetched (set, if at all, by TM_SEL_CHECK). If the table view is the first one in the current server view: -Push the TM_GET_SEL_CURSOR (only if there is no select cursor already) and the TM_PREPARE_CONTINUE events on the stack. -If use_select_count is set to 1, push TM_SET_SEL_COUNT_FLAG, TM_SEL_GEN, TM_SEL_BUILD_PERFORM, TM_SEL_COUNT_CHECK, and TM_CLEAR_SEL_COUNT_FLAG events on the stack. -Push the TM_SEL_GEN, TM_SEL_BUILD_PERFORM, and TM_SEL_CHECK events on the stack. If the table view is not the first one in the server view, nothing more is done for this request, and the number of rows fetched for this request is correctly reported as zero.
TM_GET_SEL_CURSOR	If a name does not exist for the Panther select cursor, generate it. (Depending on the engine, a Panther cursor may or may not correspond to a database cursor.)
TM_PREPARE_CONTINUE	If the select cursor does not already exist, a dummy DBMS DECLARE CURSOR command is issued. If sm_tm_continuation_validity reports that continuation file commands (like CONTINUE_TOP) are valid, DBMS STORE FILE is issued. If the function reports that those commands are invalid, DBMS STORE is issued.
TM_SET_SEL_COUNT_FLAG	If count_select is set to Yes, set TM_SV_SEL_COUNT to 1.
TM_SEL_GEN	Generate data structures with dm_gen_sql_info that are used in the TM_SEL_BUILD_PERFORM slice to build the SQL statements. If TM_SV_SEL_COUNT is 1, modify the structure to use count(*) and alias the result into the server view's count_result property.
TM_SEL_BUILD_PERFORM	Build, and then, if there was no error in building, perform SELECT (and other DBMS commands) with dm_exec_sql. Free the select information.

Table 8-36 Slice event processing for VIEW *(Continued)*

TM_SEL_COUNT_CHECK	If count_result > count_threshold, check count_warning to see if a Y/N message box should be displayed. If count_result is 0, push the TM_GIVE_UP_SEL_CURSOR event and call sm_tm_clear.
TM_CLEAR_SEL_COUNT_FLAG	Set TM_SV_SEL_COUNT to 0.
TM_SEL_CHECK	If there was an error in earlier processing, give up the select cursor. Otherwise, report the number of rows fetched to TM_OCC_COUNT. Give up the select cursor if there are no more rows un less a continuation file is in use.

If TM_VIEW for a parent table view returns no data, TM_CLEAR requests are generated for all subordinate table views, but not for table views at the same level of the tree. TM_CLEAR requests are described under CLEAR.

WALK_DELETE

Traverses the transaction tree in delete order

```
int sm_tm_command ("WALK_DELETE [ tableViewName [ tableViewScope ]
    ]");
```

Arguments

`tableViewName`

The name of a table view in the current transaction. This parameter is case sensitive.

If *`tableViewName`* is specified, the command is applied according to the *`tableViewScope`* parameter. Since the entire table view tree might not be included, this is known as a partial command, and `sm_tm_command` sets `TM_FULL` to `0`.

If *`tableViewName`* is not specified, the command is applied for each table/server view, starting with the root table view. This is known as a full command, and `sm_tm_command` sets `TM_FULL` to `1`.

`tableViewScope`

One of the following parameters, which must be preceded by a table view name.

- `TV_AND_BELOW` which applies the command to the specified table view and all table views below it on the tree. If no parameter is specified, the transaction manager acts as though `TV_AND_BELOW` was supplied.
- `BELOW_TV` which applies the command to the table views below the specified table view.
- `TV_ONLY` which applies the command to the specified table view only.
- `SV_ONLY` which applies the command only to the table views of the specified server view.

Description

`WALK_DELETE` performs a traversal of the transaction tree, using the Delete Order property in the link widgets to determine the traversal order. If a table view's transaction event function contains processing for the `TM_WALK_DELETE` request event, it is executed.

Events A `TM_WALK_DELETE` request event is generated by the `WALK_DELETE` command, but no processing is associated with this event in the transaction models.

WALK_INSERT

Traverses the transaction tree in insert order

```
int sm_tm_command ("WALK_INSERT [ tableViewName [ tableViewScope ]
    ]");
```

Arguments

`tableViewName`

The name of a table view in the current transaction. This parameter is case sensitive.

If *`tableViewName`* is specified, the command is applied according to the *`tableViewScope`* parameter. Since the entire table view tree might not be included, this is known as a partial command, and `sm_tm_command` sets `TM_FULL` to `0`.

If *`tableViewName`* is not specified, the command is applied for each table/server view, starting with the root table view. This is known as a full command, and `sm_tm_command` sets `TM_FULL` to `1`.

`tableViewScope`

One of the following parameters, which must be preceded by a table view name.

- `TV_AND_BELOW` which applies the command to the specified table view and all table views below it on the tree. If no parameter is specified, the transaction manager acts as though `TV_AND_BELOW` was supplied.
- `BELOW_TV` which applies the command to the table views below the specified table view.
- `TV_ONLY` which applies the command to the specified table view only.
- `SV_ONLY` which applies the command only to the table views of the specified server view.

Description

`WALK_INSERT` performs a traversal of the transaction tree, using the Insert Order property in the link widgets to determine the traversal order. If a table view's transaction event function contains processing for the `TM_WALK_INSERT` request event, it is executed.

Events A `TM_WALK_INSERT` request event is generated by the `WALK_INSERT` command, but no processing is associated with this event in the transaction models.

WALK_SELECT

Traverses the transaction tree in select order

```
int sm_tm_command ("WALK_SELECT [ tableViewName [ tableViewScope ]
    ]");
```

Arguments

`tableViewName`

The name of a table view in the current transaction. This parameter is case sensitive.

If *`tableViewName`* is specified, the command is applied according to the *`tableViewScope`* parameter. Since the entire table view tree might not be included, this is known as a partial command, and `sm_tm_command` sets `TM_FULL` to 0.

If *`tableViewName`* is not specified, the command is applied for each table/server view, starting with the root table view. This is known as a full command, and `sm_tm_command` sets `TM_FULL` to 1.

`tableViewScope`

One of the following parameters, which must be preceded by a table view name.

- `TV_AND_BELOW` which applies the command to the specified table view and all table views below it on the tree. If no parameter is specified, the transaction manager acts as though `TV_AND_BELOW` was supplied.
- `BELOW_TV` which applies the command to the table views below the specified table view.
- `TV_ONLY` which applies the command to the specified table view only.
- `SV_ONLY` which applies the command only to the table views of the specified server view.

Description

`WALK_SELECT` performs a traversal of the transaction tree, starting with the root table/server view, unless another table/server view is specified. If a table view's transaction event function contains processing for the `TM_WALK_SELECT` request event, it is executed.

Events A `TM_WALK_SELECT` request event is generated by the `WALK_SELECT` command, but no processing is associated with this event in the transaction models.

WALK_UPDATE

Traverses the transaction tree in update order

```
int sm_tm_command ("WALK_UPDATE [tableViewName [tableViewScope]
   ]");
```

Arguments

`tableViewName`

The name of a table view in the current transaction. This parameter is case sensitive.

If *`tableViewName`* is specified, the command is applied according to the *`tableViewScope`* parameter. Since the entire table view tree might not be included, this is known as a partial command, and `sm_tm_command` sets `TM_FULL` to 0.

If *`tableViewName`* is not specified, the command is applied for each table/server view, starting with the root table view. This is known as a full command, and sm_tm_command sets `TM_FULL` to 1.

`tableViewScope`

One of the following parameters, which must be preceded by a table view name.

- `TV_AND_BELOW` which applies the command to the specified table view and all table views below it on the tree. If no parameter is specified, the transaction manager acts as though `TV_AND_BELOW` was supplied.
- `BELOW_TV` which applies the command to the table views below the specified table view.
- `TV_ONLY` which applies the command to the specified table view only.
- `SV_ONLY` which applies the command only to the table views of the specified server view.

Description

`WALK_UPDATE` performs a traversal of the transaction tree, using the Update Order property in the link widgets to determine the traversal order. If a table view's transaction event function contains processing for the `TM_WALK_UPDATE` request event, it is executed.

Events A `TM_WALK_UPDATE` request event is generated by the `WALK_UPDATE` command, but no processing is associated with this event in the transaction models.

CHAPTER

9 Transaction Model Events

The transaction manager accesses two layers of transaction models: the common model and a database-specific model. The common transaction model contains the functionality common to all of the database engines; the database-specific model contains processing necessary for a specific database engine.

The source code for the database-specific transaction models is provided in the distribution and can be modified to make global changes in transaction manager functionality. The common model should not be modified; however, the source code is available for reference.

- Common Transaction Model (page 9-2)
- Database-Specific Transaction Models (page 9-13)

For more information, refer to *Application Development Guide*—Chapter 35, “Generating Transaction Manager Events,” for how the transaction manager uses the transaction model and to Chapter 32, “Writing Transaction Event Functions,” for an explanation of the return codes.

Common Transaction Model

Table 9-1lists the events generated for each transaction manager command in the common model. The events are listed in the table in the order in which they are processed. Error and diagnostic events are indicated for the events which might generate them, although many of the error events shown are unlikely to be encountered. The error checking events are done after the events that give rise to them, and before any other event processing.

The transaction manager command documentation contains a description of the processing for each event; refer to Chapter 8, "Transaction Manager Commands."

Reading the Event Table

For compactness, whenever it is possible, the lower level events are shown on the same line as the higher level events that give rise to them. Thus, the entry for the FETCH command compresses the information about six events into the following two lines:

Command	Request	Slice	Slice	Error	Error
FETCH	TM_FETCH	.	.	E	F
.	.	TM_SEL_CHECK	TM_SEL_CHECK	E	F

- The FETCH command generates only one request, TM_FETCH.
- TM_FETCH in its own right can cause a TM_TEST_ERROR event to be generated by the transaction manager (by returning TM_CHECK).
- The TM_TEST_ERROR event can cause a TM_NOTE_FAILURE event to generated by transaction manager (by returning TM_FAILURE).
- TM_FETCH also has a slice that it generates, TM_SEL_CHECK.

- TM_SEL_CHECK can cause a TM_TEST_ERROR event to be generated by the transaction manager (by returning TM_CHECK).
- The TM_TEST_ERROR event can cause a TM_NOTE_FAILURE event to be generated by transaction manager (by returning TM_FAILURE).

For a description of the error events, refer to page 9-12, "Error and Diagnostic Events."

Table 9-1 Transaction manager request events

Command	Request with Corresponding Error Events	Slices with Corresponding Error Events
CHANGE		
CLEAR	PRE_CLOSE	
	CLOSE	
	QUERY	
	DISCARD	
	POST_CLOSE	POST_SAVE1 ■ POST_SAVE2 - F POST_SAVE2 - F
	PRE_CLEAR	
	CLEAR	
	POST_CLEAR	

E = TM_TEST_ERROR, O = TM_TEST_ONE_ROW, F = TM_NOTE_FAILURE;
*All request and slice events have TM_ prefix.
**Slice events generated based on server view property specifications.

Table 9-1 Transaction manager request events *(Continued)*

Command	Request with Corresponding Error Events	Slices with Corresponding Error Events
`CLOSE`	`PRE_CLOSE`	
	`CLOSE`	
	`QUERY`	
	`DISCARD`	

	`POST_CLOSE`	`POST_SAVE1`
		■ `POST_SAVE2 - F`
		`POST_SAVE2 - F`

`E = TM_TEST_ERROR,  O = TM_TEST_ONE_ROW,  F = TM_NOTE_FAILURE`;
*All request and slice events have `TM_` prefix.
**Slice events generated based on server view property specifications.

Table 9-1 Transaction manager request events *(Continued)*

Command	Request with Corresponding Error Events	Slices with Corresponding Error Events
CONTINUE	FETCH - E /F	SEL_CHECK - E/F ■ GIVE_UP_SEL_CURSOR - E/F
	PRE_SELECT	
	SELECT	GET_SEL_CURSOR - F SEL_GEN - F ■ GIVE_UP_SEL_CURSOR - E/F SEL_BUILD_PERFORM - E,F/F SEL_CHECK - E/F ■ GIVE_UP_SEL_CURSOR - E/F
	CLEAR	
	POST_SELECT	
	PRE_VIEW	
	VIEW	GET_SEL_CURSOR - F SEL_GEN - F ■ GIVE_UP_SEL_CURSOR - E/F SEL_BUILD_PERFORM - E,F/F SEL_CHECK - E/F ■ GIVE_UP_SEL_CURSOR - E/F
	CLEAR	
	POST_VIEW	

E = TM_TEST_ERROR, O = TM_TEST_ONE_ROW, F = TM_NOTE_FAILURE;
*All request and slice events have TM_ prefix.
**Slice events generated based on server view property specifications.

Table 9-1 Transaction manager request events *(Continued)*

Command	Request with Corresponding Error Events	Slices with Corresponding Error Events
CONTINUE_BOTTOM	CONTINUE_BOTTOM - E,F/F	SEL_CHECK GIVE_UP_SEL_CURSOR
	PRE_SELECT	
	SELECT	GET_SEL_CURSOR - F SEL_GEN - F ■ GIVE_UP_SEL_CURSOR - E/F SEL_BUILD_PERFORM - E,F/F SEL_CHECK - E/F ■ GIVE_UP_SEL_CURSOR - E/F
	CLEAR	
	POST_SELECT	
	PRE_VIEW	
	VIEW	GET_SEL_CURSOR - F SEL_GEN - F ■ GIVE_UP_SEL_CURSOR - E/F SEL_BUILD_PERFORM - E,F/F SEL_CHECK - E/F ■ GIVE_UP_SEL_CURSOR - E/F
	CLEAR	
	POST_VIEW	
CONTINUE_DOWN	CONTINUE_DOWN - E,F/F	refer to CONTINUE_BOTTOM for the remainder of the requests and slices
CONTINUE_TOP	CONTINUE_TOP - E,F/F	refer to CONTINUE_BOTTOM for the remainder of the requests and slices

E = TM_TEST_ERROR, O = TM_TEST_ONE_ROW, F = TM_NOTE_FAILURE;
*All request and slice events have TM_ prefix.
**Slice events generated based on server view property specifications.

Table 9-1 Transaction manager request events *(Continued)*

Command	Request with Corresponding Error Events	Slices with Corresponding Error Events
CONTINUE_UP	CONTINUE_UP - E,F/F	refer to CONTINUE_BOTTOM for the remainder of the requests and slices
COPY	PRE_CLOSE	
	CLOSE	
	QUERY	
	DISCARD	
	POST_CLOSE	POST_SAVE1 ■ POST_SAVE2 - F POST_SAVE2 - F
	PRE_COPY	
	COPY	
	POST_COPY	
COPY_FOR_UPDATE	PRE_CLOSE	
	CLOSE	
	QUERY	
	DISCARD	
	POST_CLOSE	POST_SAVE1 ■ POST_SAVE2 - F POST_SAVE2 - F
	PRE_COPY_FOR_UPDATE	
	COPY_FOR_UPDATE	
	POST_COPY_FOR_UPDATE	

E = TM_TEST_ERROR, O = TM_TEST_ONE_ROW, F = TM_NOTE_FAILURE;
*All request and slice events have TM_ prefix.
**Slice events generated based on server view property specifications.

Table 9-1 Transaction manager request events *(Continued)*

Command	Request with Corresponding Error Events	Slices with Corresponding Error Events
COPY_FOR_VIEW	PRE_CLOSE	
	CLOSE	
	QUERY	
	POST_CLOSE	POST_SAVE1 ■ POST_SAVE2 - F POST_SAVE2 - F
	PRE_COPY_FOR_VIEW	
	COPY_FOR_VIEW	
	POST_COPY_FOR_VIEW	
FETCH	FETCH - E/F	SEL_CHECK - E/F ■ GIVE_UP_SEL_CURSOR - E/F
FINISH	FINISH - E/F	GIVE_UP_SAVE - E/F GIVE_UP_SEL_CURSOR - E/F
FORCE_CLOSE	PRE_CLOSE	
	DISCARD	
	POST_CLOSE	POST_SAVE1 ■ POST_SAVE2 - F POST_SAVE2 - F

E = TM_TEST_ERROR, O = TM_TEST_ONE_ROW, F = TM_NOTE_FAILURE;
*All request and slice events have TM_ prefix.
**Slice events generated based on server view property specifications.

Table 9-1 Transaction manager request events *(Continued)*

Command	Request with Corresponding Error Events	Slices with Corresponding Error Events
NEW	PRE_CLOSE	
	CLOSE	
	QUERY	
	DISCARD	
	POST_CLOSE	POST_SAVE1 ■ POST_SAVE2 - F POST_SAVE2 - F
	PRE_NEW	
	NEW	
	POST_NEW	
REFRESH		
RELEASE	PRE_RELEASE	
	RELEASE	GIVE_UP_SEL_CURSOR - E/F
	POST_RELEASE	

E = TM_TEST_ERROR, O = TM_TEST_ONE_ROW, F = TM_NOTE_FAILURE;
*All request and slice events have TM_ prefix.
**Slice events generated based on server view property specifications.

Table 9-1 Transaction manager request events *(Continued)*

Command	Request with Corresponding Error Events	Slices with Corresponding Error Events
SAVE	PRE_SAVE	
	SAVE	
	DELETE	GET_SAVE_CURSOR - F ■ SAVE_BEGIN DELETE_DECLARE - E,F/F DELETE_EXEC - O.F/F
	UPDATE	GET_SAVE_CURSOR - F ■ SAVE_BEGIN UPDATE_DECLARE - E,F/F UPDATE_EXEC - O.F/F
	INSERT	GET_SAVE_CURSOR - F ■ SAVE_BEGIN INSERT_DECLARE - E,F/F INSERT_EXEC - O.F/F
Note: TM_POST_SAVE1 is processed differently for SAVE than for other commands.	POST_SAVE	POST_SAVE1 - E,F/F ■ SAVE_ROLLBACK - E/F ■ SAVE_SET_MODE ■ SAVE_COMMIT - E/F ■ SAVE_SET_MODE ■ GIVE_UP_SAVE - E/F ■ POST_SAVE2 - F ■ POST_SAVE2 - F POST_SAVE2 - F

E = TM_TEST_ERROR, O = TM_TEST_ONE_ROW, F = TM_NOTE_FAILURE;
*All request and slice events have TM_ prefix.
**Slice events generated based on server view property specifications.

Table 9-1 Transaction manager request events *(Continued)*

Command	Request with Corresponding Error Events	Slices with Corresponding Error Events
SELECT	PRE_SELECT	
	SELECT	GET_SEL_CURSOR - F PREPARE_CONTINUE - E/F ■ GIVE_UP_SEL_CURSOR - E/F SET_SEL_COUNT_FLAG** ■ SEL_GEN - F ■ GIVE_UP_SEL_CURSOR - E/F ■ SEL_BUILD_PERFORM - E,F/F ■ COUNT_CHECK ■ GIVE_UP_SEL_CURSOR - E/F ■ CLEAR_SEL_COUNT_FLAG - F SEL_GEN - F ■ GIVE_UP_SEL_CURSOR - E/F SEL_BUILD_PERFORM - E,F/F SEL_CHECK - E/F ■ GIVE_UP_SEL_CURSOR - E/F
	CLEAR	
	POST_SELECT	
START	START	
VALIDATE_LINK (internally generated command)	PRE_ VAL_LINK	
	VAL_LINK	GET_SAVE_CURSOR - F VAL_GEN - F ■ GIVE_UP_SAVE - E/F VAL_BUILD_PERFORM - E,F/F VAL_CHECK - E/F GIVE_UP_SAVE - E/F
	POST_VAL_LINK	

E = TM_TEST_ERROR, O = TM_TEST_ONE_ROW, F = TM_NOTE_FAILURE;
*All request and slice events have TM_ prefix.
**Slice events generated based on server view property specifications.

Table 9-1 Transaction manager request events *(Continued)*

Command	Request with Corresponding Error Events	Slices with Corresponding Error Events
VIEW	PRE_ VIEW	
	VIEW	GET_SEL_CURSOR - F PREPARE_CONTINUE - E/F ■ GIVE_UP_SEL_CURSOR - E/F SEL_GEN - F ■ GIVE_UP_SEL_CURSOR - E/F SEL_BUILD_PERFORM - E,F/F SEL_CHECK - E/F ■ GIVE_UP_SEL_CURSOR - E/F
	CLEAR	
	POST_SELECT	
WALK_DELETE	WALK_DELETE	
WALK_INSERT	WALK_INSERT	
WALK_SELECT	WALK_SELECT	
WALK_UPDATE	WALK_UPDATE	

E = TM_TEST_ERROR, O = TM_TEST_ONE_ROW, F = TM_NOTE_FAILURE;
*All request and slice events have TM_ prefix.
**Slice events generated based on server view property specifications.

Error and Diagnostic Events

The transaction error and diagnostic events are generated as a result of return values for other events. The most common error and diagnostic events are shown in the Error columns in Table 63. They are: TM_TEST_ERROR, TM_TEST_ONE_ROW and TM_NOTE_FAILURE.

The slice event TM_SEL_BUILD_PERFORM in all the CONTINUE commands has two events (E and F) in the first Error column, because TM_SEL_BUILD_PERFORM can return TM_CHECK, in addition to TM_FAILURE and TM_OK. In this table, the second F is associated with the E, not with the first F. It is the TM_TEST_ERROR event from TM_SEL_CHECK that can give rise to a further TM_NOTE_FAILURE event.

Database-Specific Transaction Models

Panther transaction models perform specialized processing for their respective databases. Table 9-2 lists the supported databases, the corresponding model name, and the type of non-trivial processing performed by each model.

Table 9-2 Databases and transaction model processing

Database	Model	INITIAL mode handling	BEGIN command processing	Specialized processing
DB2	tmdb21.c			
Informix	tminf1.c	x	x	Special subroutine.
JDB	tmjdb1.c			Check for duplicates.
SQL Server	tmmss1.c	x	x	Cursor management, etc.
ODBC	tmodb1.c	x		
Oracle	tmora1.c			Name and save rollback special tp processing.
Sybase	tmsyb1.c	x	x	Cursor management, etc.

Panther support of databases is subject to change. Panther continually updates database-specific transaction models in order to be consistent with DBMS systems.

INITIAL Mode Handling

For `TM_SAVE_SET_MODE`, the model sets `TM_VALUE` to `TM_INITIAL_MODE`. This means that when a `COMMIT` has been done in the course of a `SAVE` command, the model suggests that the mode be set to `INITIAL`. (Such a suggestion of a mode change can be ignored, particularly in a web application context.)

BEGIN Command Processing

For `TM_SAVE_BEGIN`, the transaction model does a `BEGIN` command.

Special Processing

The following models perform specialized processing:

- `tminf1.c`—`INITIAL` mode processing is done only if the special subroutine `dm_inf_static_cursors` so specifies.
- `tmjdb1.c`—If rows are inserted into the database during `SAVE` processing, a special check (optional) for duplication rows in that table after the insertion. (This entails cursor name generation and giving up the cursor used for this test, so there are several slices beyond those obviously needed for the test.)
- `tmora1.c`—For `TM_SAVE_BEGIN`, the model performs a `DBMS SAVE` command; in the event that a `ROLLBACK` is necessary, the `DBMS ROLLBACK` uses the same name. For Oracle Tuxedo applications, if in an XA context, XA processing is done rather than the corresponding DBMS operations for `BEGIN`, `COMMIT` and `ROLLBACK`; the timing of the XA operations can also be somewhat different from the timing of the DBMS operations.
- `tmsyb1.c`—There is specialized cursor management. This includes reusing cursors, rather than closing cursors that are given up. It also entails flushing cursors that are used for `SELECT` operations, and giving up select cursors before inserting, updating, or deleting tables on which they were used. `commit` and `ROLLBACK` are performed on the save cursor.

CHAPTER

10 Transaction Manager Error Messages

The transaction manager error messages are listed in alphabetical order with a possible cause and solution for each message. Those containing an error constant are stored in the Panther message file. Those without an error constant are caused by errors in SQL generation.

Transaction Manager Errors

Bad arguments (`DM_BAD_ARGS`)

Cause	General error. The `START` command was issued without a transaction name or a bad value was specified for the return code of an event function.
Action	n/a

Bad field name, #, or subscript at line *`line_number`*

Cause	Standard JPL error; generally indicates the JPL procedure or variable causing the error. One cause which is not a syntax error is using the property API to query for the value of server view, table view or link when it is not in the current traversal tree or for the value of `num_key_columns` when a database modification command is not in effect.
Action	Edit the JPL procedure.

Bad mode (`DM_TM_BAD_MODE`)

Cause	Command availability varies according to the transaction mode.
Action	1. Refer to "Setting the Transaction Mode" on page 34-7 in *Application Development Guide* for the command availability in each mode. 2. Use the `COPY_FOR_UPDATE` and `COPY_FOR_VIEW` commands, which set the mode, when appropriate. 3. For menu items and push buttons, set the class property which controls the active/inactive property according to the transaction mode.

Column *`column-name`* **not found in table view** *`table-view-name`* **specified in link** *`link-name`*

Cause	1. Invalid column name specified in the link's Relations property. 2. Parent and Child entries for the Relations need to be reversed.
Action	Edit the relations property to contain the column names which join the two table views named in the link. Check that column names exist in the corresponding Parent and Child table views.

Discard all changes? (`DM_TM_DISCARD_ALL`)

Cause	Transaction manger command was executed without saving the changes made to onscreen data.
Action	Choose Yes to discard changes; choose No to return to the screen so that the `SAVE` command can be executed.

Discard latest changes? (`DM_TM_DISCARD_LATEST`)

Cause	Transaction manger command was executed without saving the changes made to a portion of onscreen data.
Action	Choose Yes to discard changes; choose No to return to the screen so that the `SAVE` command can be executed.

Error executing database command (`DM_TM_DBI_ERROR`)

Cause	Error occurred while executing a command in one of Panther's database drivers.
Action	Refer to error for action.

Error in User hook function or Transaction Model (`DM_TM_HOOK_MODEL_ERROR`)

Cause	This error lists whether a model or function is being accessed, the name of the model or function, and the event that failed. One common usage is to display the failed event after an error has been reported from the database engine.
Action	Generally, the engine error is more descriptive of the problem

Invalid field type for Version Column (`DM_TM_VC_TYPE`)

Cause	Version columns must have the C Type property set to `Int`, `Long`, `Float` or `Double`.
Action	Change C Type property. As a result, database re-design might be necessary.

Invalid sort order *`type`* specified in the sort-columns edit of tableview *`table-view`*

Cause	Value entered for the sort type is invalid.
Action	Change the sort order type to `ASC` or `DESC`.

Invalid widget *`widget`* specified in the sort-columns edit of tableview *`table-view`*

Cause	The Sort Widgets property does not contain a valid widget name.
Action	Check the table view's `sort_widgets` property and make sure the widget is on the screen; it is the widget name and not the database column name that must be specified.

Loop in transaction manager event processing (`DM_TM_EVENT_LOOP`)

Cause	1. Transaction event function specified in Function property is defined without passing it the event argument. 2. Transaction event function has incorrect or invalid return code specified, for example, `TM_CHECK` instead of `TM_PROCEED` when no database driver statement was issued for that event.
Action	For 1, add (event) after the procedure name. For 2, change return code.

Maximum depth exceeded

Cause	1) There is a circular link in the Parent and Child properties. 2) A link has both the Parent and Child properties set to the same table view.
Action	Check parent and child properties for each link, editing where necessary.

`mode` **does not permit command *command* (`DM_TM_CMD_MODE`)**

Cause	Command availability varies according to the transaction mode
Action	1. Refer to "Setting the Transaction Mode" on page 34-7 in *Application Development Guide* for the command availability in each mode. 2. Use the `COPY_FOR_UPDATE` and `COPY_FOR_VIEW` commands, which set the mode, when appropriate. 3. For menu items and push buttons, set the class property which controls the active/inactive property according to the transaction mode.

More than one row affected (`DM_TM_ONE_ROW`)

Cause	`TM_CHECK_ONE_ROW`, which calls the `TM_TEST_ONE_ROW` event to check that `@dmrowcount` is equal to 1, has been set as the return code either in the transaction model or in an event function.
Action	Change the return code in the model or event function. Change the SQL generation, for example, by expanding the primary key values so that only one row is changed. Check data in the database to make sure that duplicate key values have not been entered.

No rows affected (`DM_TM_SOME_ROWS`)

Cause	`TM_CHECK_SOME_ROWS`, which calls the `TM_TEST_SOME_ROWS` event to check that `@dmrowcount` is equal to or greater than 1, has been set as the return code either in the transaction model or in a event function.
Action	If error is valid, do nothing. Otherwise, change the return code in the model or event function.

No select columns specified, first table view *`table-view`*

Cause	For all the members of this table view, either the `column_name` property is blank or the `use_in_select` property is set to `PV_NO`.
Action	Set the appropriate properties for each widget.

No such command as *`command`* (`DM_TM_NO_SUCH_CMD`)

Cause	The syntax of `sm_tm_command` is incorrect.
Action	Edit the call to `sm_tm_command` so that a valid command name is the first parameter of its quoted command string.

No such scope as in *`scope`* (`DM_TM_NO_SUCH_SCOPE`)

Cause	The syntax of `sm_tm_command` is incorrect.
Action	Edit the call to `sm_tm_command` so that a valid scope parameter follows the command and table view parameters.

No such table view as in *`table-view`* (`DM_TM_NO_SUCH_TV`)

Cause	The syntax of `sm_tm_command` is incorrect.
Action	Edit the call to `sm_tm_command` so that a valid table view name follows the command parameter in the quoted command string.

Primary key not specified for updatable Tableview *`table-view`* (`DM_TM_PRIMARYKEY`)

Cause	For commands that could result in database modifications, like `SELECT`, `NEW`, `COPY`, or `COPY_FOR_UPDATE`, the transaction manager checks that primary key fields for a table view are available.
Action	Specify the table view's primary key in the `primary_key` property.

Root table view name not supplied or not valid (`DM_TM_NO_ROOT`)

Cause	1. Table view parameter supplied with `START` command is not valid. 2. More than one table view appears on a screen and there are no link widgets.
Action	For 1, edit `START` command specification. For 2, create a link widget with the appropriate Parent, Child and Relations property settings.

Table name not specified for tableview *`table-view`*

Cause	Table property is blank for this table view.
Action	Enter the name of the database table in the table view's Table property. For the format needed by a specific database driver, refer to "Database Drivers."

Table name not specified for Tableview (`DM_TM_TBLNAME`)

Cause	Table property is blank for this table view.
Action	Enter the name of the database table in the table view's Table property. For the format needed by a specific database driver, refer to "Database Drivers."

Tableview *`table-view`* is updatable but its primary key is incomplete

Cause	For commands that could result in database modifications, like `SELECT`, `NEW`, `COPY`, or `COPY_FOR_UPDATE`, the primary key fields of an updatable table view must either be a member of that table view or one of its parent table views. It does not have to be in the direct parent, it can be in the "grandparent" table view.
Action	1. Add the widget to the desired table view. 2. Check the link's relations property to see if the shared fields between the table views is complete. 3. Change the table view to non-updatable. 4. Check the link's relations property to make sure the relation type is valid.

Transaction in progress (`DM_TM_IN_PROGRESS`)

Cause	`sm_tm_command` is being called recursively.
Action	Do not call `sm_tm_command` in transaction manager event functions.

Transaction model not found (`DM_TM_NO_MODEL`)

Cause Model specified in table view or screen properties is not initialized.

Action Specify valid model or leave blank to use standard model.

Transaction unspecified or unavailable (`DM_TM_NO_TRANSACTION`)

Cause

1. Transaction manager command was called in an unnamed JPL procedure. Since Panther calls the unnamed JPL procedure before it calls the `START` command on screen entry, an error occurs.
2. More than one table view appears on a screen and there are no link widgets.
3. `START` command was not issued because of error in table view tree or because the table view parameter specified with the command was invalid.

Action For 1, call the command after the `START` command has been invoked, for example, in the screen entry procedure.

For 2, create a link widget with the appropriate Parent, Child and Relations property settings.

For 3, check Parent and Child properties. Check specification of additional `START` commands.

Unable to synchronize server view (`DM_TM_SYNCH_SV`)

Cause For all updatable table views, the transaction manager synchronizes the tableviews when a command that could modify data is issued, for example, `SELECT`, `NEW`, `COPY`, or `COPY_FOR_UPDATE`.

Action

1. If possible, set all the members of the same table view whose `use_in_update` property is set to `PV_YES` to the same number of occurrences
2. Change the synchronization property to `PV_NO`.

User event function not found (`DM_TM_NO_HOOK`)

Cause The table view's Function property specifies a name that is not available either as a JPL procedure or as a prototyped function.

Action

1. Check the value of the function property.
2. Check the prototyped function list.
3. Check that the JPL procedure name matches the function property and that the JPL module is available.

Version Column setting on widget is incompatible with the properties *property_name*

Cause	If a widget's `version_column` property is `PV_YES`, then the properties `in_delete_where` and `in_update_where` must be set to `PV_NO`.
Action	Set the properties to the correct values

CHAPTER

11 DBMS Statements and Commands

This chapter describes the DBMS (dbms) commands, in alphabetical order, that are supported by all database engines.

Each reference page contains the following information:

- The command name.
- Usage synopsis.
- Full description of the command, with an explanation of its parameters, outputs, and actions.
- One or more examples of JPL procedures demonstrating how the command is used.

The commands can be executed with the JPL command DBMS and with the C library function dm_dbms. Some database engines support additional commands; for DBMS commands that are specific to a database engine, refer to "Database Drivers." This includes the transaction commands and any special feature commands.

Since DBMS is a JPL command, using these commands inside a JPL statement must follow all the conventions for JPL.

DBMS Command Summary

The following listing is a summary of the `DBMS` commands by category. Some commands might appear in more than one category.

Selecting a Database Engine

`ENGINE`
: Sets the default database engine for the application.

`WITH ENGINE`
: Sets the engine to use for a command.

Using Connections

`CLOSE CONNECTION`
: Closes a named connection.

`CLOSE_ALL_CONNECTIONS`
: Closes all connections on the named or on the default engine.

`CONNECTION`
: Sets a default connection and engine for the application.

`DECLARE CONNECTION`
: Declares a named connection to a database engine.

`WITH CONNECTION`
: Sets the connection to use for a command.

Using Cursors

`CLOSE CURSOR`
: Closes a cursor.

`CONTINUE`
: Fetches the next rows from a select set.

`DECLARE CURSOR`
: Declares a named cursor.

`EXECUTE`
: Executes a named cursor.

`WITH CURSOR`
: Specifies the cursor to use for a command.

Executing SQL Statements

`QUERY`
: Specifies an SQL statement that returns one or more select sets to be passed to the database engine for processing.

`RUN`
: Specifies an SQL statement that will not return any select sets to be passed to the database engine for processing.

`SQL`
: Specifies an SQL statement to be passed to the database engine for processing (not recommended).

Changing `SELECT` Behavior

`ALIAS`
: Defines Panther variables as the destination of selected columns and/or aggregate functions in a select set.

`BINARY`
: Defines Panther variables for fetching binary values.

`CATQUERY`
: Redirects `SELECT` results to a file or a Panther variable.

`COLUMN_NAMES`
: Maps a database column name to a Panther variable.

`FORMAT`
: Formats the results of a `CATQUERY`.

`OCCUR`
: Optionally sets the number of rows for Panther to fetch to an array and chooses an occurrence where Panther should begin writing result rows.

`START`
: Sets the first row for Panther to return from a select set.

`UNIQUE`
: Suppresses repeating values in a selected column.

Paging through Multiple Rows

`CONTINUE`
: Fetches the next screenful of rows from a select set.

`CONTINUE_BOTTOM`
: Fetches the last screenful of rows from a select set.

`CONTINUE_DOWN`
: Fetches the next screenful of rows from a select set.

`CONTINUE_TOP`
: Fetches the first screenful of rows from a select set.

`CONTINUE_UP`

Fetches the previous screenful of rows from a select set.

`STORE`

Stores the rows of a select set in a temporary file so that the application can scroll through the rows.

Handling Binary Data

`BINARY`

Defines one or more binary variables.

Status and Error Processing

`ONENTRY`

Installs a function or JPL procedure which Panther calls before executing each `DBMS` statement.

`ONERROR`

Installs a function or JPL procedure which Panther calls whenever a `DBMS` statement fails.

`ONEXIT`

Installs a function or JPL procedure which Panther calls after executing each `DBMS` statement.

ALIAS

Sets aliases for a declared or default SELECT *cursor*

Synopsis

```
DBMS [WITH CURSOR cursor] ALIAS [column pantherVar
    [, column pantherVar ...]]
```

```
DBMS [WITH CURSOR cursor] ALIAS [pantherVar [, pantherVar ...]]
```

Arguments

`WITH CURSOR` *`cursor`*
: Name of a declared `SELECT` cursor. If the clause is not used, Panther uses the default `SELECT` cursor.

`column`
: Name of column in the database table.

`pantherVar`
: Name of Panther variable to contain the data.

Description

By default, database values are written to Panther variables with the same names as the selected database columns. Use `DBMS ALIAS` to map a database column or value to any Panther variable.

If a column name is given, it is associated with the variable name that follows it. For example:

```
DBMS ALIAS name title, film_minutes length
```

If the database column `name` is selected with the default cursor, its value is written to the Panther variable `title`. If the column `film_minutes` is selected with the default cursor, its value is written to the Panther variable `length`. For all other columns selected with the default cursor, the column's value is written to a variable with the same (unqualified) name as the selected column.

If *`column`* contains characters not permitted in Panther identifiers, enclose *`column`* in quotes to ensure correct parsing. For example:

```
DBMS ALIAS "last-name" last_name
```

The case of *`column`* needs to match the setting of the case flag used to initialize the database engine. For example, if the case flag is `DM_FORCE_TO_LOWER_CASE`, *`column`* must be entered in lower case. The case of *`pantherVar`* must be the case used to name the Panther variable. If *`pantherVar`* does not exist, Panther ignores the column when it executes the `SELECT`. Refer to "Database Drivers" for details of case setting for each database engine.

If no *`column`* arguments are given, the association is positional. For example:

```
DBMS ALIAS title_var, , abc
```

When the above statement is executed, each time values are selected with the default cursor, Panther writes the values of the first and third columns to the Panther variables `title_var` and `abc`, respectively. For all other columns selected with the default cursor, Panther writes to a variable with the same (unqualified) name as the selected column. The order of column names in the `select` statement determines the mapping. Named and positional aliases cannot be assigned in a single statement.

Only one `DBMS ALIAS` statement is allowed for each cursor. The last `DBMS ALIAS` statement called is the one currently in effect.

If aliases are declared for a `CATQUERY` cursor with the `HEADING ON` option, Panther uses the aliases rather than the column names to build the heading. The alias for a column selected with a `CATQUERY` cursor can be enclosed in quotes. This permits a column heading to use embedded spaces. For example:

```
DBMS DECLARE t_cursor CURSOR FOR \
   SELECT title_id, name, pricecat FROM titles
DBMS WITH CURSOR t_cursor CATQUERY TO FILE t_list
DBMS WITH CURSOR t_cursor ALIAS \
   "Title ID", Name, "Price Category"
DBMS WITH CURSOR t_cursor EXECUTE
```

Aliasing for a cursor is turned off by executing the `DBMS ALIAS` command with no arguments. Closing a cursor also turns off aliasing. If a cursor is redeclared without being closed, the cursor keeps the aliases. Aliases do not affect `INSERT`, `UPDATE`, or `DELETE` statements.

The `ALIAS` command is necessary if the name of a selected column is not a valid Panther variable name, if the application is selecting values from different tables which use the same column name for different values, or if a selection is not a column value, but the value of an aggregate function or select expression.

Example

```
// Assign named aliases for a declared cursor.

DBMS DECLARE x CURSOR FOR \
   SELECT title_id, copy_num, status FROM tapes
DBMS WITH CURSOR x ALIAS \
   title_id code, copy_num copy, status current_status
```

```
DBMS WITH CURSOR x EXECUTE
DBMS WITH CURSOR x ALIAS

// Set a positional alias for the 2nd and 4th columns.
// Use automatic mapping for the 1st and 3rd columns.

DBMS ALIAS , var_x, , var_y
DBMS QUERY SELECT title_id, name, genre_code, release_date \
   FROM titles

// Panther will write
//  column title_id     to variable title_id,
//  column name         to variable var_x,
//  column genre_code   to variable genre_code, and
//  column release_date to variable var_y.

// Note how the mappings change when the columns are
// listed in another order.

DBMS QUERY SELECT name, genre_code, release_date, title_id \
   FROM titles

// Panther will write
//  column name         to variable name,
//  column genre_code   to variable var_x,
//  column release_date to variable release_date, and
//  column title_id     to variable var_y.
```

See Also `CATQUERY`, `WITH CURSOR`

BINARY

Defines Panther variables for fetching binary values

Synopsis `DBMS BINARY` *`variable`* `[,` *`variable`* `...]`

Arguments *`variable`*

Name of binary variable Panther creates. The variable can contain the number of occurrences and/or a length. Refer to the Description for more information.

Description

Many database engines support a binary data type for byte strings and other non-printable data. There are two ways an application can fetch binary values to Panther variables (widgets, LDB variables, or JPL variables):

- To variables created with the `DBMS BINARY` command.
- To text widgets which have their C Type (`c_type`) property set to Hex Dec (`PV_HEX_DEC`). Panther converts the binary data to hexadecimal strings.

The definition for a variable created with `DBMS BINARY` can include a number of occurrences and/or a length. If a number of occurrences is supplied, it must be enclosed in square brackets. If a variable length is supplied, it must be enclosed in parentheses. If both are supplied, the number of occurrences must be first. Any of the following are permitted:

```
db_binvar
db_binvar [10] (255)
db_binvar [5]
db_binvar (8)
```

Any valid Panther variable name is a legal `BINARY` variable name. The default number of occurrences is 1, and the default length is 255. The maximum length is platform-dependent, based on the platform's maximum length for unsigned integers. Memory is allocated for the occurrences when they are used (that is, when a binary column is fetched).

If an application is selecting a binary column, use `DBMS BINARY` to create a binary variable for the column. The variable can have the same name as the column, or it can be mapped to the column with `DBMS ALIAS`. Because a binary variable is a target of a `SELECT`, Panther examines its number of occurrences when determining how many rows to fetch. Therefore, the binary variable should have the same number of occurrences as the other Panther target variables. When searching for target variables, Panther searches among the binary variables before searching among the Panther variables. You are responsible for ensuring that the binary variable names do not conflict with Panther variable names.

Binary variables can also be included in the `USING` clause of a `DBMS EXECUTE` statement. If no occurrence is given for the variable, the first occurrence is the default.

Once defined, a binary variable is available to the rest of the application. Note that

```
DBMS BINARY db_binvar[10]
DBMS BINARY timestamp[100]
```

is the same as

```
DBMS BINARY db_binvar[10], timestamp[100]
```

To delete all binary variables, execute `DBMS BINARY` with no arguments:

```
DBMS BINARY
```

Several library functions are provided for accessing and manipulating binary variables. These functions are only available in C. (Refer to the specific functions in this reference for more information.)

Example

```
// "timestamp" is a binary column and "timeval"
// is a binary variable.

DBMS BINARY timeval
DBMS ALIAS timestamp timeval
DBMS QUERY SELECT id, name, price, timestamp FROM products

DBMS DECLARE upd_cursor CURSOR FOR \
   UPDATE products SET price = ::priceval \
   WHERE id = ::idval and timestamp = ::timeval
DBMS WITH CURSOR upd_cursor EXECUTE \
   USING priceval, idval, timeval
```

See Also

`dm_bin_create_occur`, `dm_bin_delete_occur`, `dm_bin_get_dlength`, `dm_bin_get_occur`, `dm_bin_length`, `dm_bin_max_occur`, `dm_bin_set_dlength`

CATQUERY

Concatenates a full result row to a Panther variable or file

Synopsis

```
DBMS [WITH CURSOR cursor] CATQUERY TO pantherVar
   [SEPARATOR "text"] [HEADING [ON | OFF] ]

DBMS [WITH CURSOR cursor] CATQUERY TO FILE file
   [SEPARATOR "text"] [HEADING [ON | OFF] ]

DBMS [WITH CURSOR cursor] CATQUERY TO FILENAME fileVar \
   [SEPARATOR "text"] [HEADING [ON | OFF] ]
```

Arguments

`WITH CURSOR` *`cursor`*
: Name of declared `SELECT` cursor. If the clause is not used, Panther uses the default `SELECT` cursor.

`TO` *`pantherVar`*
: Name of destination Panther variable.

`TO FILE` *`file`*
: Name of destination text file. If the file already exists, it is overwritten when the `SELECT` is executed. In Panther 5.50 and later *`file`* can be in quotes.

`TO FILENAME` *`fileVar`*
: Name of variable whose value is the name of the destination text file. If the file already exists, it is overwritten when the `SELECT` is executed. You can use this variant for filenames that include spaces and/or punctuation.

`SEPARATOR "`*`text`*`"`
: Specifies text to use to separate column values in a result row. The default is two blank spaces.

`HEADING ON`
: Specifies that Panther put a heading at the beginning of the select results. This is the default for a catquery to a file. The heading is built using the column names or any aliases assigned to the cursor. The maximum length of a heading is 255 characters. Any additional characters are truncated.

`HEADING OFF`
: Specifies that Panther not use a heading. This is the default for a catquery to a Panther variable.

Description

The result columns of a SELECT statement are usually mapped to individual variables. Use CATQUERY to map full result rows to a variable's occurrences or to a text file.

Panther attempts to format the column values by searching for Panther variables of the same name and using their attributes for length, precision, and date-time or currency specifications. The application can override any default formatting with the command FORMAT.

The catquery for a cursor is turned off by executing the DBMS CATQUERY command with no arguments. Closing a cursor also turns off the catquery. If a cursor is redeclared without being closed, the cursor keeps the catquery destination as the cursor's SELECT destination.

Catquery to a Variable

When the catquery destination is a Panther variable, Panther concatenates a result row and writes it to *pantherVar* when the SELECT is executed. If *pantherVar* is an LDB or onscreen array, the result rows are written to the array occurrences. If there are more result rows than occurrences in *pantherVar*, use CONTINUE to fetch the additional rows.

If the clause HEADING ON is used, a heading is created by using the cursor's aliases and column names. If *pantherVar* has two or more occurrences, the heading is put in the first occurrence of *pantherVar*.

Catquery to a Text File

When the catquery destination is a text file, all the result rows are written to the specified text file when the SELECT is executed. Any existing file with the same name is overwritten. If a result row is longer than the page width, the row wraps to the next line.

Note: Only 1000 characters per row can be written to a file if the database column's type is defined as FT_VARCHAR. If more data output are required, consider outputting results to a report.

If the name of the file includes spaces and/or punctuation, use the FILENAME variant to name a variable that contains the file name. In Panther 5.50 and later, you can also quote *file*.

If aliases have been specified for the cursor, those aliases are used as column headings in the text file. If there are no aliases, the columns' names are used. If the HEADING OFF clause is used, a heading is not output.

Since all result rows are written to the file, the CONTINUE commands should not be used with a CATQUERY TO FILE cursor while the file is open.

The file remains open until `DBMS CATQUERY` is reset or the cursor is closed.

Example

```
// Select a customer's first and last name
// and concatenate the values in the field "fullname".

DBMS DECLARE name_cursor CURSOR FOR \
   SELECT last_name, first_name FROM customers \
   WHERE cust_id = :+cust_id
DBMS WITH CURSOR name_cursor CATQUERY TO fullname \
   SEPARATOR ","
DBMS WITH CURSOR name_cursor EXECUTE
return

// Select the maximum value from the column "cost"
// and write it to the JPL variable "hi_cost"
// formatting it with currency edit saved with the
// LDB variable "money_var".

vars hi_cost
DBMS DECLARE max_cursor CURSOR FOR \
   SELECT MAX(price) FROM pricecats
DBMS WITH CURSOR max_cursor CATQUERY TO hi_cost
DBMS WITH CURSOR max_cursor FORMAT money_var
DBMS WITH CURSOR max_cursor EXECUTE
return

// Write the results of the default SELECT cursor
// to a file with heading. Turn off ALIAS and CATQUERY
// when finished.

proc file_list
DBMS CATQUERY TO FILE titlelist
DBMS ALIAS title_id "Title ID", name "Title",\
   film_minutes "Length", pricecat "Price Category"
DBMS QUERY SELECT title_id, name, film_minutes, pricecat \
   FROM titles
DBMS CATQUERY
DBMS ALIAS
return

// Write results of the default SELECT cursor
// to a named file
proc title_list
vars fname
fname = "my titles file"
DBMS CATQUERY TO FILENAME fname
DBMS QUERY SELECT * FROM titles
DBMS ALIAS
DBMS FORMAT
return
```

CLOSE_ALL_CONNECTIONS

Closes all connections on a database engine

Synopsis `DBMS [WITH ENGINE` *`engine`*`] CLOSE_ALL_CONNECTIONS`

Arguments `WITH ENGINE` *`engine`*

Name of engine for which connections are to be closed. If the clause is not used, Panther closes all the connections on the default engine.

Description When `DBMS CLOSE_ALL_CONNECTIONS` is executed, every connection which the application declared either on the named database engine or on the default engine closes. For each connection, it closes all cursors belonging to the connection, disconnects from the database engine, and frees all structures associated with the connection.

If the application accesses multiple engines, include the `WITH ENGINE` clause and issue the statement for each engine used in the application.

Example

```
// This procedure unsets the error handler and
// then closes all connections.

proc logoff
DBMS ONERROR
DBMS CLOSE_ALL_CONNECTIONS
return
```

See Also `DECLARE CONNECTION`, `CLOSE CONNECTION`, `dm_is_connection`

CLOSE CONNECTION

Closes a declared connection

Synopsis `DBMS CLOSE CONNECTION [`*`connection`*`]`

Arguments *`connection`*

Name of connection to be closed. If connection name is not included, the default connection is closed.

Description Executing `DBMS CLOSE CONNECTION` closes all open cursors associated with the named or default connection, logs off the connection from its database engine, and frees the connection data structure.

Example

```
// This procedure unsets the error handler and
// then closes the specified connection.

proc logoff
DBMS ONERROR
DBMS CLOSE CONNECTION c1
return
```

See Also `CLOSE_ALL_CONNECTIONS`, `dm_is_connection`

CLOSE CURSOR

Closes a named or default cursor

Synopsis

```
DBMS CLOSE CURSOR [cursor]

DBMS WITH CONNECTION connection CLOSE CURSOR [cursor]

DBMS WITH CURSOR cursor CLOSE CURSOR
```

Arguments

cursor
Name of cursor to be closed. If cursor is not listed, Panther closes the default `SELECT` cursor.

`WITH CONNECTION` *connection*
Name of connection having the cursor to be closed.

`WITH CURSOR` *cursor*
Name of cursor to be closed.

Description

`DBMS CLOSE CURSOR` closes an open cursor. Closing a cursor frees all structures associated with the cursor.

Closing a cursor turns off all attributes assigned to the cursor with the `DBMS` commands: `ALIAS`, `CATQUERY`, `COLUMN_NAMES`, `FORMAT`, `OCCUR`, `START`, `STORE`, `TYPE`, and `UNIQUE`.

To close the default `SELECT` cursor on the default connection, specify:

```
DBMS CLOSE CURSOR
```

To close the default `SELECT` cursor on a specific connection, specify:

```
DBMS WITH CONNECTION connection CLOSE CURSOR
```

Panther will automatically declare another default `SELECT` cursor if needed.

To close a named cursor, specify either of the following:

```
DBMS CLOSE CURSOR cursor
DBMS WITH CURSOR cursor CLOSE CURSOR
```

Closing a connection first closes all cursors associated with the connection.

Example

```
// Assign a catquery and aliases to the default SELECT
// cursor. Close the cursor when finished.

DBMS CATQUERY TO FILE titlelist
DBMS ALIAS title_id "Title ID", name "Title",\
   film_minutes "Length", pricecat "Price Category"
DBMS QUERY SELECT title_id, name, film_minutes, pricecat \
   FROM titles
DBMS CLOSE CURSOR
```

See Also

`DECLARE CURSOR`, `dm_is_cursor`

COLUMN_NAMES

Map column names into Panther variables using a `SELECT` *statement*

Synopsis

```
DBMS [WITH CURSOR cursor] COLUMN_NAMES [pantherVar
     [, pantherVar ...] ]
```

Arguments

`pantherVar`
: Name of Panther variable to contain the column name.

`WITH CURSOR` *`cursor`*
: Name of declared `SELECT` cursor. If the clause is not used, Panther uses the default `SELECT` cursor.

Description

`DBMS COLUMN_NAMES` fetches the column names, not the column data, into Panther variables when a `SELECT` statement is executed.

The correspondence between the Panther variable and the column is positional. The first Panther variable named in the `DBMS COLUMN_NAMES` command will contain the name of the first column listed in the `SELECT` statement. If the number of Panther variables is greater than the number of columns, the remaining Panther variables are ignored. If the number of columns is greater than the number of Panther variables, the remaining columns are ignored.

If the `SELECT` statement includes data which is not a column, like an aggregate function, then the value written to the Panther variable is whatever is returned from the database engine.

A Panther variable can be a widget or JPL variable. If the variable is an array or multi-occurrence widget, the column name appears in the first occurrence unless a particular occurrence is specified.

Only one `DBMS COLUMN_NAMES` statement is allowed for each cursor. The last `DBMS COLUMN_NAMES` statement called is the one currently in effect.

Column name aliasing for a cursor is turned off by executing the `DBMS COLUMN_NAMES` command with no arguments. Closing a cursor also turns it off. If a cursor is redeclared without being closed, the cursor keeps the aliases.

Example

```
// Assign column name aliases for a declared cursor.
// The column names are written to id_title, copy_title
// and status_title.
// The data is written is title_id, copy_num and status.

DBMS DECLARE x CURSOR FOR \
   SELECT title_id, copy_num, status FROM tapes
DBMS WITH CURSOR x COLUMN_NAMES \
   id_title, copy_title, status_title
DBMS WITH CURSOR x EXECUTE
DBMS WITH CURSOR x COLUMN_NAMES

//  Assign column name aliases for the default cursor

DBMS COLUMN_NAMES id_title, copy_title, status_title
DBMS QUERY SELECT title_id, copy_num, status \
   FROM tapes
DBMS COLUMN_NAMES
```

CONNECTION

Sets or changes the default connection

Synopsis `DBMS CONNECTION` *`connection`*

Arguments

`connection`
: Name of the connection to set as the default.

Description If an application declares two or more connections, the application can set a default connection with `DBMS CONNECTION`. The default connection is used for all subsequent statements that do not use a `WITH CONNECTION` or `WITH CURSOR` clause.

Example

```
// con1 is set to be the default connection.
// The INSERT statement has a WITH CONNECTION clause
// using connection con2.
// The SELECT statement uses the default connection.


DBMS ENGINE sybase

DBMS DECLARE con1 CONNECTION FOR USER ":uname" \
    PASSWORD ":pword" SERVER "s1" DATABASE "master"

DBMS DECLARE con2 CONNECTION FOR USER ":uname" \
    PASSWORD ":pword" SERVER "s2" DATABASE "videobiz"

DBMS CONNECTION con1

DBMS WITH CONNECTION con2 DECLARE c1 CURSOR FOR \
    INSERT INTO tapes (title_id, copy_num, status) \
    VALUES (::title_id, ::copy_num, ::status)

DBMS WITH CURSOR c1 EXECUTE USING title_id, copy_num, status

DBMS SELECT title_id, name FROM titles
```

See Also `DECLARE CONNECTION`, `WITH CONNECTION`, `dm_is_connection`

CONTINUE

Fetches the next set of rows associated with a default or named `SELECT` *cursor*

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

Arguments `WITH CURSOR` *`cursor`*

Name of declared `SELECT` cursor. If the clause is not used, Panther uses the default `SELECT` cursor.

Description If a `SELECT` statement retrieves more rows than can fit in its destination variables, Panther returns as many rows as will fit. It continues fetching more rows from the select set when the application executes this command. If there are pending rows, executing `DBMS CONTINUE` clears the destination variables, and fetches the next screen-full of rows from the select set. If there are no pending rows, executing `DBMS CONTINUE` does nothing.

If the cursor's aliases have changed between the execution of the `SELECT` and the execution of `DBMS CONTINUE`, `DBMS CONTINUE` uses the new settings. `DBMS CONTINUE` should not be used with a `CATQUERY TO FILE` cursor. `CATQUERY TO FILE` always writes out the entire select set to the `CATQUERY` file.

Example

```
// This procedure fetches the specified rows
// and calls the JPL procedure check_count.

proc get_selection
DBMS DECLARE movie_list CURSOR FOR \
   SELECT * FROM titles WHERE genre_code = ::genre_code
DBMS WITH CURSOR movie_list EXECUTE USING genre_code
call check_count
return

// This procedure sets the message line according
// to the number of rows available.

proc check_count
if @dmretcode != DM_NO_MORE_ROWS
   msg setbkstat "Press %KPF1 to see more films " \
      "or press %KPF2 to specify another type."

else
   msg setbkstat "That's all folks!"
return
```

```
// This procedure is called by pressing PF1.
// It retrieves the next set of rows.
```

```
proc get_more
DBMS WITH CURSOR movie_list CONTINUE
call check_count
return
```

See Also CONTINUE_BOTTOM, CONTINUE_DOWN, CONTINUE_TOP, CONTINUE_UP, STORE

CONTINUE_BOTTOM

Fetches the last page of rows associated with the default or named `SELECT` *cursor*

Synopsis `DBMS [WITH CURSOR cursor] CONTINUE_BOTTOM`

Arguments `WITH CURSOR cursor`

Name of declared `SELECT` cursor. If the clause is not used, Panther uses the default `SELECT` cursor.

Description `DBMS CONTINUE_BOTTOM` fetches the last screen-full of rows from the cursor's select set. If the number of rows in the select set is less than the number of occurrences in the Panther variables, the request is ignored.

Some database engines automatically support `DBMS CONTINUE_BOTTOM`. Other engines require a temporary storage file created by the command `STORE`. If the `DM_BAD_CMD` error return happens when the application executes `DBMS CONTINUE_BOTTOM`, the engine needs a scrolling file. For information about a specific engine, refer to “Database Drivers.”

`DBMS CONTINUE_BOTTOM` should not be used with a `CATQUERY TO FILE` cursor.

Example

```
// Engines not requiring DBMS STORE

proc select_all
DBMS DECLARE t_cursor FOR SELECT * FROM titles
DBMS WITH CURSOR t_cursor EXECUTE
return

proc get_last
DBMS WITH CURSOR t_cursor CONTINUE_BOTTOM
return

// Engines requiring DBMS STORE

proc select_all
DBMS DECLARE t_cursor FOR SELECT * FROM titles
DBMS WITH CURSOR t_cursor STORE FILE
DBMS WITH CURSOR t_cursor EXECUTE
return

proc get_last
DBMS WITH CURSOR t_cursor CONTINUE_BOTTOM
return
```

See Also `CONTINUE`, `CONTINUE_DOWN`, `CONTINUE_TOP`, `CONTINUE_UP`, `STORE`

CONTINUE_DOWN

Fetches the next set of rows associated with the default or named `SELECT` *cursor*

Synopsis `DBMS [WITH CURSOR cursor] CONTINUE_DOWN`

Arguments `WITH CURSOR cursor`

Name of declared `SELECT` cursor. If the clause is not used, Panther uses the default `SELECT` cursor.

Description `DBMS CONTINUE_DOWN` is identical to `DBMS CONTINUE`.

Example

```
// This procedure selects the rows from the table.

proc select_all
DBMS DECLARE t_cursor FOR SELECT * FROM titles
DBMS WITH CURSOR t_cursor EXECUTE
return

// This procedure fetches the next set of rows.

proc get_more
DBMS WITH CURSOR t_cursor CONTINUE_DOWN
return
```

See Also `CONTINUE`, `CONTINUE_BOTTOM`, `CONTINUE_TOP`, `CONTINUE_UP`, `STORE`

CONTINUE_TOP

Fetches the first page of rows associated with the default or named SELECT *cursor*

Synopsis

```
DBMS [WITH CURSOR cursor] CONTINUE_TOP
```

Arguments

`WITH CURSOR cursor`
: Name of declared `SELECT` cursor. If the clause is not used, Panther uses the default `SELECT` cursor.

Description

`DBMS CONTINUE_TOP` fetches the first screen-full of rows from the cursor's select set. If the number of rows in the select set is less than the number of occurrences in the Panther variables, the request is ignored.

Some database engines automatically support `DBMS CONTINUE_TOP`. Other engines require a temporary storage file created by the command `STORE`. If the engine needs such a file and the application tries to execute `DBMS CONTINUE_TOP` without executing `DBMS STORE`, Panther returns the error `DM_BAD_CMD`. For information about a specific engine, refer to “Database Drivers.”

Example

```
//  Engines not requiring DBMS STORE

proc select_all
DBMS DECLARE t_cursor FOR SELECT * FROM titles
DBMS WITH CURSOR t_cursor EXECUTE
return

proc go_to_start
DBMS WITH CURSOR t_cursor CONTINUE_TOP
return

//  Engines requiring DBMS STORE

proc select_all
DBMS DECLARE t_cursor FOR SELECT * FROM titles
DBMS WITH CURSOR t_cursor STORE FILE
DBMS WITH CURSOR t_cursor EXECUTE
return

proc go_to_start
DBMS WITH CURSOR t_cursor CONTINUE_TOP
return
```

See Also

`CONTINUE`, `CONTINUE_BOTTOM`, `CONTINUE_DOWN`, `CONTINUE_UP`, `STORE`

CONTINUE_UP

Fetches the previous page of rows associated with the default or named `SELECT` *cursor*

Synopsis

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

Arguments

`WITH CURSOR` *`cursor`*
: Name of declared `SELECT` cursor. If the clause is not used, Panther uses the default `SELECT` cursor.

Description

`DBMS CONTINUE_UP` scrolls backwards through a select set. If number of rows in the select set is less than the number of occurrences in the Panther variables, Panther ignores the request.

Some database engines automatically support `DBMS CONTINUE_UP`. Other engines require a temporary storage file created by the command `STORE`. If the engine needs such a file and the application tries to execute `DBMS CONTINUE_UP` before executing `DBMS STORE`, Panther returns the error `DM_BAD_CMD`. For information about a specific engine, refer to "Database Drivers."

`DBMS CONTINUE_UP` should not be used with a `CATQUERY TO FILE` cursor.

Example

```
// Engines not requiring DBMS STORE

proc select_all
DBMS DECLARE t_cursor FOR SELECT * FROM titles
DBMS WITH CURSOR t_cursor EXECUTE
return

proc go_back
DBMS WITH CURSOR t_cursor CONTINUE_UP
return

// Engines requiring DBMS STORE

proc select_all
DBMS DECLARE t_cursor FOR SELECT * FROM title
DBMS WITH CURSOR t_cursor STORE FILE
DBMS WITH CURSOR t_cursor EXECUTE
return

proc go_back
DBMS WITH CURSOR t_cursor CONTINUE_UP
return
```

See Also `CONTINUE`, `CONTINUE_BOTTOM`, `CONTINUE_DOWN`, `CONTINUE_TOP`, `STORE`

DECLARE CONNECTION

Creates a named connection to a database engine

Synopsis

```
DBMS [WITH ENGINE engine] DECLARE connection CONNECTION
    [WITH option=argVar [,... ]]

DBMS [WITH ENGINE engine] DECLARE connection CONNECION
    [FOR option arg ...]
```

Arguments

`WITH ENGINE` *`engine`*
: Name of engine to associate with the connection. If the clause is not used, Panther opens the connection on the default engine.

`connection`
: Name of connection to be opened.

`option`
: Name of connection option. Names and number of available options is engine-specific.

`argVar`
: Variable that contains the value assigned to the option, or a quoted string. Use this variant when the value might contain spaces and/or punctuation. Spaces are permitted around the equal sign.

`arg`
: Literal value, either a quoted string or a colon-expanded expression, assigned to the connection option.

Description

`DBMS DECLARE CONNECTION` opens a session on a database engine. If the statement executes successfully, it allocates a connection structure and adds it to the list of open structures.

Applications which must connect to two or more database servers should declare a named connection to each server. If you are connecting to two or more database engines, you must declare a connection for each engine.

The combination of necessary or supported options is engine-specific. Common options include `USER`, `PASSWORD`, `DATABASE`, and `SERVER`. For a list of the valid options, refer to “Database Drivers.”

Options can be specified using either of two ways:

- `WITH` variant (recommended)—The option and its argument value are separated by an equal sign (spaces are permitted), option-value pairs are comma-separated, and if the argument is a variable, it is not enclosed in quotes (and not colon-expanded). If the argument is a string, it is enclosed in quotes; spaces and special punctuation characters are permitted.
  Since the variables are not colon-expanded, this variant prevents the values of variables from appearing in error messages or trace statements.
- `FOR` variant—The option is followed by its value. The argument values are enclosed within quoted strings. If the value is a variable, it must be colon-expanded.

The connection remains open until it is closed with `DBMS CLOSE CONNECTION` or `DBMS CLOSE_ALL_CONNECTIONS`.

For additional information, refer to Chapter 8, “Connecting to Databases,” in *Application Development Guide*.

Example

```
// This procedure connects to the database and has

// two variables for the user and password.

proc logon

DBMS DECLARE c1 CONNECTION \
   WITH USER=user, PASSWORD=pword, \
   DATABASE="C:\Program Files\video\videobiz"
return

// Same example as above, but using FOR rather
// than WITH.  Note that the variable names are
// quoted and colon-expanded.

proc logon
DBMS DECLARE c1 CONNECTION \
   FOR USER ":user" PASSWORD ":pword" \
   DATABASE "C:\Progra~1\video\videobiz"
return
```

See Also

`CLOSE CONNECTION`, `CLOSE_ALL_CONNECTIONS`, `CONNECTION`, `WITH CONNECTION`, `dm_get_db_conn_handle`, `dm_is_connection`

DECLARE CURSOR

Declares a named cursor for an SQL statement

Synopsis

```
DBMS [WITH CONNECTION connection] DECLARE cursor CURSOR
    FOR SQLstatement
```

Arguments

`WITH CONNECTION connection`
: Name of connection to associate with the cursor. If the clause is not used, Panther opens the cursor on the default connection.

`cursor`
: Name of cursor to be created.

`SQLstatement`
: `SQL` statement to be performed when the cursor is executed.

Description

Use `DBMS DECLARE CURSOR` to create or redeclare a named cursor.

If the application has not already declared *`cursor`*, Panther allocates a new cursor structure and adds its name to the list of declared cursors.

If a cursor with the name *`cursor`* already exists and if the connection is the same, Panther reinitializes this cursor. Reinitialization clears any information on `SELECT` columns and binding parameters. It does not clear any attributes assigned to the cursor with the statements:

- `ALIAS`
- `CATQUERY`
- `COLUMN_NAMES`
- `FORMAT`
- `OCCUR`
- `START`
- `STORE`
- `TYPE`
- `UNIQUE`

Panther uses these settings if the cursor is redeclared with a `SELECT` statement. It ignores the attributes if the cursor is redeclared with an `INSERT`, `UPDATE`, or `DELETE` statement. To redeclare the cursor with a new (empty) structure, close the cursor with `CLOSE CURSOR` before executing the new declaration.

If a cursor is redeclared on a different connection, Panther automatically closes the cursor and declares a new structure.

A cursor remains open until it is explicitly closed with the `CLOSE CURSOR` command. Closing a connection also closes all cursors on the connection.

There are few restrictions on valid cursor names. However, avoid using any DBMS, JDB, or Panther keywords as a cursor name. Panther is case sensitive regarding cursor names; for example, it considers cursor `c1` as different from cursor `C1`.

For information on the format of parameters in the SQL statement, refer to Chapter 30, "Writing Information to the Database," in *Application Development Guide* and refer to "Using Database Cursors" on page 28-3 in *Application Development Guide* for information about declaring cursors.

Example

```
// When the following statement is executed, it fetches

// a list of actors in the specified video.

proc s_entry

DBMS WITH CONNECTION c1 DECLARE act_cursor CURSOR FOR \
   SELECT actors.first_name, actors.last_name, roles.role \
   FROM actors, roles \
   WHERE actors.actor_id = roles.actor_id \
   AND roles.title_id = ::film_code

proc exec1

DBMS WITH CURSOR t_cursor EXECUTE USING film_code

return
```

See Also

`CLOSE CURSOR`, `EXECUTE`, `WITH CURSOR`, `dm_is_cursor`

ENGINE

Sets or changes the default database engine

Synopsis `DBMS ENGINE` *`engine`*

Arguments *`engine`*

Name of default database engine when two or more engines are initialized. engine is the mnemonic assigned to the database engine in the file `dbiinit.c`, `prol5w32.ini`, `prol5w64.ini`, or `prol5unix.ini`.

Description If an application initializes two or more database engines, the application can use `DBMS ENGINE` to set a default engine. If an application has only one initialized engine, Panther automatically assigns that engine as the default.

For more information on initializing database engines, refer to Chapter 7, "Initializing the Database," in *Application Development Guide*.

Example

```
// This procedure declares two connections,
// sets oracle to be the default engine, and
// then declares and executes a cursor on the
// default engine.

proc entry

DBMS WITH ENGINE oracle DECLARE c1 CONNECTION FOR \
   USER ":user" PASSWORD ":pword"

DBMS WITH ENGINE sybase DECLARE c2 CONNECTION FOR \
   USER ":user" PASSWORD ":pword" SERVER "maple" \
   DATABASE "sales"

DBMS ENGINE oracle

DBMS DECLARE t_cursor CURSOR FOR SELECT * FROM titles

DBMS WITH CURSOR t_cursor EXECUTE

return
```

See Also `WITH ENGINE`, `dm_is_engine`

EXECUTE

Executes the SQL statement declared for a named cursor

Synopsis `DBMS [WITH CURSOR *cursor*] EXECUTE [USING *args*]`

Arguments

`WITH CURSOR` *`cursor`*
Name of declared `SELECT` cursor. If the clause is not used, Panther uses the default `SELECT` cursor.

`args`
Panther variables containing parameter values.

Description

Use `DBMS EXECUTE` to execute the statement associated with a declared cursor.

`DBMS EXECUTE` does not support the `WITH CONNECTION` clause. Panther uses the connection that was specified either by name or by default when the cursor was declared. The only way to change the cursor's engine or connection is to redeclare the cursor.

If an application is executing a similar statement many times, it is often more efficient to declare a cursor for the statement. Usually the database engine saves the parsed statement, executing it when the application executes the cursor. It is not necessary to redeclare the cursor to supply new data for a `WHERE` or `VALUES` clause. Instead, the application can declare the cursor and use a substitution parameter for each value that the application supplies when it executes the cursor. Substitution parameters begin with a double colon (`::`). For example:

```
DBMS DECLARE c1 CURSOR FOR \
   SELECT * FROM titles WHERE name LIKE ::name_parm
```

`name_parm` is a place holder for the value that will be supplied when the cursor is executed. For example:

```
DBMS WITH CURSOR c1 EXECUTE USING "St%"
```

This command fetches rows where name begins with the characters `"St"`. The application could execute the cursor repeatedly, each time with a new value. It can use the value of a field to supply a value. For example:

```
DBMS WITH CURSOR c1 EXECUTE USING aname
```

Since `aname` is not quoted, Panther assumes it is a Panther variable. If an argument in the `USING` clause is a widget or LDB variable that has date/time, currency, null field, or type property specifications, Panther formats the variable's value before passing it to the database engine. Refer to Chapter 30, “Writing Information to the Database,” in *Application Development Guide* for details of this topic.

Example

```
DBMS DECLARE x CURSOR FOR \
   SELECT * FROM tapes WHERE title_id=::p1 AND copy_num=::p2

// bind by position:

DBMS WITH CURSOR x EXECUTE USING code, copy_id

// or bind by name:

DBMS WITH CURSOR x EXECUTE \
   USING p1 = code, p2 = copy_id
```

See Also `DECLARE CURSOR`, `CLOSE CURSOR`, `WITH CURSOR`

FORMAT

Formats CATQUERY values

Synopsis

```
DBMS [WITH CURSOR cursor] FORMAT [ [ column ] format
    [, [ column] format ... ] ]
```

Arguments

`WITH CURSOR` *cursor*

Name of declared `SELECT` cursor. If the clause is not used, Panther uses the default `SELECT` cursor.

column

Name of selected column. The case of column should match the setting of the case flag for the database engine. If columns are not named, the formats are applied by position.

format

Specifies how Panther should format the value. *format* is either a Panther variable or a quoted precision edit.

Description

Use `DBMS FORMAT` to format `CATQUERY` values before writing them to a variable or a text file. The options are explained below.

If *format* is a Panther variable, Panther formats the column value as if it were writing to the field. In particular, the following characteristics affect the formatting:

- Variable's maximum shifting length
- Variable's Panther type

For more information about formatting select results, refer to "Format of Select Results" on page 29-15 in *Application Development Guide*.

format can also be a precision edit. A precision edit is a quoted string beginning with a percent sign. It supplies the length of the value, and optionally, a decimal precision for numeric values.

A precision is given in the form:

"%*width*"

"%*width.precision*"

To turn off formatting on the default or named cursor, execute `DBMS FORMAT` with no arguments.

Example

```
// use column "title_id" and "copy_num" exactly as returned
// format column "due_back" with the LDB variable "today",
// format column "price" to width 5 with 2 decimal places
// format column "rental_comment" to width 25 and truncate.

proc tapes_due
DBMS CATQUERY TO FILE rentlist
DBMS FORMAT due_back today, price "%5.2", \
   rental_comment "%25"
DBMS QUERY SELECT title_id, copy_num, due_back, price, \
   rental_comment FROM rentals
return
```

OCCUR

Changes the behavior of a `SELECT` *cursor that writes to Panther arrays*

Synopsis

`DBMS [WITH CURSOR` *`cursor`*`] OCCUR` *`occInt`* `[MAX` *`int`*`]`

`DBMS [WITH CURSOR` *`cursor`*`] OCCUR CURRENT [MAX` *`int`*`]`

Arguments

`WITH CURSOR` *`cursor`*
: Name of declared `SELECT` cursor. If the clause is not used, Panther uses the default `SELECT` cursor.

`occInt`
: Occurrence number where Panther should begin placing `SELECT` results.

`CURRENT`
: Specifies that Panther use the occurrence number of the “current” field. Panther begins writing at this occurrence number in the target arrays. The current field is the one containing the Panther screen cursor and is not necessarily a target variable.

`MAX` *`int`*
: Specify maximum number of rows to fetch for a `SELECT` or `CONTINUE`. If *`int`* is less than 1, no rows are fetched.

Description

By default, if the destination of a `SELECT` is one or more arrays, Panther fetches as many rows as will fit in the arrays and begins writing at the first occurrence in the arrays. `DBMS OCCUR` changes this default behavior for a `SELECT` cursor.

The setting is turned off by executing the `DBMS OCCUR` command with no arguments. Closing a cursor also turns off the setting. If a cursor is redeclared without being closed, the cursor continues to use the setting for `SELECT` statements and `CONTINUE` commands.

`DBMS OCCUR` is ignored with a `CATQUERY` cursor.

Example

```
// When executed, this procedure starts writing
// the result set at the current occurrence.

proc select_type
DBMS DECLARE genre_cursor CURSOR FOR \
   SELECT * FROM titles WHERE genre_code = :+code
```

```
DBMS WITH CURSOR title_cursor OCCUR CURRENT
DBMS WITH CURSOR title_cursor EXECUTE
return
```

ONENTRY

Installs an entry handler

Synopsis

```
DBMS ONENTRY CALL function
DBMS ONENTRY JPL jplEntryPoint
DBMS ONENTRY STOP
```

Arguments

`CALL` *`function`*
: Name of prototyped function.

`JPL` *`jplEntryPoint`*
: Name of JPL procedure.

`STOP`
: Remove any installed entry handler.

Use `DBMS ONENTRY` to install a function or JPL procedure which Panther calls before it executes each `DBMS` statement.

This function is for informational purposes only. For instance, you can log statements to a text file before executing them. You can use this function with an exit function to track the start and end time for a query or any other database operation.

The function is passed three arguments:

1. A copy of the first 255 characters of the statement; if the statement is executed from JPL, this is the first 255 characters after the command word `DBMS` or `DBMS SQL`.
2. The name of the database engine for the statement.
3. Context flag; for the entry handler its value is 0.

The function's return code is not used.

Example

The following sample function logs the current statement in a text file.

```
/* This function is installed as a prototyped function.*/
/* It writes the current time, name of the current */
/* engine, and the command which Panther will execute */
/* to a file called dbi.log. */
/* dbms ONENTRY CALL dbientry  */

#include <smdefs.h>

int
dbientry (stmt, engine, flag)
char *stmt;
char *engine;
int flag;

{
   FILE *fp;
   time_t timeval;

   fp = fopen ("dbi.log", "a");
   timeval = time(NULL)
   fprintf (fp, "%s\n%s\n%s\n\n",
             ctime(&timeval), engine, stmt);
   fclose (fp);
   return 0;
}
```

This sample function displays a message before performing any database operations.

```
// dbms ONENTRY JPL entrymsg

proc entrymsg
   msg setbkstat "Processing. Please be patient..."
   flush
   return 0
```

See Also

`ONEXIT`, Chapter 12, “DBMS Global Variables,” Chapter 37, “Processing Application Errors,” in *Application Development Guide*

ONERROR

Sets the behavior of the error handler

Synopsis

```
DBMS ONERROR CALL function

DBMS ONERROR CONTINUE

DBMS ONERROR JPL jplEntryPoint

DBMS ONERROR STOP
```

Arguments

`CALL` *`function`*
: Name of prototyped C function.

`JPL` *`jplEntryPoint`*
: Install a user function as the error handler. If Panther or the database engine find an error, Panther updates the global error and status variables (the `@dm` variables) and calls the installed function.

    `function`
    : Name of prototyped C function.

    `jplEntryPoint`
    : Name of JPL procedure.

`CONTINUE`
: Prevents default error handler from aborting a JPL procedure where a Panther error occurs. Message display is not changed.

`STOP`
: Restores default error handler.

Description

Use `DBMS ONERROR` to set or change the behavior of the Panther database error handler for the application. The default error handler displays the following:

- Statement which caused the error.
- Source of the message. If the database engine generated the message, only the engine name is listed. If Panther's database driver generated the message, database interface is listed along with the engine name.
- Error code number from Panther's database driver or from the database engine.

- Error message from Panther's database driver or from the database engine.

If an error occurs while executing a JPL procedure, the default handler aborts the procedure, returning -1 to the calling procedure.

An application can override the default error handler with the command `DBMS ONERROR` and an argument. The error handler is global to the application. Each execution of this command overrides the previous error handler.

The function displays any error messages and its return code controls whether or not JPL execution is aborted.

The function is passed three arguments:

1. The first 255 characters of the statement; if the statement was executed from JPL, this is the first 255 characters after the command word `DBMS` or `DBMS SQL`.
2. The name of the database engine for the statement.
3. Context flag; for the error handler its value is 2.

The function's return code is returned to the application. If an `ONEXIT` function and an `ONERROR` function are both installed, the return code from the `ONERROR` function takes precedence.

If the error occurred while executing a JPL statement with a DBMS command:

- 0 returns control to the JPL procedure where the error occurred.
- 1 aborts the JPL procedure where the error occurred and returns 1 to the procedure’s caller (either Panther or another JPL procedure).

If the error occurred while executing a statement with the `dm_dbms` library function, the function returns the error handler's return code.

To use a C function as an error handler, you must first install the function as a prototyped function. Refer to “Prototyped Functions” on page 44-8 in *Application Development Guide* for more information on prototyped functions.

Example

```
//Error handler installed in JPL.

proc entry
DBMS ONERROR JPL dbi_err
return

proc dbi_err (stmt, engine)
if @dmengerrcode == 0
```

```
    msg emsg stmt "%N" "Panther error: " @dmretmsg
else
    msg emsg stmt "%N" "Panther error: " @dmretmsg "%N"\
        ":engine error: " @dmengerrcode " " @dmengerrmsg
return
```

The next example first checks to see if the Panther error is `DM_ALREADY_ON`. In this case, it simply displays a message and returns 0. For all other errors, it checks for an engine error code. If there is an engine error, it calls another subroutine to check for engine-specific errors. For any other errors, it displays the standard Panther message.

```
proc entry
DBMS ONERROR JPL dbi_error_handler
return

proc dbi_error_handler (stmt, engine, flag)

    if (@dmretcode == DM_ALREADY_ON)
    {
        msg emsg "You are already logged on."
        return 0
    }

    if (@dmengerrcode != 0)
    {
        msg emsg @dmretmsg
        call engine_errors (engine)
    }
    else
    {
        msg emsg "Application Error:  " \
            @dmretmsg \
            "See the DBA for assistance."
    }
    return 1

proc engine_errors (engine_name)
    if engine_name == "xyzdb"
    ...
// Examine DBMS error codes here.
```

See Also

`ONENTRY`, `ONEXIT`, Chapter 12, “DBMS Global Variables,” Chapter 37, “Processing Application Errors,” in *Application Development Guide*

ONEXIT

Installs an exit handler

Synopsis

```
DBMS ONEXIT CALL function

DBMS ONEXIT JPL jplEntryPoint

DBMS ONEXIT STOP
```

Arguments

`CALL` *`function`*
: Name of prototyped C function.

`JPL` *`jplEntryPoint`*
: Name of JPL procedure.

`STOP`
: Remove any installed exit function.

Use `DBMS ONEXIT` to install a function which Panther calls after executing a DBMS command from JPL or C. Use this function to process error and status codes after every command.

The exit handler is global to the application. Each execution of `DBMS ONEXIT` overrides the previous exit handler.

The function is passed three arguments:

1. The first 255 characters of the statement. If the statement was executed from JPL, this is the first 255 characters after the command word `DBMS` or `DBMS SQL`.
2. The name of the database engine for the statement.
3. Context flag; for the exit handler its value is 1.

The function's return code is returned to the application. If an `ONEXIT` function and an `ONERROR` function are both installed, the return code from the `ONERROR` function takes precedence.

If an error occurred while executing a JPL statement with a DBMS command and there is no `ONEXIT` function, then

- 0 returns control to the JPL procedure where the error occurred.

- 1 aborts the JPL procedure where the error occurred and returns 1 to the procedure's caller (either Panther or another JPL procedure).

If the error occurred while executing a statement with the `dm_dbms` library function and there is an `ONEXIT` function, the function returns the exit handler's return code.

To use a C function as an exit handler, you must first install the function as a prototyped function. For more information, refer to "Prototyped Functions" on page 44-8 in *Application Development Guide*.

Example

```
//JPL procedure processes warnings from the database engine.

proc entry
DBMS ONEXIT JPL dbi_warn
return

proc dbi_warn (stmt, engine, flag)
if @dmengerrcode == 0
   msg emsg stmt "%N" "Error: " @dmretmsg
else
   msg emsg stmt "%N" "Error: " @dmretmsg "%N" \
      ":engine error: " @dmengwarncode " " @dmengwarnmsg
return
```

See Also

`ONENTRY`, `ONERROR,` Chapter 12, "DBMS Global Variables," Chapter 37, "Processing Application Errors," in *Application Development Guide*

QUERY

Executes an SQL statement that returns one or more select sets

Synopsis

`DBMS [WITH CONNECTION` *`connection`*`] QUERY` *`SQLstatement`*

Arguments

`WITH CONNECTION` *`connection`*
: Name of connection to associate with the statement. If the clause is not used, Panther uses the default connection.

`SQLstatement`
: SQL statement to be sent to the database engine. The syntax of the statement can be the specific to the database engine.

Description

`DBMS QUERY` prepares *`SQLstatement`* (in general, one that returns rows) by indicating to the database engine where to put return data (if any), and then tells the database to execute the SQL statement.

This command is used for statements, such as `SELECT` statements and stored procedures, that return select data. For statements that do not return data, use `RUN`.

For additional information, refer to Chapter 29, "Reading Information from the Database," in *Application Development Guide*.

Example

```
DBMS QUERY SELECT title_id, name, dir_first_name, \
   dir_last_name FROM titles
```

See Also

`RUN`

RUN

Executes an SQL statement that does not return any select sets

Synopsis `DBMS [WITH CONNECTION` *`connection`*`] RUN` *`SQLstatement`*

Arguments `WITH CONNECTION` *`connection`*

Name of connection to associate with the statement. If the clause is not used, Panther uses the default connection.

`SQLstatement`

SQL statement (`INSERT`, `UPDATE`, `DELETE`, `CREATE TABLE`, etc.) to be sent to the database engine. The syntax of the statement can be specific to database engine.

Description `DBMS RUN` executes *`SQLstatement`* on the assumption that no data will be returned by submitting the SQL to the database for immediate execution.

If, however, a data selection SQL statement is executed (one which returns data from the database), the statement will be executed a second time in order that the selected data can be fetched. For this reason, use the `QUERY` statement when selected data may be returned.

Example

```
DBMS RUN INSERT INTO actors \
   (actor_id, last_name, first_name) VALUES \
   (:+actor_id, :+last_name, :+first_name)
```

See Also `QUERY`

SQL

Executes an SQL statement

Synopsis `DBMS [WITH CONNECTION` *`connection`*`] SQL` *`SQLstatement`*

Arguments

`WITH CONNECTION` *`connection`*
: Name of connection to associate with the statement. If the clause is not used, Panther uses the default connection.

`SQLstatement`
: SQL statement to be sent to the database engine. The syntax of the statement can be the format needed by your database engine.

Description

This command is not recommended. For SQL statements that return data, use `QUERY`. For SQL statements that do not return data (such as `INSERT`, `UPDATE` and `DELETE`), use `RUN`.

`DBMS SQL` sends the specified statement to the database engine for execution after colon expansion is performed. If a connection is not specified, Panther uses the default cursor on the default connection.

`SQLstatement` can be in the format needed by your database engine. This allows you to access all the features of your database engine.

Example

```
DBMS SQL SELECT title_id, name, dir_first_name, \
    dir_last_name FROM titles
```

```
DBMS SQL INSERT INTO actors \
    (actor_id, last_name, first_name) VALUES \
    (:+actor_id, :+last_name, :+first_name)
```

See Also `QUERY`, `RUN`

START

Specifies a starting row in a `SELECT` set

Synopsis `DBMS [WITH CURSOR` *`cursor`*`] START [`*`int`*`]`

Arguments

`WITH CURSOR` *`cursor`*
: Name of declared `SELECT` cursor. If the clause is not used, Panther uses the default `SELECT` cursor.

`int`
: Number indicating the row at which to begin the fetch.

Description By default, when a select set contains more than one row, Panther fetches them sequentially beginning with the first row in the select set. Use `DBMS START` to begin fetching at row *`int`*. Panther fetches and discards *`int`* - 1 rows from the select set before returning the requested rows to the application. The discarded rows do not update `@dmrowcount`. If *`int`* is greater than the number of rows in the select set, no rows are fetched.

The setting is turned off by executing `DBMS START` with no arguments. Closing a cursor also turns off the setting. If a cursor is redeclared without being closed, the cursor continues to use the setting for `SELECT` statements.

Example

```
proc discard_100
DBMS START 100
DBMS QUERY SELECT * FROM actors
if @dmrowcount == 0
   msg emsg "There are fewer than 100 rows."
DBMS START
return
```

STORE

Sets up a continuation file for a named or default cursor

Synopsis `DBMS [WITH CURSOR` *`cursor`*`] STORE [FILE [`*`filename`*`]]`

Arguments `WITH CURSOR` *`cursor`*

Name of declared `SELECT` cursor. If the clause is not used, Panther uses the default `SELECT` cursor.

`filename`

Name of temporary binary file.

Description When `DBMS STORE` is used with a `SELECT` cursor, Panther maintains a copy of the result rows in a temporary binary file. The file permits an application to scroll forward and backward in a select set, even if the database has no native support for backward scrolling.

A continuation file remains open for the life of the cursor, or until the feature is turned off with the command,

```
DBMS [WITH CURSOR cursor] STORE
```

Executing the command without the keyword `FILE` closes and deletes the file and turns off the feature for the named or default cursor. Closing the cursor also closes and deletes the file. If a cursor is not closed but simply redeclared for another `SELECT` statement, the file is cleared. Therefore, a continuation file holds the results of one `SELECT` statement only.

The use of a continuation file does not force the database engine to return the entire select set when the `SELECT` is executed. Panther examines the number of occurrences in the destination variable to determine the number of rows to fetch. Each time it fetches rows from the database engine by executing the `SELECT` or a `DBMS CONTINUE`, Panther updates the screen and appends the new data to the continuation file. If the application wishes to see rows already fetched, Panther uses the continuation file to get the rows and update the screen. If Panther reaches the end of the continuation file and the application executes another `DBMS CONTINUE`, Panther attempts to get more rows from the database engine. When the engine returns the no-more-rows code, Panther sets `@dmretcode` to the value of `DM_NO_MORE_ROWS`. Similarly, if the application attempts to scroll back past the first row in the continuation file, Panther sets `@dmretcode` to `DM_NO_MORE_ROWS`. Write errors are not reported.

`DBMS STORE` provides several advantages:

- A means for displaying very large select sets without keeping all rows in memory at once.
- Better response time for very large select sets; since fetches are incremental, it is not necessary to get the entire select set at once.
- A means for forcing an database engine to release a shared lock on a large select set.

For information on engine-specific scrolling issues, refer to "Database Drivers."

Example

```
// Use of STORE FILE with JPL procedures to fetch more rows.

proc title_select
    DBMS DECLARE t_cursor CURSOR FOR SELECT * FROM titles
    DBMS WITH CURSOR t_cursor STORE FILE
    DBMS WITH CURSOR t_cursor EXECUTE
    return

proc get_next
    DBMS WITH CURSOR t_cursor CONTINUE_DOWN
    return

proc get_previous
    DBMS WITH CURSOR t_cursor CONTINUE_UP
    return

// Use of STORE FILE and map keys in order to fetch more rows.

proc select_titles
    DBMS DECLARE t_cursor CURSOR FOR SELECT * FROM titles
    DBMS WITH CURSOR t_cursor STORE FILE
    DBMS WITH CURSOR t_cursor EXECUTE
    return

// This procedure is called on screen entry.

proc entry (name, flag)
if (flag & K_ENTRY)
{
    call sm_keyoption (SPGD, KEY_XLATE, APP1)
    call sm_keyoption (SPGU, KEY_XLATE, APP2)
}
```

```
...
return

//This procedure is called on screen exit.

proc exit (name, flag)
if (flag & K_EXIT)
{
   call sm_keyoption (SPGU, KEY_XLATE, SPGU)
   call sm_keyoption (SPGD, KEY_XLATE, SPGD)
}
...
return

proc scroll_up
// Control strings contains:
// APP1 = ^scroll_up
   DBMS WITH CURSOR t_cursor CONTINUE_UP
   return

proc scroll_down
// Control strings contains:
// APP2 = ^scroll_down
   DBMS WITH CURSOR t_cursor CONTINUE
   return
```

See Also `CONTINUE`

UNIQUE

Suppresses repeating values in selected columns

Synopsis `DBMS [WITH CURSOR` *`cursor`*`] UNIQUE` *`column`* `[,` *`column`* `...]`

Arguments `WITH CURSOR` *`cursor`*

Name of declared `SELECT` cursor. If the clause is not used, Panther uses the default `SELECT` cursor.

`column`

Column name in the `SELECT` statement.

Description `DBMS UNIQUE` suppresses repeating values in each named column of a select set when the values are in adjacent rows. Typically, this feature is set for a column named in an `ORDER BY` clause.

If the destination variable has a null edit, an occurrence containing a suppressed value is blank, not null.

The setting is turned off by executing the `DBMS UNIQUE` command with no arguments. Closing a cursor also turns off the setting. If a cursor is redeclared without being closed, the cursor continues to use to the setting for `SELECT` statements and `CONTINUE` commands.

Example

```
// Since several titles can be rented to the same customer,
// suppress repeating customer numbers when listing
// outstanding rentals.

proc rent_list

DBMS DECLARE rent_cursor CURSOR FOR \
    SELECT cust_id, title_id, copy_num, due_back FROM rentals \
    WHERE due_back < today \
    ORDER BY cust_id

DBMS WITH CURSOR rent_cursor UNIQUE cust_id
DBMS WITH CURSOR rent_cursor EXECUTE
DBMS WITH CURSOR rent_cursor UNIQUE
return
```

WITH CONNECTION

Uses a named connection for the duration of a statement

Synopsis

`DBMS WITH CONNECTION` *`connection DBMSstatement`*`...`

Arguments

`WITH CONNECTION` *`connection`*

Specifies connection for the execution of the command, overriding the default connection. connection must be declared and open. Colon expansion on the connection name is allowed.

`DBMSstatement`

Text of the DBMS command.

Description

The most frequent use of the `DBMS WITH CONNECTION` clause is in a `DECLARE CURSOR` statement.

```
DBMS WITH CONNECTION connection DECLARE cursor CURSOR...
```

Once a cursor is declared it remains associated with the connection on which it was declared. After declaring the cursor, the `DBMS WITH CONNECTION` clause must not be used in statements that manipulate the cursor. However, the `DBMS WITH CONNECTION` clause can be used on statements that manipulate the default cursor on any declared connection. Therefore, the following commands

```
DBMS WITH CONNECTION connection ALIAS ...
DBMS WITH CONNECTION connection CATQUERY ...
DBMS WITH CONNECTION connection CLOSE CURSOR
DBMS WITH CONNECTION connection COLUMN_NAMES
DBMS WITH CONNECTION connection CONTINUE
DBMS WITH CONNECTION connection CONTINUE_BOTTOM
DBMS WITH CONNECTION connection CONTINUE_TOP
DBMS WITH CONNECTION connection CONTINUE_UP
DBMS WITH CONNECTION connection FORMAT ...
DBMS WITH CONNECTION connection OCCUR ...
DBMS WITH CONNECTION connection QUERY ...
DBMS WITH CONNECTION connection RUN ...
DBMS WITH CONNECTION connection SQL ...
DBMS WITH CONNECTION connection START ...
DBMS WITH CONNECTION connection STORE ...
DBMS WITH CONNECTION connection UNIQUE ...
```

perform the request on the default `SELECT` cursor on the named connection.

Some engine-specific DBMS commands can also support the `WITH CONNECTION` clause. For more information, refer to “Database Drivers.”

Example

```
// This procedure performs a commit before closing the
// connection.

proc cleanup (connection)
DBMS WITH CONNECTION :connection COMMIT
DBMS CLOSE CONNECTION :connection
return 0
```

See Also `DECLARE CONNECTION`, `dm_is_connection`

WITH CURSOR

Uses a named cursor for the duration of a statement

Synopsis `DBMS WITH CURSOR` *`cursor DBMSstatement`*

Arguments `WITH CURSOR` *`cursor`*

Name of declared `SELECT` cursor. If the clause is not used, Panther uses the default `SELECT` cursor. Colon expansion of the cursor name is allowed.

`DBMSstatement`

Text of the DBMS command.

Description The `DBMS WITH CURSOR` clause specifies the name of a declared cursor on which Panther executes the DBMS command. Once a cursor is declared, the application can manipulate or execute the cursor by using the `WITH CURSOR` clause with the following commands:

```
DBMS WITH CURSOR cursor ALIAS ...
DBMS WITH CURSOR cursor CATQUERY ...
DBMS WITH CURSOR cursor COLUMN_NAMES
DBMS WITH CURSOR cursor CONTINUE
DBMS WITH CURSOR cursor CONTINUE_BOTTOM
DBMS WITH CURSOR cursor CONTINUE_TOP
DBMS WITH CURSOR cursor CONTINUE_UP
DBMS WITH CURSOR cursor EXECUTE ...
DBMS WITH CURSOR cursor FORMAT ...
DBMS WITH CURSOR cursor OCCUR ...
DBMS WITH CURSOR cursor START ...
DBMS WITH CURSOR cursor STORE ...
DBMS WITH CURSOR cursor UNIQUE ...
```

If the `DBMS WITH CURSOR` clause is not used with these commands, Panther uses the default SELECT cursor. The application can also manipulate the default cursor by using the `DBMS WITH CONNECTION` clause.

Some engine-specific DBMS commands can also support the `WITH CURSOR` clause. For more information, refer to "Database Drivers."

Example

```
// Uses colon expansion on the cursor name to remove
// the command attributes for named cursors.

proc cursor_refresh (cursor_name)
DBMS WITH CURSOR :cursor_name ALIAS
DBMS WITH CURSOR :cursor_name CATQUERY
return 0
```

See Also `DECLARE CURSOR`, `CLOSE CURSOR`, `dm_is_cursor`

WITH ENGINE

Uses a named database engine for the duration of a statement

Synopsis `DBMS WITH ENGINE` *`engine command`*`...`

Arguments

`WITH ENGINE`
: Name of engine to associate with the command. If the clause is not specified, Panther uses the default engine.

`engine`
: Mnemonic associated with the engine when you make your Panther executables. Colon expansion of the engine name is allowed.

`command`
: Text of the DBMS command.

Description The `DBMS WITH ENGINE` clause specifies which database engine Panther should use when executing a command. If only one database engine is initialized, that engine is automatically the default. An application using two or more engines can set the default engine with the `DBMS ENGINE` command.

The following commands accept an optional `WITH ENGINE` clause:

```
DBMS WITH ENGINE engine DECLARE connection CONNECTION ...
DBMS WITH ENGINE engine CLOSE_ALL_CONNECTIONS
```

Once a connection is declared, it remains associated with the database engine on which it was declared. After declaring the connection, the `WITH ENGINE` clause is no longer necessary or valid in any statement except for `DBMS CLOSE_ALL_CONNECTIONS` which allows you to close the connections for the default or named engine.

Example Refer to "Connecting to Multiple Engines" on page 8-5 in *Application Development Guide*.

See Also `ENGINE`, `dm_is_engine`

CHAPTER

12 DBMS Global Variables

This chapter describes the global variables, in alphabetical order, available in Panther's database drivers.

A reference page for each global variable includes:

- A description of the variable.
- A list of related variables and commands.
- An example.

Since some variables store engine-specific values, refer to “Database Drivers” for additional information. For more information on using the global variables as part of your error processing, refer to Chapter 37, “Processing Application Errors,” in *Application Development Guide*.

Variable Overview

The global variables available through Panther's database drivers are automatically defined at initialization. All the global variable names used in the database drivers begin with the characters `@dm`. Since the character `@` is not permitted in user-defined Panther variables, these variables will never conflict with any screen, LDB or JPL variables defined by your application.

These variables and their values are available to JPL commands and to Panther library functions like `sm_getfield` and `sm_fptr`.

The variables are automatically maintained by Panther. Before executing a `DBMS` command, Panther clears the contents of all the DBMS global variables. After executing the command and before returning control to the application, Panther updates the variables to indicate the current status.

Table 12-1 Error Data

Variable	Description
`@dmretcode`	Error code from Panther's database driver. Codes are the same for all engines.
`@dmretmsg`	Error message from Panther's database driver. Messages are the same for all engines.
`@dmengerrcode`	Engine error code. Codes are unique to the engine.
`@dmengerrmsg`	Engine error message. Messages are unique to the engine. Some engines do not supply messages.
`@dmerrsqlstate`	Engine status code signaling an error condition. Not used by all engines.

Table 12-2 Status Data

Variable	Description
`@dmretcode`	Status code available for “no more rows” or “end of procedure.”
`@dmretmsg`	Status message available for “no more rows” or “end of procedure.”
`@dmengreturn`	Engine return code from a stored procedure. Not used by all engines.
`@dmrowcount`	Count of the number of rows fetched to Panther by the last `SELECT` or `CONTINUE`. Used by all engines.
`@dmserial`	A serial value returned after inserting a row into a table with a serial column. Not used by all engines.
`@dmengwarncode`	A code or byte signalling a non-fatal error or unusual condition. Not used by all engines.
`@dmengwarnmsg`	A message corresponding to an engine warning code. Not used by all engines.
`@dmwarnsqlstate`	Engine status code signaling a warning condition. Not used by all engines.

@dmengerrcode

Contains an engine-specific error code

Description

`@dmengerrcode` is set to `0` before executing a `DBMS` command. If the database engine detects an error, Panther writes the engine's error code to this variable. In cases where a database engine can generate multiple error codes for one statement, `@dmengerrcode` is an array, and each error code is written to a different occurrence.

A `0` (zero) value in this variable does not guarantee that the last statement executed without error. Some errors are detected by Panther's database driver before a request is made to the engine. For example, if an application attempts a `SELECT` before declaring a connection, Panther detects the error. Use the global variable `@dmretcode` to check for errors in Panther's database drivers.

Because the value of `@dmengerrcode` is engine-specific, it is strongly recommended that you install an error handler to test for these errors. In a multi-engine application, the error handler can call another function to do this depending on the engine.

If the default error handler is in use, Panther displays the statement which failed and an error message from either Panther's database driver or from the database engine. If the application has installed its own error handler, the installed function controls what messages are displayed. Refer to the *Database Drivers* for more information about the codes for a particular engine.

Example

```
proc dbi_errhandle (stmt, engine, flag)
if @dmengerrcode == 0
msg emsg @dmretmsg
else if engine == "xyzdb"

   call xyzerror (@dmengerrcode)
else if engine == "oracle"
   call oraerror (@dmengerrcode)
else
   msg emsg "Unknown engine."
return 1

proc xyzerror (error)
# Check for specific xyzdb error codes.
if error == 90931
   msg emsg "Invalid user name."
else if error == ...
   ...
else
```

```
        msg emsg @dmengerrmsg
    return
```

@dmengerrmsg

Contains an engine-specific error message

Description

`@dmengerrmsg` is set to `""` before executing a `DBMS` command. If the database engine returns an error message after attempting to execute the command, Panther writes the message to this variable. If a database engine can generate multiple error messages for one command, `@dmengerrmsg` is an array, and each error message is written to a different occurrence.

If `@dmengerrcode` is `0`, this variable contains no message. It will also be `0` if the engine does not supply error messages. Refer to the *Database Drivers* for more information about the availability of this variable.

Example

```
proc dbi_errhandle (stmt, engine, flag)
if @dmengerrcode == 0
   msg emsg @dmretmsg
else
   msg emsg @dmretmsg "%N" @dmengerrmsg
return 1
```

@dmengreturn

Contains a return code from a stored procedure

Description

Use `@dmengreturn` to get a stored procedure's return or status code. This variable is only available if your engine supports stored procedures and their return codes and if the Panther database driver supports stored procedure return codes.

Since database engines implement stored procedures differently, refer to the *Database Drivers* for engine-specific information and examples.

By default, Panther pauses the execution of a stored procedure if the procedure executes a `SELECT` statement and the number of rows in the select set is greater than the number of occurrences in the Panther destination variables. The application must execute `CONTINUE` or `DBMS NEXT` to resume execution. If the value of `@dmengreturn` is null after calling a stored procedure, the procedure might be pending. If the engine has completed the execution of the procedure, `@dmretcode` will contain the `DM_END_OF_PROC` code and `@dmengreturn` will contain the procedure's return code.

The value of this variable is cleared once another `DBMS` command is executed. If the application needs the value for a longer period of time, it should copy it to a standard Panther variable or some other static location.

Example

```
# This is an example of a SYBASE stored procedure:
# create proc checkid @id int as
# if (SELECT COUNT (*) FROM titles WHERE title_id = @id) = 1
#   return 1

# else
#   return 2


DBMS RUN EXEC checkid :+title_id
if @dmengreturn == 1
   call addrow

else if @dmengreturn == 2
   msg emsg "Sorry, " title_id " is not a valid code."
return

proc addrow
DBMS RUN INSERT INTO tapes VALUES \
   (:+title_id, :+copy_num, 'O', 0)
return
```

@dmerrsqlstate

Contains an engine-specific status code for error conditions

Description

Some database engines support a `SQLSTATE` status code which is updated after each SQL statement. `SQLSTATE` is currently supported by Panther's ODBC, Informix and DB2 drivers.

`SQLSTATE` is a five-character string which can be set for warning or error conditions. In Panther, warning conditions from `SQLSTATE` are written to `@dmwarnsqlstate`; error conditions are written to `@dmerrsqlstate`.

If the database engine does not support `SQLSTATE`, the value of `@dmerrsqlstate` will be `"00000"`, the value that represents success.

A `"00000"` (five zeros) value in this variable does not guarantee that the last statement executed without error. Some errors are detected by Panther's database driver before a request is made to the engine. For example, if an application attempts a `SELECT` before declaring a connection, Panther detects the error. Use the global variable `@dmretcode` to check for errors in Panther's database drivers.

Because the value of `@dmerrsqlstate` is engine-specific, it is strongly recommended that you install an error handler to test for these errors.

If the application accesses multiple database engines, the database driver for each engine must support `SQLSTATE` in order to use its values for application processing. `@dmerrsqlstate` is set to `"00000"` before each `DBMS` statement. If you need its value for later processing, it should be copied to another variable.

@dmengwarncode

Contains an engine-specific warning code

Description

Most engines supply a mechanism for signalling an unusual, but non-fatal condition.

Some engines use an eight-element array. If there is a warning, it sets the first element to indicate a warning and then sets one or more additional elements to describe the warning. Other engines use codes and messages similar to those used for errors. Those of a high severity are handled as errors and those of a low severity are handled as warnings. Refer to the *Database Drivers* for engine-specific information and examples.

By default, Panther ignores warnings. If an application needs to alert users to warning codes, it must use a JPL or C function to check for them. There is no default warning handler. The most efficient way to process warning codes is with an installed exit handler using `ONEXIT`.

@dmengwarnmsg

Contains an engine-specific warning message

Description

Most engines supply a mechanism for signalling an unusual, but non-fatal condition. Some engines uses a warning array or byte. These engines do not supply warning messages and therefore do not use `@dmengwarnmsg`. Others use a code and message for low-severity errors. Refer to the *Database Drivers* for engine-specific information and an example.

By default, Panther ignores warnings. If an application needs to alert users to warning codes or messages, it must use a JPL or C function to check for them. There is no default warning handler. The most efficient way to process warning values is with an installed exit handler using `ONEXIT`.

@dmretcode

Contains an engine-independent error or status code

Description

`@dmretcode` is set to `0` before Panther executes a new `DBMS` command. If the command fails because of an error detected either by the engine or by Panther's database driver, Panther writes an error code to `@dmretcode` describing the failure.

Usually a non-zero value in `@dmretcode` indicates that an error occurred. The default or an installed error handler is called for an error. If the default handler is in use, Panther displays the statement which failed and an error message from either Panther's database driver or from the database engine. If the application has installed its own error handler, the installed function controls what messages are displayed.

There are two non-zero codes for `@dmretcode` which are not errors: `DM_NO_MORE_ROWS` and `DM_END_OF_PROC`. When an engine indicates that it has returned all rows for a select set, Panther writes the `DM_NO_MORE_ROWS` code to `@dmretcode`. Since this is not considered an error, Panther does not call the default or installed error handler. You can test for `DM_MORE_ROWS` after executing a `SELECT` or in an exit handler.

Panther uses `DM_END_OF_PROC` with engines that support stored procedures. When an engine indicates that it has completed executing the stored procedure, Panther writes the `DM_END_OF_PROC` code to `@dmretcode`. This is not an error. An application can test for this code in an exit procedure or after calling a stored procedure. Refer to the *Database Drivers* for information on stored procedures.

The values for `@dmretcode` are listed alphabetically in Table 12-3 (in the source code, they reside in `dmerror.h`).

Table 12-3 @dmretcode error codes and corresponding messages

Code constant	Message
`DM_ABORTED`	Processing aborted due to DB error.
`DM_ALREADY_INIT`	Engine already installed.
`DM_ALREADY_ON`	Already logged in.
`DM_ARGS_NEEDED`	Arguments required.

Table 12-3 @dmretcode error codes and corresponding messages ***(Continued)***

Code constant	Message
`DM_BAD_ARGS`	Bad arguments.
`DM_BAD_CMD`	Bad command.
`DM_BIND_COUNT`	Incorrect number of bind variables.
`DM_BIND_VAR`	Bad or missing bind variable.
`DM_COMMIT`	Commit failed.
`DM_DESC_COL`	Describe select column error.
`DM_END_OF_PROC`	End of procedure.
`DM_FETCH`	Error during fetch.
`DM_INVALID_DATE`	Invalid date.
`DM_KEYWORD`	Bad or missing keyword.
`DM_LOGON_DENIED`	Logon denied.
`DM_MANY_CURSORS`	Too many cursors.
`DM_NO_CURSOR`	Cursor does not exist.
`DM_NO_MORE_ROWS`	No more rows indicator.
`DM_NO_NAME`	No name specified.
`DM_NO_TRANSACTION`	Transaction does not exist.
`DM_NOCONNECTION`	No connection active.
`DM_NODATABASE`	No database selected.
`DM_NOTLOGGEDON`	Not logged in.
`DM_NOTSUPPORTED`	Command not supported for specified engine.
`DM_PARSE_ERROR`	SQL parse error.
`DM_ROLLBACK`	Rollback failed.

Table 12-3 @dmretcode error codes and corresponding messages *(Continued)*

Code constant	Message
`DM_TRAN_PEND`	Transaction pending.

Example

```
proc entry
DBMS ONERROR JPL dbi_errhandle
DBMS ONEXIT JPL dbi_exithandle
...
return

proc dbi_errhandle (stmt, engine, flag)
# Check for logon errors.
if @dmretcode == DM_ALREADY_ON
   return 0

else if @dmretcode == DM_LOGON_DENIED
   msg emsg @dmretmsg "%N" @dmengerrmsg
....
return 1

proc dbi_exithandle (stmt, engine, flag)
if @dmretcode == DM_NO_MORE_ROWS
   msg emsg "All rows returned."
return 0
```

@dmretmsg

Contains an engine-independent error or status message

Description

`@dmretmsg` is cleared before Panther executes a new `DBMS` command. If the command fails because of an error detected either by the engine or by Panther's data base driver, Panther writes an error message to `@dmretmsg` describing the failure. These messages are defined in `dmerror.h` and are engine-independent. Refer to Table 12-3 on page 12-11 for a listing of the codes and messages.

If `@dmretcode` is `0`, `@dmretmsg` is always empty.

Example

```
proc dbi_errhandle (stmt, engine, flag)
   msg emsg "Statement " stmt " failed." "%N"\
       @dmretmsg "%N" @dmengerrmsg
   return 1
```

@dmrowcount

Contains a count of the number of rows either fetched to Panther or affected by the previous statement

Description

The use of this variable is dependent on the database engine. On all engines, `@dmrowcount` is set to the number of rows fetched to Panther variables in a `SELECT` statement or `CONTINUE` command. On some engines, it can also reflect the number of rows affected by an `INSERT`, `UPDATE`, or `DELETE` statement.

`@dmrowcount` is set to 0 before each new `DBMS` command is executed. You must copy its value to another location if you want to use the value after subsequent commands.

If the command fetches rows, Panther updates `@dmrowcount` writing the number of rows fetched to Panther variables. Most SQL syntaxes provide an aggregate function `COUNT` to count the number of values in a column or the number of rows in a select set. The value of `@dmrowcount` is not the number of rows in a select set; rather, it is the number of rows returned to Panther variables. Therefore if a select set has 14 rows in total, and its target Panther variables are arrays, each with ten occurrences, `@dmrowcount` is 10 after the `SELECT` is executed, and 4 after `CONTINUE` is executed. If `CONTINUE` is executed a second time, `@dmrowcount` would equal 0.

The value of `@dmrowcount` is either less than or equal to the maximum number of rows that can be written to the target Panther destinations; the maximum number of rows is the number of occurrences in a destination variable. If the value in `@dmrowcount` is less than the maximum number of occurrences, then the entire select set is written to the target variables and no further processing is needed. If `@dmrowcount` equals the maximum number of occurrences, then the `SELECT` might fetch more rows than can fit in the variables. To display the rest of the select set, the application must execute `DBMS CONTINUE` until `@dmrowcount` is less than the maximum number of occurrences (or equals 0) or until `@dmretcode` receives the `DM_NO_MORE_ROWS` code.

For information on whether the variable can be used to obtain the number of rows affected by an `INSERT`, `UPDATE`, or `DELETE` statement, refer to the *Database Drivers* for the specified engine.

If you are using the transaction manager, call `sm_tm_inquire(TM_OCC_COUNT)` to find the number of rows fetched in the current server view. Since a transaction command can consist of more than one `DBMS` command, `@dmrowcount` might have already been overwritten.

Example

```
proc get_selection
   DBMS QUERY SELECT * FROM titles WHERE genre_code=:+type
   call check_count
   return

proc check_count
 # If rows are returned but not the NO_MORE_ROWS code,
 # let the user know there are rows pending.
   if (@dmrowcount > 0) && \
      (@dmretcode != DM_NO_MORE_ROWS)
      msg setbkstat "Press %KPF1 to see more."
   else
      msg setbkstat "All rows returned."
   return

proc get_more
 # This function is called by pressing PF1.
 # It retrieves the next set of rows.
   DBMS CONTINUE
   call check_count
   return
```

@dmserial

Contains a serial column value after performing INSERT

Description

Some engines supply the data type serial to assist applications that need to assign a unique numeric value to each row in a table. When an application inserts a row in a table with a serial column, the engine generates a serial number, inserts the row with the number, and returns the number to the application. Refer to the *Database Drivers* for information about support for this on your engine.

Before executing a new DBMS command, Panther writes a 0 to @dmserial. If the statement is an INSERT and the engine returns a serial value, Panther writes the value to @dmserial. Since this variable is cleared before executing a new DBMS command, you must copy its value to another location if you want to use the value in subsequent commands.

Example

```
proc new_order
vars i(3), order_id(5)
DBMS BEGIN
# First INSERT row into invoices table.
# Column order_id in table invoices is a SERIAL.
   DBMS RUN INSERT INTO invoices \
      (order_id, order_date, cust_num) VALUES \
      (0, :+today, :+cust_num)
# Copy the serial value to a JPL variable for use with
# subsequent INSERTS.
order_id = @dmserial

# Use order number to insert new rows to the orders
# table. Column order_id in table orders is an INT.
   for i=1 while i<=max step 1

   DBMS RUN INSERT INTO orders \
      (order_id, part_id, quant, u_cost) VALUES \
      (:order_id, :+part_id[i], :+quant[i], :+u_cost[i])
DBMS COMMIT

msg emsg "Order completed. Invoice number is " order_id
   return
```

@dmwarnsqlstate

Contains an engine-specific status code indicating a warning

Description

Some database engines support a `SQLSTATE` status code which is updated after each SQL statement. `SQLSTATE` is a five-character string which can be set for warning or error conditions. In Panther, warning conditions from `SQLSTATE` are written to `@dmwarnsqlstate`; error conditions are written to `@dmerrsqlstate`.

By default, Panther ignores warnings. If an application needs to alert users to warning codes, it must use a JPL or C function to check for them. There is no default warning handler. The most efficient way to process warning codes is with an installed exit handler using `ONEXIT`.

If the database engine does not support `SQLSTATE`, the value of `@dmwarnsqlstate` will be `"00000"`, the value that represents success.

If the application accesses multiple database engines, the database driver for each engine must support `SQLSTATE` in order to use its values for application process ing.`@dmwarnsqlstate` is set to `"00000"` before each `DBMS` statement. If you need its value for further processing, it should be copied to another variable.

CHAPTER

13 Keywords in Database Drivers

This chapter lists the keywords for Panther's database drivers. Avoid using these keywords as identifiers, particularly for cursors, connections, engines, and transactions. Also, avoid using these keywords when naming Panther variables which will be used in a DBMS statement. Since keywords are not case-sensitive, the following two statements are equivalent:

```
dbms close_all_connections
DBMS CLOSE_ALL_CONNECTIONS
```

Table 13-1 Keywords in the database drivers

alias	application	autocommit
begin	binary	browse
call	cancel	catalog_function
catquery	checkpt_interval	close
close_all_connections	column_names	commit
completion	conn_string	connect
connected	connection	continue
continue_bottom	continue_down	continue_top

Table 13-1 Keywords in the database drivers *(Continued)*

continue_up	create_proc	create_trigger
count	ct_command	ct_cursor
current	cursor	cursors
database	datasource	db
dbms	declare	disconnect
drop_proc	drop_trigger	
end	engine	error
error_continue	exec	execute
execute_all		
flush	file	for
format		
heading	host	
interfaces		
jpl		
locklevel	locktimeout	logon
logoff		

Table 13-1 Keywords in the database drivers *(Continued)*

max		
next	null	
occur	off	on
onentry	onerror	onexit
options	out	output
parsing_mode	password	prepare_commit
print	proc	proc_control
query		
redirect	return	retvar
rfjournal	rollback	rpc
run	run_default	
save	schema	select
select_aliases	separator	serial
server	set	set_buffer
single_step	sql	sqltimeout
start	stop	stop_at_fetch
store	supreps	

Table 13-1 Keywords in the database drivers *(Continued)*

tee	timeout	to
tranid	transaction	transport
type		
unique	update	use
user	using	
warn	width	

CHAPTER

14 ActiveX Controls

This chapter contains descriptions of the ActiveX Controls distributed with Panther. Information about each ActiveX control is organized into the following sections:

- Description
- Properties
- Methods
- Events

PrlSpinner

Enter a numeric value within a specified range

Description	The Panther Spinner ActiveX Control is a text entry box combined with an up-down control (up arrow/down arrow button). Enter ordered, numeric values into the control directly or increment/decrement the value by the buttons. Additionally, properties can be set to control the maximum or minimum of the value. The control can be installed using the file `$SMBASE\Samples\Activex\PrlSpinner.ocx.`
CLSID	`DA2599CE-F939-11D0-A19E-00A02481A2E9`

Properties	`Maximum`
	Set to any integer, defaults to 100.
	`Minimum`
	Set to any integer, defaults to 0.
	`BlankIfZero`
	Set to yes/no.
	`Value`
	Set to its numeric content. This is the default property.

Events	None.
Methods	None.

Index